



Reference Documentation

Version: 4.2.0

provided by



Table of Contents

Preface	xvi
1. Technology Overview	1
1.1. Introduction to CEP and event stream analysis	1
1.2. CEP and relational databases	1
1.3. The Esper engine for CEP	1
1.4. Required 3rd Party Libraries	2
2. Event Representations	3
2.1. Event Underlying Objects	3
2.2. Event Properties	4
2.2.1. Escape Characters	4
2.3. Dynamic Event Properties	5
2.4. Fragment and Fragment Type	6
2.5. Object Events	6
2.5.1. Object Event Properties	7
2.5.2. Property Names	8
2.5.3. Constants and Enumeration	9
2.5.4. Parameterized Types	9
2.5.5. Known Limitations	10
2.6. Map Events	10
2.6.1. Overview	10
2.6.2. Map Properties	10
2.6.3. Map Supertypes	11
2.6.4. Advanced Map Property Types	12
2.6.4.1. Nested Properties	12
2.6.4.2. Map Event Type Properties	13
2.6.4.3. One-to-Many Relationships	13
2.7. System.Xml.XmlNode XML Events	14
2.7.1. Schema-Provided XML Events	15
2.7.1.1. Getting Started	15
2.7.1.2. Property Expressions and Namespaces	16
2.7.1.3. Property Expression to XPath Rewrite	16
2.7.1.4. Array Properties	17
2.7.1.5. Dynamic Properties	17
2.7.1.6. Transposing Properties	17
2.7.1.7. Event Sender	18
2.7.2. No-Schema-Provided XML Events	18
2.7.3. Explicitly-Configured Properties	19
2.7.3.1. Simple Explicit Property	19
2.7.3.2. Explicit Property Casting and Parsing	19
2.7.3.3. Node and Nodeset Explicit Property	19
2.8. Additional Event Representations	20
2.9. Updating, Merging and Versioning Events	20
2.10. Coarse-Grained Events	21
2.11. Event Objects Populated by Insert Into	21
3. Processing Model	23
3.1. Introduction	23
3.2. Insert Stream	23
3.3. Insert and Remove Stream	24

3.4. Filters and Where-clauses	25
3.5. Time Windows	27
3.5.1. Time Window	27
3.5.2. Time Batch	28
3.6. Batch Windows	29
3.7. Aggregation and Grouping	30
3.7.1. Insert and Remove Stream	30
3.7.2. Output for Aggregation and Group-By	30
3.7.2.1. Un-aggregated and Un-grouped	31
3.7.2.2. Fully Aggregated and Un-grouped	31
3.7.2.3. Aggregated and Un-Grouped	31
3.7.2.4. Fully Aggregated and Grouped	32
3.7.2.5. Aggregated and Grouped	32
3.8. Event Visibility and Current Time	32
4. EPL Reference: Clauses	33
4.1. EPL Introduction	33
4.2. EPL Syntax	34
4.2.1. Specifying Time Periods	34
4.2.2. Using Comments	35
4.2.3. Reserved Keywords	35
4.2.4. Escaping Strings	36
4.2.5. Data Types	36
4.2.5.1. Data Type of Constants	37
4.2.6. Annotation	38
4.2.6.1. Application-Provided Annotations	38
4.2.6.2. Built-In Annotations	39
4.2.6.3. @Name	40
4.2.6.4. @Description	40
4.2.6.5. @Tag	40
4.2.6.6. @Priority	40
4.2.6.7. @Drop	41
4.2.6.8. @Hint	41
4.2.6.9. @Hook	41
4.2.6.10. @Audit	41
4.2.7. Expression Declaration	41
4.3. Choosing Event Properties And Events: the Select Clause	43
4.3.1. Choosing all event properties: select *	43
4.3.2. Choosing specific event properties	44
4.3.3. Expressions	44
4.3.4. Renaming event properties	44
4.3.5. Choosing event properties and events in a join	45
4.3.6. Choosing event properties and events from a pattern	46
4.3.7. Selecting insert and remove stream events	46
4.3.8. Qualifying property names and stream names	47
4.3.9. Select Distinct	47
4.4. Specifying Event Streams: the From Clause	48
4.4.1. Filter-based Event Streams	48
4.4.1.1. Specifying an Event Type	49
4.4.1.2. Specifying Filter Criteria	49
4.4.1.3. Filtering Ranges	50
4.4.1.4. Filtering Sets of Values	51
4.4.1.5. Filter Limitations	51

4.4.2. Pattern-based Event Streams	51
4.4.3. Specifying Views	52
4.4.4. Multiple Data Window Views	52
4.4.5. Using the Stream Name	53
4.5. Specifying Search Conditions: the Where Clause	54
4.6. Aggregates and grouping: the Group-by Clause and the Having Clause	54
4.6.1. Using aggregate functions	54
4.6.2. Organizing statement results into groups: the Group-by clause	55
4.6.3. Selecting groups of events: the Having clause	57
4.6.4. How the stream filter, Where, Group By and Having clauses interact	58
4.6.5. Comparing the Group By clause and the std:groupwin view	59
4.7. Stabilizing and Controlling Output: the Output Clause	59
4.7.1. Output Clause Options	59
4.7.1.1. Controlling Output Using an Expression	61
4.7.1.2. Suppressing Output With After	62
4.7.2. Aggregation, Group By, Having and Output clause interaction	63
4.7.3. Runtime Considerations	64
4.8. Sorting Output: the Order By Clause	64
4.9. Limiting Row Count: the Limit Clause	65
4.10. Merging Streams and Continuous Insertion: the Insert Into Clause	66
4.10.1. Transposing a Property To a Stream	67
4.10.2. Merging Streams By Event Type	67
4.10.3. Merging Disparate Types of Events: Variant Streams	68
4.10.4. Decorated Events	69
4.10.5. Event as a Property	69
4.10.6. Populating an Underlying Event Object	69
4.11. Subqueries	70
4.11.1. The 'exists' Keyword	72
4.11.2. The 'in' and 'not in' Keywords	72
4.11.3. The 'any' and 'some' Keywords	72
4.11.4. The 'all' Keyword	73
4.11.5. Multi-Column Selection	74
4.11.6. Multi-Row Selection	74
4.11.7. Hints Related to Subqueries	75
4.12. Joining Event Streams	76
4.12.1. Introducing Joins	76
4.12.2. Inner (Default) Joins	77
4.12.3. Outer, Left and Right Joins	78
4.12.4. Unidirectional Joins	79
4.12.5. Hints Related to Joins	80
4.13. Accessing Relational Data via SQL	81
4.13.1. Joining SQL Query Results	82
4.13.2. SQL Query and the EPL Where Clause	83
4.13.3. Outer Joins With SQL Queries	84
4.13.4. Using Patterns to Request (Poll) Data	84
4.13.5. Polling SQL Queries via Enumerator	84
4.13.6. ADO.NET Implementation Overview	85
4.13.7. No-Metadata Workaround	85
4.13.8. SQL Input Parameter and Column Output Conversion	86
4.13.9. SQL Row Object Conversion	86
4.14. Accessing Non-Relational Data via Method Invocation	86
4.14.1. Joining Method Invocation Results	87

4.14.2. Polling Method Invocation Results via Iterator	88
4.14.3. Providing the Method	88
4.14.4. Using a Map Return Type	89
4.15. Creating and Using Named Windows	90
4.15.1. Creating Named Windows: the Create Window clause	90
4.15.1.1. Creation by Modelling after an Existing Type	91
4.15.1.2. Creation By Defining Columns Names and Types	92
4.15.1.3. Dropping or Removing Named Windows	92
4.15.2. Inserting Into Named Windows	93
4.15.2.1. Named Windows Holding Decorated Events	94
4.15.2.2. Named Windows Holding Events As Property	94
4.15.3. Selecting From Named Windows	94
4.15.4. Triggered Select on Named Windows: the On Select clause	95
4.15.5. Triggered Playback from Named Windows: the On Insert clause	97
4.15.6. Populating a Named Window from an Existing Named Window	98
4.15.7. Updating Named Windows: the On Update clause	98
4.15.8. Deleting From Named Windows: the On Delete clause	100
4.15.8.1. Using Patterns in the On Delete Clause	101
4.15.9. Triggered Upsert using the On-Merge Clause	101
4.15.10. Explicitly Indexing Named Windows	104
4.15.11. Versioning and Revision Event Type Use with Named Windows	105
4.16. Declaring an Event Type: Create Schema	107
4.16.1. Declare an Event Type by Providing Names and Types	107
4.16.2. Declare an Event Type by Providing a Class Name	108
4.16.3. Declare a Variant Stream	108
4.17. Splitting and Duplicating Streams	109
4.18. Variables	110
4.18.1. Creating Variables: the Create Variable clause	110
4.18.2. Setting Variable Values: the On Set clause	111
4.18.3. Using Variables	112
4.18.4. Object-Type Variables	113
4.18.5. Class and Event-Type Variables	113
4.19. Contained-Event Selection	114
4.19.1. Select Clause in a Contained-Event Selection	116
4.19.2. Where Clause in a Contained-Event Selection	117
4.19.3. Contained-Event Selection and Joins	117
4.20. Updating an Insert Stream: the Update IStream Clause	119
4.20.1. Immutability and Updates	120
4.21. Controlling Event Delivery : The For Clause	121
5. EPL Reference: Patterns	123
5.1. Event Pattern Overview	123
5.2. How to use Patterns	123
5.2.1. Pattern Syntax	124
5.2.2. Patterns in EPL	124
5.2.3. Subscribing to Pattern Events	125
5.2.4. Pulling Data from Patterns	125
5.2.5. Pattern Error Reporting	126
5.3. Operator Precedence	126
5.4. Filter Expressions In Patterns	127
5.5. Pattern Operators	128
5.5.1. Every	128
5.5.1.1. Limiting Subexpression Lifetime	130

5.5.1.2. Every Operator Example	131
5.5.1.3. Sensor Example	132
5.5.2. Every-Distinct	132
5.5.3. Repeat	133
5.5.4. Repeat-Until	134
5.5.4.1. Unbound Repeat	135
5.5.4.2. Bound Repeat Overview	135
5.5.4.3. Bound Repeat - Open Ended Range	136
5.5.4.4. Bound Repeat - High Endpoint Range	136
5.5.4.5. Bound Repeat - Bounded Range	136
5.5.4.6. Tags and the Repeat Operator	136
5.5.5. And	137
5.5.6. Or	138
5.5.7. Not	138
5.5.8. Followed-by	139
5.5.8.1. Limiting Sub-Expression Count	139
5.5.9. Pattern Guards	140
5.5.9.1. The timer:within Pattern Guard	141
5.5.9.2. The timer:withinmax Pattern Guard	141
5.5.9.3. The while Pattern Guard	142
5.5.9.4. Guard Time Interval Expressions	143
5.5.9.5. Combining Guard Expressions	143
5.6. Pattern Atoms	143
5.6.1. Filter Atoms	143
5.6.2. Time-based Observer Atoms	143
5.6.2.1. timer:interval	143
5.6.2.2. timer:at	144
6. EPL Reference: Match Recognize	147
6.1. Overview	147
6.2. Comparison of Match Recognize and EPL Patterns	147
6.3. Syntax	148
6.3.1. Syntax Example	149
6.4. Pattern and Pattern Operators	150
6.4.1. Operator Precedence	150
6.4.2. Concatenation	150
6.4.3. Alternation	151
6.4.4. Quantifiers Overview	151
6.4.5. Variables Can be Singleton or Group	152
6.4.5.1. Additional Aggregation Functions	152
6.4.6. Eliminating Duplicate Matches	153
6.4.7. Greedy Or Reluctant	153
6.4.8. Quantifier - One Or More (+ and +?)	154
6.4.9. Quantifier - Zero Or More (* and *?)	155
6.4.10. Quantifier - Zero Or One (? and ??)	155
6.5. Define Clause	156
6.5.1. The Prev Operator	156
6.6. Measure Clause	157
6.7. Datwindow-Bound	157
6.8. Interval	158
6.9. Limitations	159
7. EPL Reference: Operators	160
7.1. Arithmetic Operators	160

7.2. Logical And Comparison Operators	160
7.3. Concatenation Operators	160
7.4. Binary Operators	161
7.5. Array Definition Operator	161
7.6. Dot Operator	162
7.6.1. Duck Typing	162
7.7. The 'in' Keyword	162
7.8. The 'between' Keyword	163
7.9. The 'like' Keyword	164
7.10. The 'regex' Keyword	164
7.11. The 'any' and 'some' Keywords	165
7.12. The 'all' Keyword	166
8. EPL Reference: Functions	167
8.1. Single-row Function Reference	167
8.1.1. The Case Control Flow Function	169
8.1.2. The Cast Function	169
8.1.3. The Coalesce Function	170
8.1.4. The Current_Timestamp Function	170
8.1.5. The Exists Function	170
8.1.6. The Instance-Of Function	170
8.1.7. The Min and Max Functions	171
8.1.8. The Previous Function	171
8.1.8.1. Restrictions	173
8.1.8.2. Comparison to the prior Function	173
8.1.9. The Previous-Tail Function	173
8.1.9.1. Restrictions	174
8.1.10. The Previous-Window Function	174
8.1.10.1. Restrictions	175
8.1.11. The Previous-Count Function	175
8.1.11.1. Restrictions	176
8.1.12. The Prior Function	176
8.1.13. The Type-Of Function	176
8.2. Aggregate Functions	177
8.2.1. SQL-Standard Functions	178
8.2.2. Data Window Aggregation Functions	179
8.2.2.1. First Aggregation Function	179
8.2.2.2. Last Aggregation Function	180
8.2.2.3. Window Aggregation Function	181
8.2.3. Additional Aggregation Functions	181
8.3. User-Defined Functions	183
9. EPL Reference: Enumeration Methods	186
9.1. Overview	186
9.2. Example Events	190
9.3. How to Use	191
9.3.1. Syntax	191
9.3.2. Introductory Examples	192
9.3.3. Input, Output and Limitations	192
9.4. Inputs	193
9.4.1. Subquery Results	193
9.4.2. Named Window	194
9.4.3. Event Property	195
9.4.4. Data Window Aggregation	195

9.4.5. prev, prevwindow and prevtail Single-Row Functions as Input	196
9.4.6. Single-Row Function or UDF	196
9.4.7. Declared Expression	197
9.5. Example	197
9.6. Reference	198
9.6.1. Aggregate	198
9.6.2. AllOf	199
9.6.3. AnyOf	199
9.6.4. Average	199
9.6.5. CountOf	200
9.6.6. Except	200
9.6.7. FirstOf	201
9.6.8. GroupBy	201
9.6.9. Intersect	202
9.6.10. LastOf	202
9.6.11. LeastFrequent	202
9.6.12. Max	203
9.6.13. MaxBy	203
9.6.14. Min	204
9.6.15. MinBy	204
9.6.16. MostFrequent	205
9.6.17. OrderBy and OrderByDesc	205
9.6.18. Reverse	206
9.6.19. SelectFrom	206
9.6.20. SequenceEqual	206
9.6.21. SumOf	207
9.6.22. Take	207
9.6.23. TakeLast	207
9.6.24. TakeWhile	208
9.6.25. TakeWhileLast	208
9.6.26. ToMap	209
9.6.27. Union	209
9.6.28. Where	209
10. EPL Reference: Date-Time Methods	211
10.1. Overview	211
10.2. How to Use	214
10.2.1. Syntax	214
10.3. Calendar and Formatting Reference	215
10.3.1. Between	215
10.3.2. Format	216
10.3.3. Get (By Field)	216
10.3.4. Get (By Name)	217
10.3.5. Minus	217
10.3.6. Plus	217
10.3.7. RoundCeiling	217
10.3.8. RoundFloor	218
10.3.9. RoundHalf	218
10.3.10. Set (By Field)	218
10.3.11. WithDate	219
10.3.12. WithMax	219
10.3.13. WithMin	219
10.3.14. WithTime	219

10.3.15. ToCalendar	220
10.3.16. ToDate	220
10.3.17. ToMillisec	220
10.4. Interval Algebra Reference	220
10.4.1. After	222
10.4.2. Before	222
10.4.3. Coincides	223
10.4.4. During	224
10.4.5. Finishes	225
10.4.6. Finished By	225
10.4.7. Includes	226
10.4.8. Meets	227
10.4.9. Met By	227
10.4.10. Overlaps	228
10.4.11. Overlapped By	228
10.4.12. Starts	229
10.4.13. Started By	230
11. EPL Reference: Views	231
11.1. Window views	233
11.1.1. Length window (win:length)	233
11.1.2. Length batch window (win:length_batch)	234
11.1.3. Time window (win:time)	234
11.1.4. Externally-timed window (win:ext_timed)	234
11.1.5. Time batch window (win:time_batch)	235
11.1.6. Time-Length combination batch window (win:time_length_batch)	236
11.1.7. Time-Accumulating window (win:time_accum)	237
11.1.8. Keep-All window (win:keepall)	237
11.1.9. First Length (win:firstlength)	238
11.1.10. First Time (win:firsttime)	238
11.2. Standard view set	238
11.2.1. Unique (std:unique)	238
11.2.2. Grouped Data Window (std:groupwin)	239
11.2.3. Size (std:size)	240
11.2.4. Last Event (std:lastevent)	241
11.2.5. First Event (std:firstevent)	242
11.2.6. First Unique (std:firstunique)	242
11.3. Statistics views	242
11.3.1. Univariate statistics (stat:uni)	242
11.3.2. Regression (stat:linest)	243
11.3.3. Correlation (stat:correl)	244
11.3.4. Weighted average (stat:weighted_avg)	245
11.4. Extension View Set	246
11.4.1. Sorted Window View (ext:sort)	246
11.4.2. Time-Order View (ext:time_order)	246
12. API Reference	248
12.1. API Overview	248
12.2. The Service Provider Interface	248
12.3. The Administrative Interface	249
12.3.1. Creating Statements	249
12.3.2. Receiving Statement Results	250
12.3.3. Setting a Subscriber Object	252
12.3.3.1. Row-By-Row Delivery	252

12.3.3.2. Multi-Row Delivery	254
12.3.4. Adding Event Handlers	255
12.3.4.1. Subscription Snapshot and Atomic Delivery	255
12.3.5. Using Enumerators	255
12.3.6. Managing Statements	257
12.3.7. Runtime Configuration	257
12.4. The Runtime Interface	257
12.4.1. Event Sender	258
12.4.2. Receiving Unmatched Events	258
12.4.3. On-Demand Snapshot Query Execution	259
12.4.3.1. On-Demand Query API	259
12.5. Event and Event Type	260
12.5.1. Event Type Metadata	260
12.5.2. Event Object	261
12.5.3. Query Example	261
12.5.4. Pattern Example	262
12.6. Engine Threading and Concurrency	263
12.6.1. Advanced Threading	265
12.6.1.1. Inbound Threading	266
12.6.1.2. Outbound Threading	266
12.6.1.3. Timer Execution Threading	266
12.6.1.4. Route Execution Threading	266
12.6.1.5. Threading Service Provider Interface	266
12.6.2. Processing Order	267
12.6.2.1. Competing Statements	267
12.6.2.2. Competing Events in a Work Queue	267
12.7. Controlling Time-Keeping	267
12.7.1. Controlling Time Using Time Span Events	269
12.7.2. Additional Time-Related APIs	270
12.8. Time Resolution	270
12.9. Service Isolation	271
12.9.1. Overview	271
12.9.2. Example: Suspending a Statement	272
12.9.3. Example: Catching up a Statement from Historical Data	272
12.9.4. Isolation for Insert-Into	273
12.9.5. Isolation for Named Windows	273
12.9.6. Runtime Considerations	274
12.10. Exception Handling	274
12.11. Condition Handling	274
12.12. Statement Object Model	275
12.12.1. Building an Object Model	275
12.12.2. Building Expressions	276
12.12.3. Building a Pattern Statement	277
12.12.4. Building a Select Statement	277
12.12.5. Building a Create-Variable and On-Set Statement	278
12.12.6. Building Create-Window, On-Delete and On-Select Statements	278
12.13. Prepared Statement and Substitution Parameters	279
12.14. Engine and Statement Metrics Reporting	280
12.14.1. Engine Metrics	281
12.14.2. Statement Metrics	281
12.15. Event Rendering to XML and JSON	282
12.15.1. JSON Event Rendering Conventions and Options	282

12.15.2. XML Event Rendering Conventions and Options	283
12.16. Plug-in Loader	283
12.17. Interrogating EPL Annotations	284
12.18. CLR-Java Differences	284
13. Configuration	285
13.1. Programmatic Configuration	285
13.2. Configuration via XML File	285
13.3. XML Configuration File	286
13.4. Configuration Items	286
13.4.1. Events represented by CLR types	286
13.4.1.1. Package of CLR Event Classes	286
13.4.1.2. Event type name to CLR type mapping	286
13.4.1.3. Legacy Event Classes	287
13.4.1.4. Specifying Event Properties for CLR Types	288
13.4.1.5. Turning off Code Generation	288
13.4.1.6. Case Sensitivity and Property Names	289
13.4.1.7. Factory and Copy Method	289
13.4.2. Events represented by System.Collection.Generic.IDictionary	290
13.4.3. Events represented by System.Xml.XmlNode	291
13.4.3.1. Schema Resource	292
13.4.3.2. Explicit XPath Property	292
13.4.3.3. Absolute or Deep Property Resolution	293
13.4.3.4. XPath Variable and Function Resolver	293
13.4.3.5. Auto Fragment	293
13.4.3.6. XPath Property Expression	293
13.4.3.7. Event Sender Setting	294
13.4.4. Events represented by Plug-in Event Representations	294
13.4.4.1. Enabling an Custom Event Representation	294
13.4.4.2. Adding Plug-in Event Types	294
13.4.4.3. Setting Resolution URIs	295
13.4.5. Class and package imports	295
13.4.6. Cache Settings for From-Clause Method Invocations	295
13.4.7. Variables	296
13.4.8. Relational Database Access	296
13.4.8.1. Connections obtained via DataSource	297
13.4.8.2. Connections obtained via DataSource Factory	297
13.4.8.3. Connections obtained via DriverManager	298
13.4.8.4. Connections-level settings	299
13.4.8.5. Connections lifecycle settings	299
13.4.8.6. Cache settings	299
13.4.8.7. Column Change Case	300
13.4.8.8. SQL Types Mapping	301
13.4.8.9. Metadata Origin	301
13.4.9. Engine Settings related to Concurrency and Threading	301
13.4.9.1. Preserving the order of events delivered to listeners	301
13.4.9.2. Preserving the order of events for insert-into streams	302
13.4.9.3. Internal Timer Settings	303
13.4.9.4. Advanced Threading Options	303
13.4.10. Engine Settings related to Event Metadata	304
13.4.10.1. Type Property Names, Case Sensitivity and Accessor Style	304
13.4.11. Engine Settings related to View Resources	304
13.4.11.1. Sharing View Resources between Statements	304

13.4.11.2. Configuring Multi-Expiry Policy Defaults	304
13.4.12. Engine Settings related to Logging	305
13.4.12.1. Execution Path Debug Logging	305
13.4.12.2. Query Plan Logging	305
13.4.12.3. JDBC Logging	306
13.4.13. Engine Settings related to Variables	306
13.4.13.1. Variable Version Release Interval	306
13.4.14. Engine Settings related to Stream Selection	307
13.4.14.1. Default Statement Stream Selection	307
13.4.15. Engine Settings related to Time Source	307
13.4.15.1. Default Time Source	307
13.4.16. Engine Settings related to Metrics Reporting	308
13.4.17. Engine Settings related to Language and Locale	309
13.4.18. Engine Settings related to Expression Evaluation	309
13.4.18.1. Integer Division and Division by Zero	309
13.4.18.2. Subselect Evaluation Order	310
13.4.18.3. User-Defined Function or Static Method Cache	310
13.4.18.4. Extended Built-in Aggregation Functions	311
13.4.18.5. Duck Typing	311
13.4.19. Engine Settings related to Execution of Statements	311
13.4.19.1. Prioritized Execution	311
13.4.20. Engine Settings related to Exception Handling	312
13.4.21. Engine Settings related to Condition Handling	312
13.4.22. Revision Event Type	313
13.4.23. Variant Stream	314
13.5. Type Names	315
13.6. Runtime Configuration	315
13.7. Logging Configuration	316
13.7.1. Log4net Logging Configuration	316
14. Development Lifecycle	317
14.1. Authoring	317
14.2. Testing	317
14.3. Debugging	317
14.3.1. @Audit Annotation	317
14.4. Packaging and Deploying Overview	318
14.5. EPL Modules	319
14.6. The Deployment Administrative Interface	320
14.6.1. Reading Module Content	320
14.6.2. Ordering Multiple Modules	321
14.6.3. Deploying and Undeploying	321
14.6.4. Listing Deployments	321
14.6.5. State Transitioning a Module	321
14.6.6. Best Practices	322
14.7. J2EE Packaging and Deployment	322
14.7.1. J2EE Deployment Considerations	322
14.7.2. Servlet Context Listener	323
15. Extension and Plug-in	325
15.1. Overview	325
15.2. Virtual Data Window	325
15.2.1. How to Use	326
15.2.1.1. Query Access Path	326
15.2.2. Implementing the Factory	327

15.2.3. Implementing the Virtual Data Window	328
15.2.4. Implementing the Lookup	329
15.3. Single-Row Functions	330
15.3.1. Implementing a Single-Row Function	330
15.3.2. Configuring the Single-Row Function Name	330
15.4. Derived-value and Data Window View	331
15.4.1. Implementing a View Factory	331
15.4.2. Implementing a View	333
15.4.3. View Contract	333
15.4.4. Configuring View Namespace and Name	334
15.4.5. Requirement for Data Window Views	334
15.4.6. Requirement for Grouped Views	334
15.5. Aggregation Functions	335
15.5.1. Implementing an Aggregation Function	335
15.5.2. Configuring the Aggregation Function Name	337
15.5.3. Accepting Multiple Parameters	337
15.6. Pattern Guard	338
15.6.1. Implementing a Guard Factory	338
15.6.2. Implementing a Guard Class	339
15.6.3. Configuring Guard Namespace and Name	339
15.7. Pattern Observer	340
15.7.1. Implementing an Observer Factory	340
15.7.2. Implementing an Observer Class	341
15.7.3. Configuring Observer Namespace and Name	342
15.8. Event Type And Event Object	342
15.8.1. How It Works	342
15.8.2. Steps	343
15.8.3. URI-based Resolution	343
15.8.4. Example	344
15.8.4.1. Sample Event Type	344
15.8.4.2. Sample Event Bean	345
15.8.4.3. Sample Event Representation	346
15.8.4.4. Sample Event Bean Factory	347
16. Examples, Tutorials, Case Studies	349
16.1. Examples Overview	349
16.2. Running the Examples	350
16.3. AutoID RFID Reader	351
16.4. Runtime Configuration	351
16.5. Market Data Feed Monitor	352
16.5.1. Input Events	352
16.5.2. Computing Rates Per Feed	352
16.5.3. Detecting a Fall-off	352
16.5.4. Event generator	352
16.6. OHLC Plug-in View	353
16.7. Transaction 3-Event Challenge	353
16.7.1. The Events	353
16.7.2. Combined event	354
16.7.3. Real time summary data	354
16.7.4. Find problems	354
16.7.5. Event generator	354
16.8. Self-Service Terminal	355
16.8.1. Events	355

16.8.2. Detecting Customer Check-in Issues	355
16.8.3. Absence of Status Events	355
16.8.4. Activity Summary Data	356
16.9. Assets Moving Across Zones - An RFID Example	356
16.10. StockTicker	357
16.11. MatchMaker	357
16.12. Named Window Query	357
16.13. Quality of Service	357
17. Performance	359
17.1. Performance Results	359
17.2. Performance Tips	359
17.2.1. Understand how to tune your application	359
17.2.2. Compare Esper to other solutions	360
17.2.3. Input and Output Bottlenecks	360
17.2.4. Advanced Threading	361
17.2.5. Select the underlying event rather than individual fields	361
17.2.6. Prefer stream-level filtering over post-data-window filtering	361
17.2.7. Reduce the use of arithmetic in expressions	362
17.2.8. Remove Unnecessary Constructs	362
17.2.9. End Pattern Sub-Expressions	362
17.2.10. Consider using EventPropertyGetter for fast access to event properties	362
17.2.11. Consider casting the underlying event	363
17.2.12. Turn off logging	363
17.2.13. Disable view sharing	363
17.2.14. Tune or disable delivery order guarantees	364
17.2.15. Use a Subscriber Object to Receive Events	364
17.2.16. High-Arrival-Rate Streams and Single Statements	364
17.2.17. Joins And Where-clause And Data Windows	365
17.2.18. Patterns and Pattern Sub-Expression Instances	366
17.2.19. The Keep-All Data Window	366
17.2.20. Statement Design for Reduced Memory Consumption	366
17.2.21. Performance, JVM, OS and hardware	367
17.2.22. Consider using Hints	367
17.2.23. Optimizing Stream Filter Expressions	368
17.3. Using the performance kit	368
17.3.1. How to use the performance kit	368
17.3.2. How we use the performance kit	371
18. References	372
18.1. Reference List	372
A. Output Reference and Samples	373
A.1. Introduction and Sample Data	373
A.2. Output for Un-aggregated and Un-grouped Queries	374
A.2.1. No Output Rate Limiting	374
A.2.2. Output Rate Limiting - Default	375
A.2.3. Output Rate Limiting - Last	376
A.2.4. Output Rate Limiting - First	377
A.2.5. Output Rate Limiting - Snapshot	378
A.3. Output for Fully-aggregated and Un-grouped Queries	379
A.3.1. No Output Rate Limiting	379
A.3.2. Output Rate Limiting - Default	380
A.3.3. Output Rate Limiting - Last	381
A.3.4. Output Rate Limiting - First	382

A.3.5. Output Rate Limiting - Snapshot	383
A.4. Output for Aggregated and Un-grouped Queries	384
A.4.1. No Output Rate Limiting	384
A.4.2. Output Rate Limiting - Default	385
A.4.3. Output Rate Limiting - Last	386
A.4.4. Output Rate Limiting - First	387
A.4.5. Output Rate Limiting - Snapshot	388
A.5. Output for Fully-aggregated and Grouped Queries	389
A.5.1. No Output Rate Limiting	389
A.5.2. Output Rate Limiting - Default	390
A.5.3. Output Rate Limiting - All	391
A.5.4. Output Rate Limiting - Last	392
A.5.5. Output Rate Limiting - First	393
A.5.6. Output Rate Limiting - Snapshot	394
A.6. Output for Aggregated and Grouped Queries	395
A.6.1. No Output Rate Limiting	395
A.6.2. Output Rate Limiting - Default	396
A.6.3. Output Rate Limiting - All	397
A.6.4. Output Rate Limiting - Last	398
A.6.5. Output Rate Limiting - First	399
A.6.6. Output Rate Limiting - Snapshot	400
B. Reserved Keywords	402
Index	406

Preface

Analyzing and reacting to information in real-time oftentimes requires the development of custom applications. Typically these applications must obtain the data to analyze, filter data, derive information and then indicate this information through some form of presentation or communication. Data may arrive with high frequency requiring high throughput processing. And applications may need to be flexible and react to changes in requirements while the data is processed. Esper is an event stream processor that aims to enable a short development cycle from inception to production for these types of applications.

This document is a resource for software developers who develop event driven applications. It also contains information that is useful for business analysts and system architects who are evaluating Esper.

It is assumed that the reader is familiar with the CLR and the programming languages that can be used to interoperate with the CLR.

This document is relevant in all phases of your software development project: from design to deployment and support.

If you are new to Esper, please follow these steps:

1. Read the tutorials, case studies and solution patterns available on the Esper public web site at <http://esper.codehaus.org>
2. Read Section 1.1, “Introduction to CEP and event stream analysis” if you are new to CEP and ESP (complex event processing, event stream processing)
3. Read Chapter 2, *Event Representations* that explains the different ways of representing events to Esper
4. Read Chapter 3, *Processing Model* to gain insight into EPL continuous query results
5. Read Section 4.1, “EPL Introduction” for an introduction to event stream processing via EPL
6. Read Section 5.1, “Event Pattern Overview” for an overview over event patterns
7. Read Section 6.1, “Overview” for an overview over event patterns using the match recognize syntax.
8. Then glance over the examples Section 16.1, “Examples Overview”
9. Finally to test drive Esper performance, read Chapter 17, *Performance*

Chapter 1. Technology Overview

1.1. Introduction to CEP and event stream analysis

The Esper engine has been developed to address the requirements of applications that analyze and react to events. Some typical examples of applications are:

- Business process management and automation (process monitoring, BAM, reporting exceptions)
- Finance (algorithmic trading, fraud detection, risk management)
- Network and application monitoring (intrusion detection, SLA monitoring)
- Sensor network applications (RFID reading, scheduling and control of fabrication lines, air traffic)

What these applications have in common is the requirement to process events (or messages) in real-time or near real-time. This is sometimes referred to as complex event processing (CEP) and event stream analysis. Key considerations for these types of applications are throughput, latency and the complexity of the logic required.

- High throughput - applications that process large volumes of messages (between 1,000 to 100k messages per second)
- Low latency - applications that react in real-time to conditions that occur (from a few milliseconds to a few seconds)
- Complex computations - applications that detect patterns among events (event correlation), filter events, aggregate time or length windows of events, join event streams, trigger based on absence of events etc.

The Esper engine was designed to make it easier to build and extend CEP applications.

1.2. CEP and relational databases

Relational databases and the standard query language (SQL) are designed for applications in which most data is fairly static and complex queries are less frequent. Also, most databases store all data on disks (except for in-memory databases) and are therefore optimized for disk access.

To retrieve data from a database an application must issue a query. If an application need the data 10 times per second it must fire the query 10 times per second. This does not scale well to hundreds or thousands of queries per second.

Database triggers can be used to fire in response to database update events. However database triggers tend to be slow and often cannot easily perform complex condition checking and implement logic to react.

In-memory databases may be better suited to CEP applications than traditional relational database as they generally have good query performance. Yet they are not optimized to provide immediate, real-time query results required for CEP and event stream analysis.

1.3. The Esper engine for CEP

The Esper engine works a bit like a database turned upside-down. Instead of storing the data and running queries against stored data, the Esper engine allows applications to store queries and run the data through. Response from the Esper engine is real-time when conditions occur that match queries. The execution model is thus continuous rather than only when a query is submitted.

Esper provides two principal methods or mechanisms to process events: event patterns and event stream queries.

Esper offers an event pattern language to specify expression-based event pattern matching. Underlying the pattern matching engine is a state machine implementation. This method of event processing matches expected sequences of presence or absence of events or combinations of events. It includes time-based correlation of events.

Esper also offers event stream queries that address the event stream analysis requirements of CEP applications. Event stream queries provide the windows, aggregation, joining and analysis functions for use with streams of events. These queries are following the EPL syntax. EPL has been designed for similarity with the SQL query language but differs from SQL in its use of views rather than tables. Views represent the different operations needed to structure data in an event stream and to derive data from an event stream.

Esper provides these two methods as alternatives through the same API.

1.4. Required 3rd Party Libraries

Esper requires the following 3rd-party libraries at runtime:

- ANTLR is the parser generator used for parsing and parse tree walking of the pattern and EPL syntax. Credit goes to Terence Parr at <http://www.antlr.org>. The ANTLR license is in the lib directory. The library is required for compile-time only.
- Log4net is a logging API. This open source software is under the Apache license. The Apache 2.0 license is in the lib directory.

Esper requires the following 3rd-party libraries at compile-time and for running the test suite:

- NUnit is a unit testing framework designed for .NET. Its license has also been placed in the lib directory. The library is required for build-time only.

Chapter 2. Event Representations

This section outlines the different means to model and represent events.

Esper uses the term *event type* to describe the type information available for an event representation.

Your application may configure predefined event types at startup time or dynamically add event types at runtime via API or EPL syntax. See Section 13.4, “Configuration Items” for startup-time configuration and Section 12.3.7, “Runtime Configuration” for the runtime configuration API.

The EPL `create schema` syntax allows declaring an event type at runtime using EPL, see Section 4.16, “Declaring an Event Type: Create Schema”.

In Section 12.5, “Event and Event Type” we explain how an event type becomes visible in EPL statements and output events delivered by the engine.

2.1. Event Underlying Objects

An event is an immutable record of a past occurrence of an action or state change. Event properties capture the state information for an event.

In Esper, an event can be represented by any of the following underlying objects:

Table 2.1. Event Underlying Vanilla Objects

CLR Type	Description
<code>System.Object</code>	Any CLR type with read accessible properties.
<code>System.Collections.Generic.IDictionary<string, object></code>	Map (dictionary) events are key-values pairs and can also contain objects, further Map, and arrays thereof.
<code>System.Xml.XmlNode</code>	XML document object model (DOM).
Application classes	Plug-in event representation via the extension API.

Esper provides multiple choices for representing an event. There is no absolute need for you to create new types to represent an event.

Event representations have the following in common:

- All event representations support nested, indexed and mapped properties (aka. property expression), as explained in more detail below. There is no limitation to the nesting level.
- All event representations provide event type metadata. This includes type metadata for nested properties.
- All event representations allow transposing the event itself and parts of all of its property graph into new events. The term transposing refers to selecting the event itself or event properties that are themselves nestable property graphs, and then querying the event's properties or nested property graphs in further statements. The Apache Axiom event representation is an exception and does not currently allow transposing event properties but does allow transposing the event itself.
- The object and Map representations allow supertypes.

The API behavior for all event representations is the same, with minor exceptions noted in this chapter.

The benefits of multiple event representations are:

- For applications that already have events in one of the supported representations, there is no need to transform events into an object before processing.
- Event representations are exchangeable, reducing or eliminating the need to change statements when the event representation changes.
- Event representations are interoperable, allowing all event representations to interoperate in same or different statements.
- The choice makes it possible to consciously trade-off performance, ease-of-use, the ability to evolve and effort needed to import or externalize events and use existing event type metadata.

2.2. Event Properties

Event properties capture the state information for an event. Event properties be simple as well as indexed, mapped and nested event properties. The table below outlines the different types of properties and their syntax in an event expression. This syntax allows statements to query deep objects graphs, XML structures and Map events.

Table 2.2. Types of Event Properties

Type	Description	Syntax	Example
Simple	A property that has a single value that may be retrieved.	<code>name</code>	<code>sensorId</code>
Indexed	An indexed property stores an ordered collection of objects (all of the same type) that can be individually accessed by an integer-valued, non-negative index (or subscript).	<code>name[<i>index</i>]</code>	<code>sensor[0]</code>
Mapped	A mapped property stores a keyed collection of objects (all of the same type).	<code>name('key')</code>	<code>sensor('light')</code>
Nested	A nested property is a property that lives within another property of an event.	<code>name.nestedname</code>	<code>sensor.value</code>

Combinations are also possible. For example, a valid combination could be `person.address('home').street[0]`.

2.2.1. Escape Characters

If your application uses `System.Collection.Generic.IDictionary` or XML to represent events, then event property names may themselves contain the dot ('.') character. The backslash ('\') character can be used to escape dot characters in property names, allowing a property name to contain dot characters.

For example, the EPL as shown below expects a property by name `part1.part2` to exist on event type `MyEvent`:

```
select part1\.part2 from MyEvent
```

Sometimes your event properties may overlap with EPL language keywords. In this case you may use the back-wards apostrophe ` character to escape the property name.

The next example assumes a `Quote` event that has a property by name `order`, while `order` is also a reserved keyword:

```
select `order` from Quote
```

2.3. Dynamic Event Properties

Dynamic (unchecked) properties are event properties that need not be known at statement compilation time. Such properties are resolved during runtime: they provide duck typing functionality.

The idea behind dynamic properties is that for a given underlying event representation we don't always know all properties in advance. An underlying event may have additional properties that are not known at statement compilation time, that we want to query on. The concept is especially useful for events that represent rich, object-oriented domain models.

The syntax of dynamic properties consists of the property name and a question mark. Indexed, mapped and nested properties can also be dynamic properties:

Table 2.3. Types of Event Properties

Type	Syntax
Dynamic Simple	<code>name?</code>
Dynamic Indexed	<code>name[<i>index</i>]?</code>
Dynamic Mapped	<code>name('key')?</code>
Dynamic Nested	<code>name?.nestedPropertyName</code>

Dynamic properties always return the `System.Object` type. Also, dynamic properties return a `null` value if the dynamic property does not exist on events processed at runtime.

As an example, consider an `OrderEvent` event that provides an "item" property. The "item" property is of type `Object` and holds a reference to an instance of either a `Service` or `Product`.

Assume that both `Service` and `Product` classes provide a property named "price". Via a dynamic property we can specify a query that obtains the price property from either object (`Service` or `Product`):

```
select item.price? from OrderEvent
```

As a second example, assume that the `Service` class contains a "serviceName" property that the `Product` class does not possess. The following query returns the value of the "serviceName" property for `Service` objects. It returns a `null`-value for `Product` objects that do not have the "serviceName" property:

```
select item.serviceName? from OrderEvent
```

Consider the case where `OrderEvent` has multiple implementation classes, some of which have a "timestamp" property. The next query returns the timestamp property of those implementations of the `OrderEvent` interface that feature the property:

```
select timestamp? from OrderEvent
```

The query as above returns a single column named "timestamp?" of type `Object`.

When dynamic properties are nested, then all properties under the dynamic property are also considered dynamic properties. In the below example the query asks for the "direction" property of the object returned by the "detail" dynamic property:

```
select detail?.direction from OrderEvent
```

Above is equivalent to:

```
select detail?.direction? from OrderEvent
```

The functions that are often useful in conjunction with dynamic properties are:

- The `cast` function casts the value of a dynamic property (or the value of an expression) to a given type.
- The `exists` function checks whether a dynamic property exists. It returns `true` if the event has a property of that name, or `false` if the property does not exist on that event.
- The `instanceof` function checks whether the value of a dynamic property (or the value of an expression) is of any of the given types.
- The `typeof` function returns the string type name of a dynamic property.

Dynamic event properties work with all event representations outlined next: Objects, Map-based and XML DOM-based events.

2.4. Fragment and Fragment Type

Sometimes an event can have properties that are itself events. Esper uses the term *fragment* and *fragment type* for such event pieces. The best example is a pattern that matches two or more events and the output event contains the matching events as fragments. In other words, output events can be a composite event that consists of further events, the fragments.

Fragments have the same metadata available as their enclosing composite events. The metadata for enclosing composite events contains information about which properties are fragments, or have a property value that can be represented as a fragment and therefore as an event itself.

Fragments and type metadata can allow your application to navigate composite events without the need for using the reflection API and reducing the coupling to the underlying event representation. The API is further described in Section 12.5, "Event and Event Type".

2.5. Object Events

Object events are object instances that expose properties. While properties are the preferred method for expos-

ing information through Esper, properties can also be exposed if an accessor-style or accessor-method is defined via configuration.

Esper supports event types that extend a superclass or implement one or more interfaces. Also, Esper event pattern and EPL statements can refer to interface classes and abstract classes.

Classes that represent events should be made immutable. As events are recordings of a state change or action that occurred in the past, the relevant event properties should not be changeable. However this is not a hard requirement and the Esper engine accepts events that are mutable as well.

The `GetHashCode` and `Equals` methods do not need to be implemented. The implementation of these methods by an event type does not affect the behavior of the engine in any way.

Please see Chapter 13, *Configuration* on options for naming event types represented by object event types. Types that do not follow standard property conventions, such as those that expose public fields, require additional configuration. Via configuration it is also possible to control case sensitivity in property name resolution. The relevant section in the chapter on configuration is Section 13.4.1.3, “Legacy Event Classes”.

2.5.1. Object Event Properties

As outlined earlier, the different property types are supported by the standard CLR properties specification, and some of which are uniquely supported by Esper:

- *Simple* properties have a single value that may be retrieved. The underlying property type might be a CLR primitive (such as `int`), a simple object (such as a `System.String`), or a more complex object whose class is defined either by the CLR type structure, by the application, or by a type library included with the application.
- *Indexed* - An indexed property stores an ordered collection of objects (all of the same type) that can be individually accessed by an integer-valued, non-negative index (or subscript).
- *Mapped* - Esper considers any property that accepts a `String`-valued key a mapped property.
- *Nested* - A nested property is a property that lives within another object which itself is a property of an event.

Assume there is an `NewEmployeeEvent` event type as shown below. The mapped and indexed properties in this example return objects but could also return CLR primitive types (such as `Int32` or `String`). The `Address` object and `Employee` can themselves have properties that are nested within them, such as a street name in the `Address` object or a name of the employee in the `Employee` object.

```
public class NewEmployeeEvent {
    public String FirstName { get; }
    public Address GetAddress(String type);
    public Employee GetSubordinate(int index);
    public Employee[] AllSubordinates { get; }
}
```

Simple event properties require a read accessible type property returns the property value. This is standard convention for CLR-based applications and your specific language bindings will vary. In this example, the `FirstName` property returns the `firstName` event property of type `String`.

Indexed event properties require either one of the following getter-methods. A method that takes an integer-type key value and returns the property value, such as the `GetSubordinate` method, or a method that returns an array-type, or a class that implements `IEnumerable`. An example is the `GetSubordinates` getter method, which returns an array of `Employee` but could also return an `Iterable`. In an EPL or event pattern statement, indexed properties are accessed via the `property[index]` syntax.

Mapped event properties require a getter-method that takes a String-typed key value and returns the property value, such as the `GetAddress` method. In an EPL or event pattern statement, mapped properties are accessed via the `property('key')` syntax.

Nested event properties require a getter-method that returns the nesting object. The `Address` property and `GetSubordinate` method are mapped and indexed properties that return a nesting object. In an EPL or event pattern statement, nested properties are accessed via the `property.NestedProperty` syntax.

All event pattern and EPL statements allow the use of indexed, mapped and nested properties (or a combination of these) anywhere where one or more event property names are expected. The below example shows different combinations of indexed, mapped and nested properties in filters of event pattern expressions (each line is a separate EPL statement):

```
every NewEmployeeEvent(FirstName='myName')
every NewEmployeeEvent(Address('home').StreetName='Park Avenue')
every NewEmployeeEvent(Subordinate[0].Name='anotherName')
every NewEmployeeEvent(AllSubordinates[1].Name='thatName')
every NewEmployeeEvent(Subordinate[0].Address('home').StreetName='Water Street')
```

Similarly, the syntax can be used in EPL statements in all places where an event property name is expected, such as in select lists, where-clauses or join criteria.

```
select FirstName, Address('work'), Subordinate[0].Name, Subordinate[1].Name
from NewEmployeeEvent
where Address('work').StreetName = 'Park Ave'
```

2.5.2. Property Names

Property names are the result of a mixture of CLR standards and light method introspection of the type. Any properties that are defined via the CLR are automatically exposed. Additionally, any methods that are defined using the naming convention `GetProperty` is exposed as a property. In addition, Esper configuration provides a flag to turn off case-sensitive property names. A sample list of getter methods and property names is:

Table 2.4. Getter Methods and Property Names

Method	Property Name	Example
<code>GetPrice()</code>	<code>price</code>	<code>select Price from MyEvent</code>
<code>GetName()</code>	<code>NAME</code>	<code>select NAME from MyEvent</code>
<code>GetItemDesc()</code>	<code>itemDesc</code>	<code>select ItemDesc from MyEvent</code>
<code>GetQ()</code>	<code>q</code>	<code>select Q from MyEvent</code>
<code>GetQN()</code>	<code>QN</code>	<code>select QN from MyEvent</code>
<code>GetS()</code>	<code>s</code>	<code>select S from MyEvent</code>

It is important to note that Java and CLR property conventions are not compatible. In most cases, case conventions are opposite (e.g. Java uses camel case, whereas the CLR uses Pascal case).

2.5.3. Constants and Enumeration

Public static or const fields may also participate in expressions of all kinds, as this example shows:

```
select * from MyEvent where property=MyConstantClass.FIELD_VALUE
```

Event properties that are enumeration values can be compared by their enumeration value:

```
select * from MyEvent where enumProp=EnumClass.ENUM_VALUE_1
```

Alternatively, a static method may be employed on a class, such as the enumeration class 'EnumClass' as below:

```
select * from MyEvent where enumProp=EnumClass.valueOf('ENUM_VALUE_1')
```

If your application does not import, through configuration, the package that contains the enumeration type, then it must also specify the namespace of the type. Enumeration types that are inner classes must be qualified with + following conventions.

For example, the Color enumeration as an inner class to MyEvent in package org.myorg can be referenced as shown:

```
select * from MyEvent(enumProp=org.myorg.MyEvent+Color.GREEN).std:firstevent()
```

Instance methods may also be invoked on event instances by specifying a stream name, as shown below:

```
select myevent.computeSomething() as result from MyEvent as myevent
```

Chaining instance methods is supported as this example shows:

```
select myevent.GetComputerFor('books', 'movies').Calculate() as result
from MyEvent as myevent
```

2.5.4. Parameterized Types

When your getter methods or accessor fields return a parameterized type, for example `IEnumerable<MyEventData>` for an indexed property or `IDictionary<String, MyEventData>` for a mapped property, then property expressions may refer to the properties available through the class that is the type parameter.

An example event that has properties that are parameterized types is:

```
public class NewEmployeeEvent {
    public String Name { get; }
    public IEnumerable<EducationHistory> Education { get; }
    public IDictionary<String, Address> Addresses { get; }
}
```

A sample of valid property expressions for this event is shown next:

```
select Name, Education, Education[0].Date, Addresses('home').Street
from NewEmployeeEvent
```

2.5.5. Known Limitations

Esper employs byte code generation for fast access to event properties. When byte code generation is unsuccessful, the engine logs a warning and uses CLR reflection to obtain property values instead.

A known limitation is that when an interface has an attribute of a particular type and the actual type returns a subclass of that attribute, the engine logs a warning and uses reflection for that property.

2.6. Map Events

2.6.1. Overview

Events can also be represented by objects that implement the `System.Collection.IDictionary<String, Object>` interface. Henceforth, we will refer to this type loosely as a `DataMap`. Event properties of `DataMap` events are the values in the map accessible through the `indexer` exposed by the `IDictionary` interface.

The Map event type is a comprehensive type system that can eliminate the need to use CLR types as event types, thereby making it easier to change types at runtime or generate type information from another source.

A given Map event type can have one or more supertypes that must also be Map event types. All properties available on any of the Map supertypes are available on the type itself. In addition, anywhere within EPL that an event type name of a Map supertype is used, any of its Map subtypes and their subtypes match that expression.

Your application can add properties to an existing Map event type during runtime using the configuration operation `UpdateMapEventType`. Properties may not be updated or deleted - properties can only be added, and nested properties can be added as well. The runtime configuration also allows removing Map event types and adding them back with new type information.

After your application configures a Map event type by providing a type name, the type name can be used when defining further Map event types by specifying the type name as a property type or an array property type.

One-to-Many relationships in Map event types are represented via arrays. A property in a Map event type may be an array of primitive, an array of object or an array of Map.

The engine can process `DataMap` events via the `SendEvent(DataMap map, String eventName)` method on the `EPRuntime` interface. Entries in the Map represent event properties. Keys must be of type `System.String` for the engine to be able to look up event property names specified by pattern or EPL statements.

The engine does not validate Map event property names or values. Your application should ensure that objects passed in as event properties match the `create schema` property names and types, or the configured event type information when using runtime or static configuration.

2.6.2. Map Properties

Map event properties can be of any type. Map event properties that are application objects or that are of type `DataMap` (or arrays thereof) offer additional power:

- Properties that are application objects can be queried via the nested, indexed, mapped and dynamic property syntax as outlined earlier.
- Properties that are of type `Map` allow Maps to be nested arbitrarily deep and thus can be used to represent

complex domain information. The nested, indexed, mapped and dynamic property syntax can be used to query Maps within Maps and arrays of Maps within Maps.

In order to use `Map` events, the event type name and property names and types must be made known to the engine via `Configuration`. Please see the examples in Section 13.4.2, “Events represented by `System.Collection.Generic.IDictionary`”.

The code snippet below creates and processes a `Map` event. It defines a `CarLocationUpdateEvent` event type first:

```
var mapEvent = new Dictionary<string,object>();
mapEvent["carId"] = carId;
mapEvent["direction"] = direction;
epRuntime.SendEvent(mapEvent, "CarLocUpdateEvent");
```

The `CarLocUpdateEvent` can now be used in a statement:

```
select carId from CarLocUpdateEvent.win:time(1 min) where direction = 1
```

The engine can also query objects as values in a `Map` event via the nested property syntax. Thus `Map` events can be used to aggregate multiple data structures into a single event and query the composite information in a convenient way. The example below demonstrates a `Map` event with a transaction and an account object.

```
var mapEvent = new Dictionary<string,object>();
mapEvent["txn"] = txn;
mapEvent["account"] = account;
epRuntime.SendEvent(mapEvent, "TxnEvent");
```

An example statement could look as follows.

```
select account.id, account.rate * txn.amount
from TxnEvent.win:time(60 sec)
group by account.id
```

2.6.3. Map Supertypes

Your `Map` event type may declare one or more supertypes when configuring the type at engine initialization time or at runtime through the administrative interface.

Supertypes of a `Map` event type must also be `Map` event types. All property names and types of a supertype are also available on a subtype and override such same-name properties of the subtype. In addition, anywhere within EPL that an event type name of a `Map` supertype is used, any of its `Map` subtypes also matches that expression (similar to the concept of interface).

This example assumes that the `BaseUpdate` event type has been declared and acts as a supertype to the `AccountUpdate` event type (both `Map` event types):

```
epService.EPAdministrator.GetConfiguration().
    AddEventType("AccountUpdate", accountUpdateDef,
        new String[] { "BaseUpdate" });
```

Your application EPL statements may select `BaseUpdate` events and receive both `BaseUpdate` and `AccountUpdate` events, as well as any other subtypes of `BaseUpdate` and their subtypes.

```
// Receive BaseUpdate and any subtypes including subtypes of subtypes
select * from BaseUpdate
```

Your application Map event type may have multiple supertypes. The multiple inheritance hierarchy between Maps can be arbitrarily deep, however cyclic dependencies are not allowed. If using runtime configuration, supertypes must exist before a subtype to a supertype can be added.

See Section 13.4.2, “Events represented by System.Collection.Generic.IDictionary” for more information on configuring Map event types.

2.6.4. Advanced Map Property Types

Nested Properties

Strongly-typed nested Map-within-Map events can be used to build rich, type-safe event types on the fly. Use the `AddEventType` method on `Configuration` or `ConfigurationOperations` for initialization-time and runtime-time type definition.

Noteworthy points are:

- Objects can appear as properties in Map-within-Map.
- One may represent Map-within-Map and Map-Array within Map using the name of a previously registered Map event type.
- There is no limit to the number of nesting levels.
- Dynamic properties can be used to query Map-within-Map keys that may not be known in advance.
- The engine returns a `null` value for properties for which the access path into the nested structure cannot be followed where map entries do not exist.

For demonstration, in this example our top-level event type is an `AccountUpdate` event, which has an `UpdatedField` structure as a property. Inside the `UpdatedField` structure the example defines various fields, as well as a property by name 'history' that holds a type `UpdateHistory` to represent the update history for the account. The code snippet to define the event type is thus:

```
IDictionary<String, Object> updatedFieldDef = new Dictionary<String, Object>();
updatedFieldDef["name"] = typeof(string);
updatedFieldDef["addressLine1"] = typeof(string);
updatedFieldDef["history"] = typeof(UpdateHistory);

IDictionary<String, Object> accountUpdateDef = new Dictionary<String, Object>();
accountUpdateDef["accountId"] = typeof(long);
accountUpdateDef["fields"] = updatedFieldDef;

epService.EPAdministrator.GetConfiguration().
    AddEventType("AccountUpdate", accountUpdateDef);
```

The next code snippet populates a sample event and sends the event into the engine:

```
IDictionary<String, Object> updatedField = new Dictionary<String, Object>();
updatedField["name"] = "Joe Doe";
updatedField["addressLine1"] = "40 Popular Street";
updatedField["history"] = new UpdateHistory();

IDictionary<String, Object> accountUpdate = new Dictionary<String, Object>();
accountUpdate["accountId"] = 10009901;
accountUpdate["fields"] = updatedField;

epService.EPRuntime.SendEvent(accountUpdate, "AccountUpdate");
```

Last, a sample query to interrogate `AccountUpdate` events is as follows:

```
select accountId, fields.name, fields.addressLine1, fields.history.lastUpdate
```

```
from AccountUpdate
```

Note that type information for nested maps is only available to the immediately selecting stream. For example, the second select-query does not work:

```
insert into MyStream select fields from NestedMapEvent
// this does not work ... instead select the individual fields in the insert-into statement
select fields.name from MyStream
```

Map Event Type Properties

Your application may declare a Map event type for reuse within other Map event types or for one-to-many properties represented by an array of Maps.

This example declares a Map event type by name `AmountCurrency` with amount and currency properties:

```
IDictionary<String, Object> amountAndCurr = new Dictionary<String, Object>();
amountAndCurr["amount"] = typeof(double);
amountAndCurr["currency"] = typeof(string);

epService.EPAdministrator.GetConfiguration().
    AddEventType("AmountCurrency", amountAndCurr);
```

The `AmountCurrency` type is now available for use as a property type itself. Below code snippet declares `OrderItem` to hold an item number and `AmountCurrency`:

```
IDictionary<String, Object> orderItem = new Dictionary<String, Object>();
orderItem["itemNum"] = typeof(int);
orderItem["price"] = "AmountCurrency";    // The property type is the name itself

epService.EPAdministrator.GetConfiguration().
    AddEventType("OrderItem", orderItem);
```

One-to-Many Relationships

To model repeated properties within a Map, you may use arrays as properties in a Map. You may use an array of primitive types or an array of objects or an array of a previously declared Map event type.

When using a previously declared Map event type as an array property, the literal `[]` must be appended after the event type name.

This following example defines a Map event type by name `Sale` to hold array properties of the various types. It assumes a `SalesPerson` type exists and a Map event type by name `OrderItem` was declared:

```
Map<String, Object> sale = new HashMap<String, Object>();
sale["userids"] = typeof(int[]);
sale["salesPersons"] = typeof(SalesPerson[]);
sale["items"] = "OrderItem[]";    // The property type is the name itself appended by []

epService.EPAdministrator.GetConfiguration().
    AddEventType("SaleEvent", sale);
```

The three properties that the above example declares are:

- An integer array of user ids.
- An array of `SalesPerson` objects.

- An array of Maps for order items.

The next EPL statement is a sample query asking for property values held by arrays:

```
select userids[0], salesPersons[1].name,
       items[1], items[1].price.amount from SaleEvent
```

2.7. `System.Xml.XmlNode` XML Events

Events can be represented as `System.Xml.XmlNode` instances and send into the engine via the `SendEvent` method on `EPRuntime` or via `EventSender`. Please note that configuration is required so the event type name and root element name is known. See Chapter 13, *Configuration*.

If a XML schema document (XSD file) can be made available as part of the configuration, then Esper can read the schema and appropriately present event type metadata and validate statements that use the event type and its properties. See Section 2.7.1, “Schema-Provided XML Events”.

When no XML schema document is provided, XML events can still be queried, however the return type and return values of property expressions are string-only and no event type metadata is available other than for explicitly configured properties. See Section 2.7.2, “No-Schema-Provided XML Events”.

In all cases Esper allows you to configure explicit XPath expressions as event properties. You can specify arbitrary XPath functions or expressions and provide a property name and type by which result values will be available for use in EPL statements. See Section 2.7.3, “Explicitly-Configured Properties”.

Nested, mapped and indexed event properties are also supported in expressions against `System.Xml.XmlNode` events. Thus XML trees can conveniently be interrogated via the property expression syntax.

Only one event type per root element name may be configured. The engine recognizes each event by its root element name or you may use `EventSender` to send events.

This section uses the following XML document as an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<Sensor xmlns="SensorSchema">
  <ID>urn:epc:1:4.16.36</ID>
  <Observation Command="READ_PALLET_TAGS_ONLY">
    <ID>00000001</ID>
    <Tag>
      <ID>urn:epc:1:2.24.400</ID>
    </Tag>
    <Tag>
      <ID>urn:epc:1:2.24.401</ID>
    </Tag>
  </Observation>
</Sensor>
```

The schema for the example is:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="Sensor">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="ID" type="xs:string"/>
        <xs:element ref="Observation" />
      </xs:sequence>
    </xs:complexType>
```

```

</xs:element>

<xs:element name="Observation">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="ID" type="xs:string"/>
      <xs:element ref="Tag" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="Command" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>

<xs:element name="Tag">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="ID" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>

```

2.7.1. Schema-Provided XML Events

If you have a XSD schema document available for your XML events, Esper can interrogate the schema. The benefits are:

- New EPL statements that refer to event properties are validated against the types provided in the schema.
- Event type metadata becomes available for retrieval as part of the `EventType` interface.

Getting Started

The engine reads a XSD schema file from an URL you provide. Make sure files imported by the XSD schema file can also be resolved.

The configuration accepts a schema URL. This is a sample code snippet to determine a schema URL from the resource manager:

```
URL schemaURL = ResourceManager.ResolveResourceURL("sensor.xsd");
```

Here is a sample use of the runtime configuration API, please see Chapter 13, *Configuration* for further examples.

```

epService = EPServiceProviderManager.GetDefaultProvider();
ConfigurationEventTypeXMLDOM sensorcfg = new ConfigurationEventTypeXMLDOM();
sensorcfg.RootElementName = "Sensor";
sensorcfg.SchemaResource = schemaURL.ToString();
epService.EPAdministrator.GetConfiguration()
    .AddEventType("SensorEvent", sensorcfg);

```

You must provide a root element name. This name is used to look up the event type for the `SendEvent(System.Xml.XmlNode node)` method. An `EventSender` is a useful alternative method for sending events if the type lookup based on the root or document element name is not desired.

After adding the event type, you may create statements and send events. Next is a sample statement:

```

select ID, Observation.Command, Observation.ID,
       Observation.Tag[0].ID, Observation.Tag[1].ID
from SensorEvent

```

As you can see from the example above, property expressions can query property values held in the XML document's elements and attributes.

There are multiple ways to obtain a XML DOM document instance from a XML string. The next code snippet shows how to obtain a XML DOM `System.Xml.XmlDocument` instance:

```
XmlDocument doc = new XmlDocument();
doc.LoadXml(xml);
```

Send the `System.Xml.XmlNode` Or `Document` object into the engine for processing:

```
epService.EPRuntime.SendEvent(doc);
```

Property Expressions and Namespaces

By default, property expressions such as `Observation.Tag[0].ID` are evaluated by a fast DOM-walker implementation provided by Esper. This DOM-walker implementation is not namespace-aware.

Should you require namespace-aware traversal of the DOM document, you must set the `xpath-property-expr` configuration option to true (default is false). This flag causes Esper to generate namespace-aware XPath expressions from each property expression instead of the DOM-walker, as described next. Setting the `xpath-property-expr` option to true requires that you also configure namespace prefixes as described below.

When matching up the property names with the XSD schema information, the engine determines whether the attribute or element provides values. The algorithm checks attribute names first followed by element names. It takes the first match to the specified property name.

Property Expression to XPath Rewrite

By setting the `xpath-property-expr` option the engine rewrites each property expression as an XPath expression, effectively handing the evaluation over to the underlying XPath implementation available from classpath. Most JVM have a built-in XPath implementation and there are also optimized, fast implementations such as Jaxen that can be used as well.

Set the `xpath-property-expr` option if you need namespace-aware document traversal, such as when your schema mixes several namespaces and element names are overlapping.

The below table samples several property expressions and the XPath expression generated for each, without namespace prefixes to keep the example simple:

Table 2.5. Property Expression to XPath Expression

Property Expression	Equivalent XPath
<code>Observation.ID</code>	<code>/Sensor/Observation/ID</code>
<code>Observation.Command</code>	<code>/Sensor/Observation/@Command</code>
<code>Observation.Tag[0].ID</code>	<code>/Sensor/Observation/Tag[position() = 1]/ID</code>

For mapped properties that are specified via the syntax `name('key')`, the algorithm looks for an attribute by name `id` and generates a XPath expression as `mapped[@id='key']`.

Finally, here is an example that includes all different types of properties and their XPath expression equivalent

in one property expression:

```
select nested.mapped('key').indexed[1].attribute from MyEvent
```

The equivalent XPath expression follows, this time including `n0` as a sample namespace prefix:

```
/n0:rootelement/n0:nested/n0:mapped[@id='key']/n0:indexed[position() = 2]/@attribute
```

Array Properties

All elements that are unbound or have `max` occurs greater than 1 in the XSD schema are represented as indexed properties and require an index for resolution.

For example, the following is not a valid property expression in the sample `Sensor` document: `Observation.Tag.ID`. As no index is provided for `Tag`, the property expression is not valid.

Repeated elements within a parent element in which the repeated element is a simple type also are represented as an array.

Consider the next XML document:

```
<item>
  <book sku="8800090">
    <author>Isaac Asimov</author>
    <author>Robert A Heinlein</author>
  </book>
</item>
```

Here, the result of the expression `book.author` is an array of type `String` and the result of `book.author[0]` is a `String` value.

Dynamic Properties

Dynamic properties are not validated against the XSD schema information and their result value is always `System.Xml.XmlNode`. You may use a user-defined function to process dynamic properties returning `Node`. As an alternative consider using an explicit property.

An example dynamic property is `Origin?.ID` which will look for an element by name `Origin` that contains an element or attribute node by name `LocationCode`:

```
select Origin?.LocationCode from SensorEvent
```

Transposing Properties

When providing a XSD document, the default configuration allows to transpose property values that are themselves complex elements, as defined in the XSD schema, into a new stream. This behavior can be controlled via the flag `auto-fragment`.

For example, consider the next query:

```
insert into ObservationStream
select ID, Observation from SensorEvent
```

The `Observation` as a property of the `SensorEvent` gets itself inserted into a new stream by name `ObservationStream`. The `ObservationStream` thus consists of a string-typed `ID` property and a complex-typed property

named `Observation`, as described in the schema.

A further statement can use this stream to query:

```
select Observation.Command, Observation.Tag[0].ID from ObservationStream
```

Before continuing the discussion, here is an alternative syntax using the wildcard-select, that is also useful:

```
insert into TagListStream
select ID as sensorId, Observation.* from SensorEvent
```

The new `TagListStream` has a string-typed `ID` and `Command` property as well as an array of `Tag` properties that are complex types themselves as defined in the schema.

Next is a sample statement to query the new stream:

```
select sensorId, Command, Tag[0].ID from TagListStream
```

Please note the following limitations:

- The XPath standard prescribes that XPath expressions against `System.Xml.Node` are evaluated against the owner document of the `Node`. Therefore XPath is not relative to the current node but absolute against each node's owner document. Since Esper does not create new document instances for transposed nodes, transposing properties is not possible when the `xpath-property-expr` flag is set.
- Complex elements that have both simple element values and complex child elements are not transposed. This is to ensure their property value is not hidden. Use an explicit XPath expression to transpose such properties.

Esper automatically registers a new event type for transposed properties. It generates the type name of the new XML event type from the XML event type name and the property names used in the expression. The synopsis is `type_name.property_name[.property_name...]`. The type name can be looked up, for example for use with `EventSender` or can be created in advance.

Event Sender

An `IEventSender` sends events into the engine for a given type, saving a type lookup based on element name.

This brief example sends an event via `IEventSender`:

```
EventSender sender = epRuntime.GetEventSender("SensorEvent");
sender.sendEvent(node);
```

The XML DOM event sender checks the root element name before processing the event. Use the `event-sender-validates-root` setting to disable validation. This forces the engine to process XML documents according to any predefined type without validation of the root element name.

2.7.2. No-Schema-Provided XML Events

Without a schema document a XML event may still be queried. However there are important differences in the metadata available without a schema document and therefore the property expression results. These differences are outlined below.

All property expressions against a XML type without schema are assumed valid. There is no validation of the property expression other than syntax validation. At runtime, property expressions return string-type values or `null` if the expression did not yield a matching element or attribute result.

When asked for property names or property metadata, a no-schema type returns empty array.

In all other aspects the type behaves the same as the schema-provided type described earlier.

2.7.3. Explicitly-Configured Properties

Regardless of whether or not you provide a XSD schema for the XML event type, you can always fall back to configuring explicit properties that are backed by XPath expressions.

For further documentation on XPath, please consult the XPath standard or other online material.

Simple Explicit Property

Shown below is an example configuration that adds an explicit property backed by a XPath expression and that defines namespace prefixes:

```
epService = EPServiceProviderManager.GetDefaultProvider();
ConfigurationEventTypeXMLDOM sensorcfg = new ConfigurationEventTypeXMLDOM();
sensorcfg.AddXPathProperty("countTags", "count(/ss:Sensor/ss:Observation/ss:Tag)",
    XPathResultType.Number);
sensorcfg.AddNamespacePrefix("ss", "SensorSchema");
sensorcfg.RootElementName = "Sensor";
epService.EPAdministrator.GetConfiguration()
    .AddEventType("SensorEvent", sensorcfg);
```

The countTags property is now available for querying:

```
select countTags from SensorEvent
```

The XPath expression count(...) is a XPath built-in function that counts the number of nodes, for the example document the result is 2.

Explicit Property Casting and Parsing

Esper can parse or cast the result of your XPath expression to the desired type. Your property configuration provides the type to cast to, like this:

```
sensorcfg.AddXPathProperty("countTags", "count(/ss:Sensor/ss:Observation/ss:Tag)",
    XPathResultType.Number, "int");
```

The type supplied to the property configuration must be one of the built-in types. Arrays of built-in type are also possible, requiring the XPathResultType.NodeSet type returned by your XPath expression, as follows:

```
sensorcfg.AddXPathProperty("idarray", "//ss:Tag/ss:ID",
    XPathResultType.NodeSet, "String[]");
```

The XPath expression //ss:Tag/ss:ID returns all ID nodes under a Tag node, regardless of where in the node tree the element is located. For the example document the result is 2 array elements urn:epc:1:2.24.400 and urn:epc:1:2.24.40.

Node and Nodeset Explicit Property

An explicit property may return XPathResultType.Any or XPathResultType.NodeSet and can provide the event type name of a pre-configured event type for the property. The method name to add such properties is AddXPathPropertyFragment.

This code snippet adds two explicit properties and assigns an event type name for each property:

```
sensorcfg.AddXPathPropertyFragment("tagOne", "//ss:Tag[position() = 1]",  
    XPathResultType.Any, "TagEvent");  
sensorcfg.AddXPathPropertyFragment("tagArray", "//ss:Tag",  
    XPathResultType.NodeSet, "TagEvent");
```

The configuration above references the `TagEvent` event type. This type must also be configured. Prefix the root element name with `"/"` to cause the lookup to search the nested schema elements for the definition of the type:

```
ConfigurationEventTypeXMLDOM tagcfg = new ConfigurationEventTypeXMLDOM();  
tagcfg.RootElementName = "/Tag";  
tagcfg.SchemaResource = schemaURL;  
epAdministrator.GetConfiguration()  
    .AddEventType("TagEvent", tagcfg);
```

The `tagOne` and `tagArray` properties are now ready for selection and transposing to further streams:

```
insert into TagOneStream select tagOne.* from SensorEvent
```

Select from the new stream:

```
select ID from TagOneStream
```

An example with indexed properties is shown next:

```
insert into TagArrayStream select tagArray as mytags from SensorEvent
```

Select from the new stream:

```
select mytags[0].ID from TagArrayStream
```

2.8. Additional Event Representations

Part of the extension and plug-in features of Esper is an event representation API. This set of classes allow an application to create new event types and event instances based on information available elsewhere, statically or dynamically at runtime when EPL statements are created. Please see Section 15.8, “Event Type And Event Object” for details.

Creating a plug-in event representation can be useful when your application has existing types that carry event metadata and event property values and your application does not want to (or cannot) extract or transform such event metadata and event data into one of the built-in event representations (CLR objects, Map or XML DOM).

Further use of a plug-in event representation is to provide a faster or short-cut access path to event data. For example, access to event data stored in a XML format through the Streaming API for XML (StAX) is known to be very efficient. A plug-in event representation can also provide network lookup and dynamic resolution of event type and dynamic sourcing of event instances.

The chapter on Section 15.8, “Event Type And Event Object” explains how to create your own custom event representation.

2.9. Updating, Merging and Versioning Events

To summarize, an event is an immutable record of a past occurrence of an action or state change, and event properties contain useful information about an event.

The length of time an event is of interest to the event processing engine (retention time) depends on your EPL statements, and especially the data window, pattern and output rate limiting clauses of your statements.

During the retention time of an event more information about the event may become available, such as additional properties or changes to existing properties. Esper provides three concepts for handling updates to events.

The first means to handle updating events is the `update istream` clause as further described in Section 4.20, “Updating an Insert Stream: the Update IStream Clause”. It is useful when you need to update events as they enter a stream, before events are evaluated by any particular consuming statement to that stream.

The second means to update events is the `on-merge` and `on-update` clauses, for use with named windows only, as further described in Section 4.15.9, “Triggered Upsert using the On-Merge Clause” and Section 4.15.7, “Updating Named Windows: the On Update clause”. `On-merge` is similar to the SQL `merge` clause and provides what is known as an "Upsert" operation: Update existing events or if no existing event(s) are found then insert a new event, all in one atomic operation provided by a single EPL statement. `On-update` can be used to update individual properties of events held in a named window.

The third means to handle updating events is the revision event types, for use with named windows only, as further described in Section 4.15.11, “Versioning and Revision Event Type Use with Named Windows”. With revision event types one can declare, via configuration only, multiple different event types and then have the engine present a merged event type that contains a superset of properties of all merged types, and have the engine merge events as they arrive without additional EPL statements.

Note that patterns do not reflect changes to past events. For the temporal nature of patterns, any changes to events that were observed in the past do not reflect upon current pattern state.

2.10. Coarse-Grained Events

Your application events may consist of fairly comprehensive, coarse-grained structures or documents. For example in business-to-business integration scenarios, XML documents or other event objects can be rich deeply-nested graphs of event properties.

To extract information from a coarse-grained event or to perform bulk operations on the rows of the property graph in an event, Esper provides a convenient syntax: When specifying a filter expression in a pattern or in a `select` clause, it may contain a contained-event selection syntax, as further described in Section 4.19, “Contained-Event Selection”.

2.11. Event Objects Populated by `Insert Into`

The `insert into` clause can populate plain-old object events and `IDictionary` events directly from the results of `select` clause expressions. Simply use the event type name as the stream name in the `insert into` clause as described in Section 4.10, “Merging Streams and Continuous Insertion: the Insert Into Clause”.

The column names specified in the `select` and `insert into` clause must match available writable properties in the event object to be populated (the target event type). The expression result types of any expressions in the `select` clause must also be compatible with the property types of the target event type.

Consider the following example statement:

```
insert into com.mycompany.NewEmployeeEvent
select fname as firstName, lname as lastName from HRSystemEvent
```

The above example specifies the fully-qualified class name of `NewEmployeeEvent`. The engine instantiates `NewEmployeeEvent` for each result row and populates the `firstName` and `lastName` properties of each instance from the result of `select` clause expressions. The `HRSystemEvent` in the example is assumed to have `lname` and `fname` properties.

Note how the example uses the `as`-keyword to assign column names that match the property names of the `NewEmployeeEvent` target event. If the property names of the source and target events are the same, the `as`-keyword is not required.

The next example is an alternate form and specifies property names within the `insert into` clause instead. The example also assumes that `NewEmployeeEvent` has been defined or imported via configuration since it does not specify the event class package name:

```
insert into NewEmployeeEvent(firstName, lastName)
select fname, lname from HRSystemEvent
```

Finally, this example populates `HRSystemEvent` events. The example populates the value of a `type` property where the event has the value 'NEW' and populates a new event object with the value 'HIRED', copying the `fname` and `lname` property values to the new event object:

```
insert into HRSystemEvent
select fname, lname, 'HIRED' as type from HRSystemEvent(type='NEW')
```

The matching of the `select` or `insert into`-clause column names to target event type's property names is case-sensitive. It is allowed to only populate a subset of all available columns in the target event type. Wildcard (*) is also allowed and copies all fields of the events or multiple events in a join.

For CLR object events, your event class must provide writable properties. The event type should also provide a default constructor taking no parameters. If your event class does not have a default constructor, your application may configure a factory method via `ConfigurationEventTypeLegacy`.

The engine follows CLR standards in terms of widening, performing widening automatically in cases where widening type conversion is allowed without loss of precision, for both boxed and primitive types.

When inserting array-typed properties into a CLR or Map-type underlying event the event definition should declare the target property as an array.

Please note the following limitations:

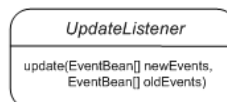
- Event types that utilize XML `System.Xml.XmlNode` underlying event objects cannot be target of an `insert into` clause.

Chapter 3. Processing Model

3.1. Introduction

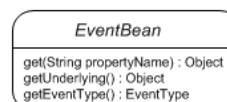
The Esper processing model is continuous: Update listeners and/or subscribers to statements receive updated data as soon as the engine processes events for that statement, according to the statement's choice of event streams, views, filters and output rates.

As outlined in Chapter 12, *API Reference* the delegate for event handlers is `com.espertech.esper.client.UpdateEventHandler`. Implementations are invoked by the engine when results become available:



A second, strongly-typed and native, highly-performant method of result delivery is provided: A subscriber object is a direct binding of query results to an object. The object receives statement results via method invocation. The subscriber class need not implement an interface or extend a superclass. Please see Section 12.3.3, “Setting a Subscriber Object”.

The engine provides statement results to update listeners by placing results in `com.espertech.esper.client.EventBean` instances. A typical listener implementation queries the `EventBean` instances via getter methods to obtain the statement-generated results.



The `Get` method on the `EventBean` interface can be used to retrieve result columns by name. The property name supplied to the `Get` method can also be used to query nested, indexed or array properties of object graphs as discussed in more detail in Chapter 2, *Event Representations* and Section 12.5, “Event and Event Type”

The `Underlying` properties on the `EventBean` interface allows update listeners to obtain the underlying event object. For wildcard selects, the underlying event is the event object that was sent into the engine via the `SendEvent` method. For joins and select clauses with expressions, the underlying object implements `System.Collections.Generic.IDictionary`.

3.2. Insert Stream

In this section we look at the output of a very simple EPL statement. The statement selects an event stream without using a data window and without applying any filtering, as follows:

```
select * from Withdrawal
```

This statement selects all `Withdrawal` events. Every time the engine processes an event of type `Withdrawal` or any sub-type of `Withdrawal`, it invokes all update listeners, handing the new event to each of the statement's listeners.

The term *insert stream* denotes the new events arriving, and entering a data window or aggregation. The insert stream in this example is the stream of arriving Withdrawal events, and is posted to listeners as new events.

The diagram below shows a series of Withdrawal events 1 to 6 arriving over time. The number in parenthesis is the withdrawal amount, an event property that is used in the examples that discuss filtering.

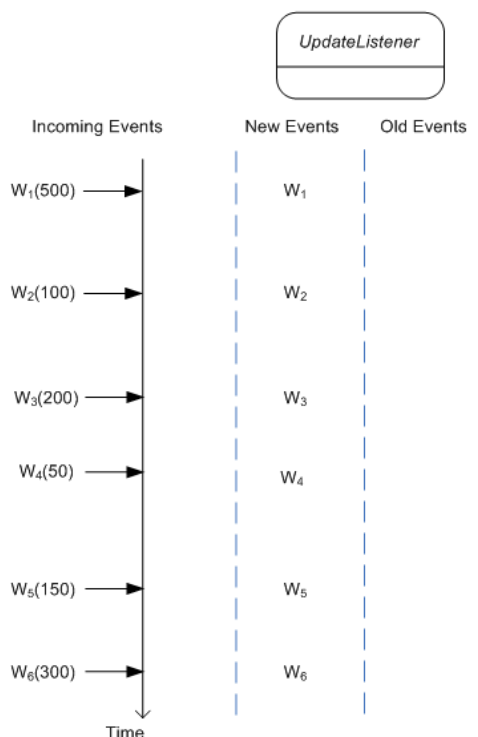


Figure 3.1. Output example for a simple statement

The example statement above results in only new events and no old events posted by the engine to the statement's listeners.

3.3. Insert and Remove Stream

A length window instructs the engine to only keep the last N events for a stream. The next statement applies a length window onto the Withdrawal event stream. The statement serves to illustrate the concept of data window and events entering and leaving a data window:

```
select * from Withdrawal.win:length(5)
```

The size of this statement's length window is five events. The engine enters all arriving Withdrawal events into the length window. When the length window is full, the oldest Withdrawal event is pushed out the window. The engine indicates to listeners all events entering the window as new events, and all events leaving the window as old events.

While the term *insert stream* denotes new events arriving, the term *remove stream* denotes events leaving a data window, or changing aggregation values. In this example, the remove stream is the stream of Withdrawal events that leave the length window, and such events are posted to listeners as old events.

The next diagram illustrates how the length window contents change as events arrive and shows the events pos-

ted to an update listener.

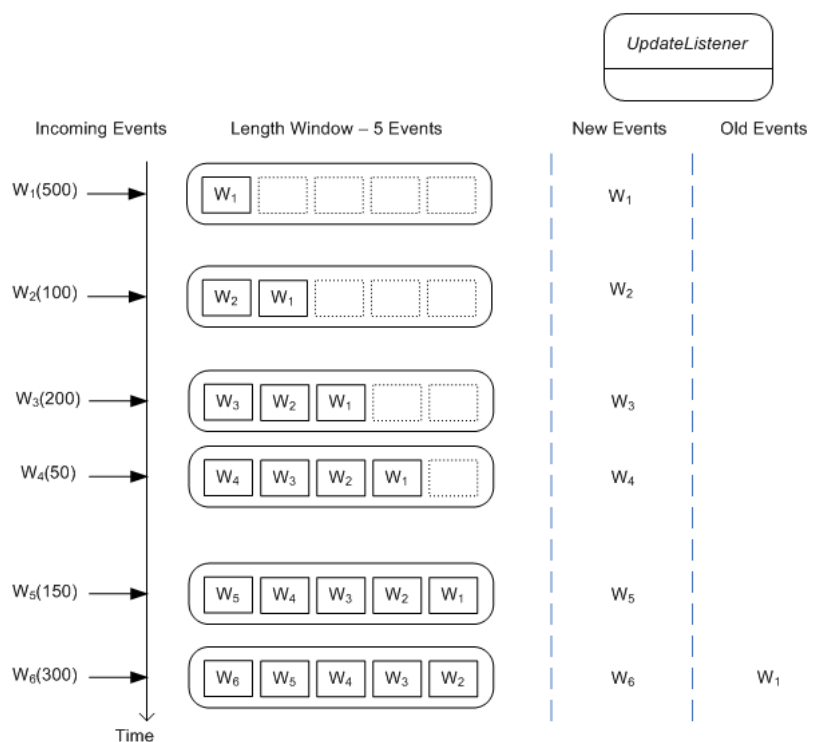


Figure 3.2. Output example for a length window

As before, all arriving events are posted as new events to listeners. In addition, when event W_1 leaves the length window on arrival of event W_6 , it is posted as an old event to listeners.

Similar to a length window, a time window also keeps the most recent events up to a given time period. A time window of 5 seconds, for example, keeps the last 5 seconds of events. As seconds pass, the time window actively pushes the oldest events out of the window resulting in one or more old events posted to update listeners.

Note: By default the engine only delivers the insert stream to listeners and observers. EPL supports optional `istream`, `irstream` and `rstream` keywords on select-clauses and on insert-into clauses to control which stream to deliver, see Section 4.3.7, “Selecting insert and remove stream events”. There is also a related, engine-wide configuration setting described in Section 13.4.14, “Engine Settings related to Stream Selection”.

3.4. Filters and Where-clauses

Filters to event streams allow filtering events out of a given stream before events enter a data window. The statement below shows a filter that selects Withdrawal events with an amount value of 200 or more.

```
select * from Withdrawal(amount>=200).win:length(5)
```

With the filter, any Withdrawal events that have an amount of less than 200 do not enter the length window and are therefore not passed to update listeners. Filters are discussed in more detail in Section 4.4.1, “Filter-based Event Streams” and Section 5.4, “Filter Expressions In Patterns”.

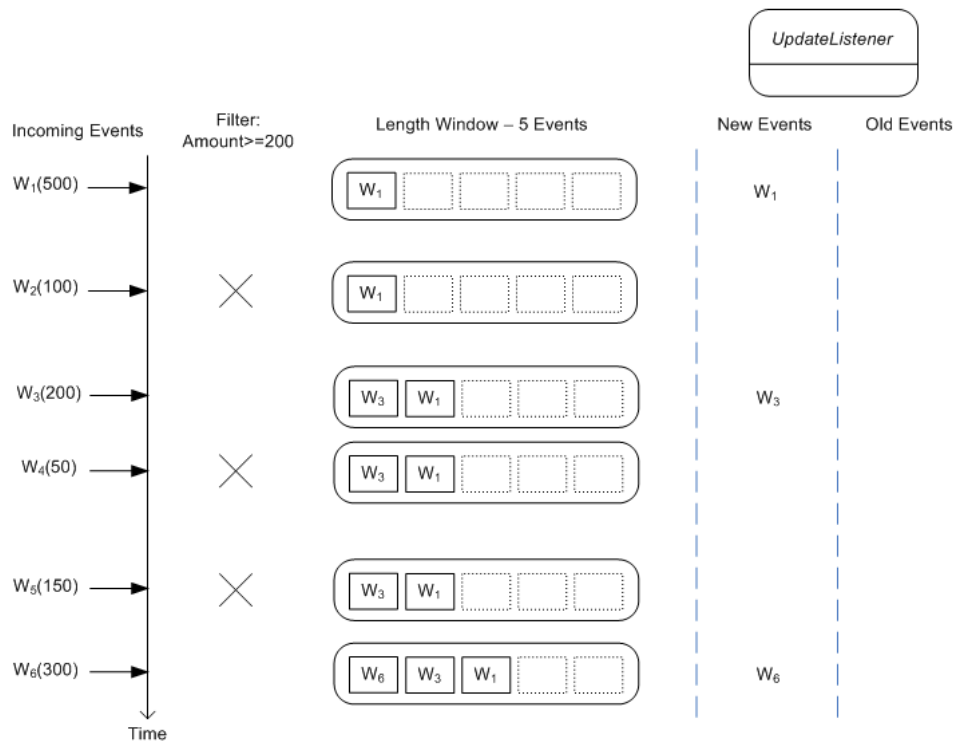


Figure 3.3. Output example for a statement with an event stream filter

The where-clause and having-clause in statements eliminate potential result rows at a later stage in processing, after events have been processed into a statement's data window or other views.

The next statement applies a where-clause to Withdrawal events. Where-clauses are discussed in more detail in Section 4.5, "Specifying Search Conditions: the Where Clause".

```
select * from Withdrawal.win:length(5) where amount >= 200
```

The where-clause applies to both new events and old events. As the diagram below shows, arriving events enter the window however only events that pass the where-clause are handed to update listeners. Also, as events leave the data window, only those events that pass the conditions in the where-clause are posted to listeners as old events.

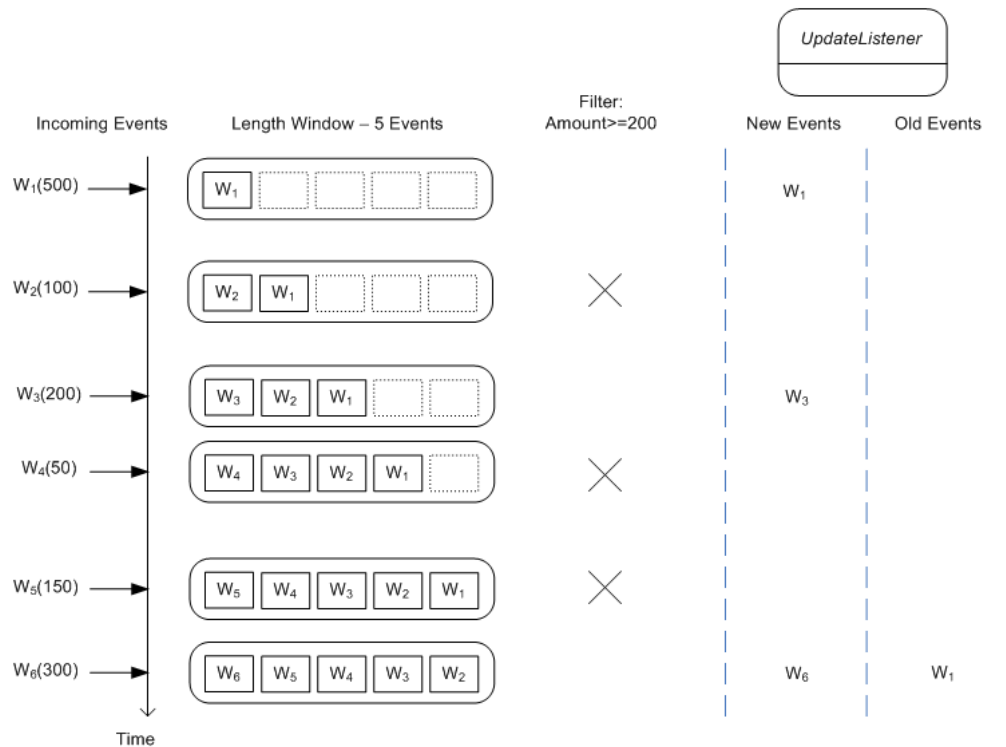


Figure 3.4. Output example for a statement with where-clause

The where-clause can contain complex conditions while event stream filters are more restrictive in the type of filters that can be specified. The next statement's where-clause applies the `ceil` function of the `System.Math` library type in the where clause. The insert-into clause makes the results of the first statement available to the second statement:

```
insert into WithdrawalFiltered select * from Withdrawal where Math.Ceiling(amount) >= 200
```

```
select * from WithdrawalFiltered
```

3.5. Time Windows

In this section we explain the output model of statements employing a time window view and a time batch view.

3.5.1. Time Window

A time window is a moving window extending to the specified time interval into the past based on the system time. Time windows enable us to limit the number of events considered by a query, as do length windows.

As a practical example, consider the need to determine all accounts where the average withdrawal amount per account for the last 4 seconds of withdrawals is greater than 1000. The statement to solve this problem is shown below.

```
select account, avg(amount)
from Withdrawal.win:time(4 sec)
group by account
having amount > 1000
```

The next diagram serves to illustrate the functioning of a time window. For the diagram, we assume a query that simply selects the event itself and does not group or filter events.

```
select * from Withdrawal.win:time(4 sec)
```

The diagram starts at a given time t and displays the contents of the time window at $t + 4$ and $t + 5$ seconds and so on.

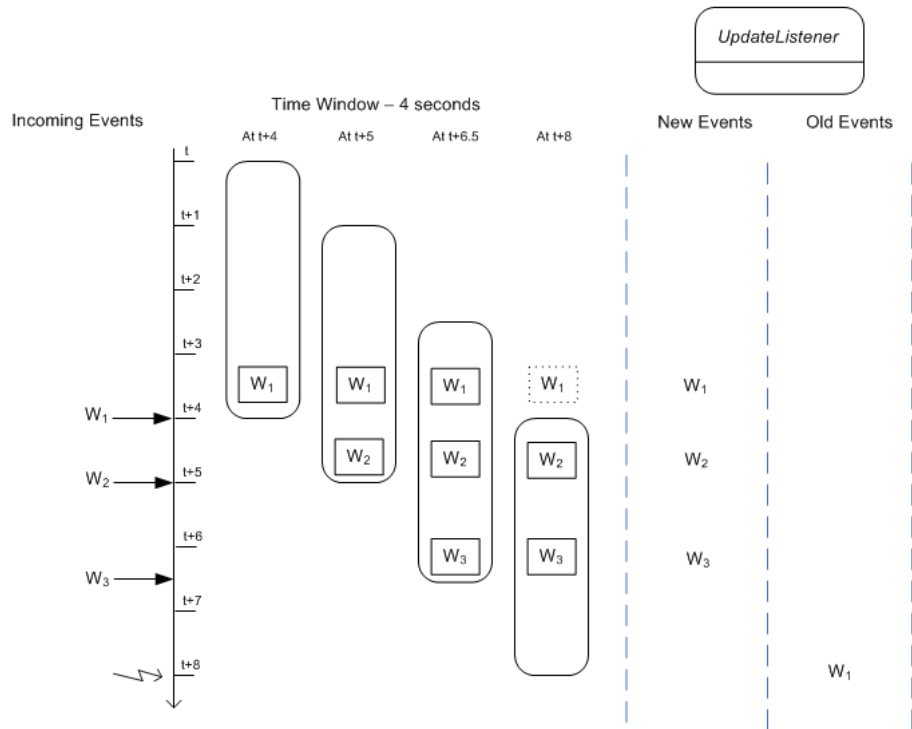


Figure 3.5. Output example for a statement with a time window

The activity as illustrated by the diagram:

1. At time $t + 4$ seconds an event w_1 arrives and enters the time window. The engine reports the new event to update listeners.
2. At time $t + 5$ seconds an event w_2 arrives and enters the time window. The engine reports the new event to update listeners.
3. At time $t + 6.5$ seconds an event w_3 arrives and enters the time window. The engine reports the new event to update listeners.
4. At time $t + 8$ seconds event w_1 leaves the time window. The engine reports the event as an old event to update listeners.

3.5.2. Time Batch

The time batch view buffers events and releases them every specified time interval in one update. Time windows control the evaluation of events, as does the length batch window.

The next diagram serves to illustrate the functioning of a time batch view. For the diagram, we assume a simple

query as below:

```
select * from Withdrawal.win:time_batch(4 sec)
```

The diagram starts at a given time t and displays the contents of the time window at $t + 4$ and $t + 5$ seconds and so on.

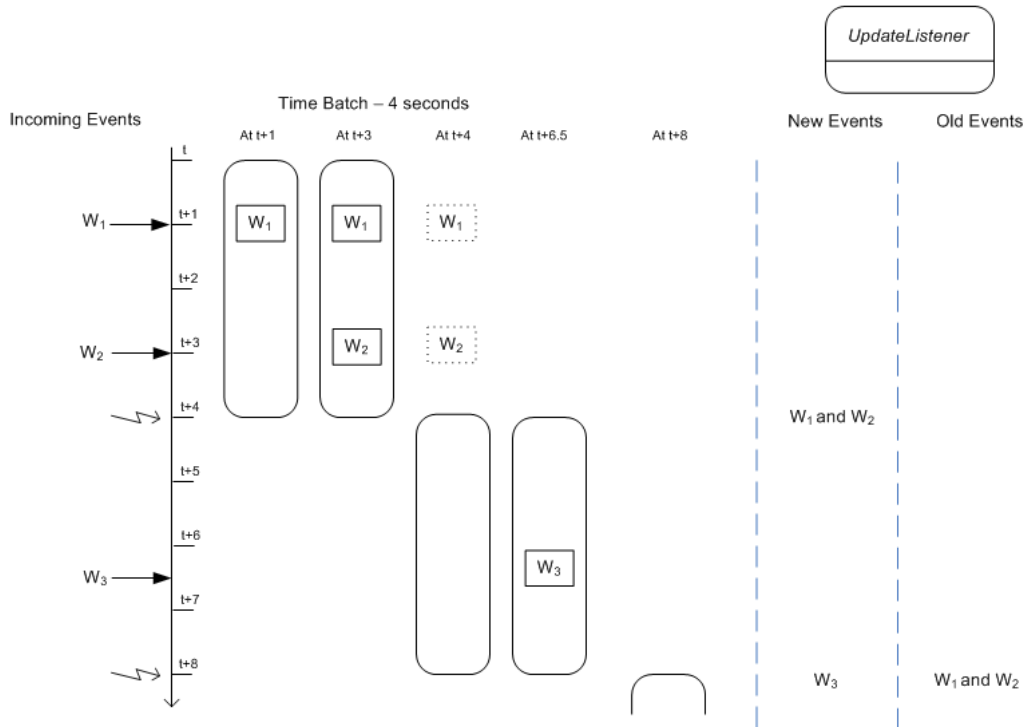


Figure 3.6. Output example for a statement with a time batch view

The activity as illustrated by the diagram:

1. At time $t + 1$ seconds an event w_1 arrives and enters the batch. No call to inform update listeners occurs.
2. At time $t + 3$ seconds an event w_2 arrives and enters the batch. No call to inform update listeners occurs.
3. At time $t + 4$ seconds the engine processes the batched events and starts a new batch. The engine reports events w_1 and w_2 to update listeners.
4. At time $t + 6.5$ seconds an event w_3 arrives and enters the batch. No call to inform update listeners occurs.
5. At time $t + 8$ seconds the engine processes the batched events and starts a new batch. The engine reports the event w_3 as new data to update listeners. The engine reports the events w_1 and w_2 as old data (prior batch) to update listeners.

3.6. Batch Windows

The built-in data windows that act on batches of events are the `win:time_batch` and the `win:length_batch` views. The `win:time_batch` data window collects events arriving during a given time interval and posts collec-

ted events as a batch to listeners at the end of the time interval. The `win:length_batch` data window collects a given number of events and posts collected events as a batch to listeners when the given number of events has collected.

Related to batch data windows is output rate limiting. While batch data windows retain events the `output` clause offered by output rate limiting can control or stabilize the rate at which events are output, see Section 4.7, “Stabilizing and Controlling Output: the Output Clause”.

Let's look at how a time batch window may be used:

```
select account, amount from Withdrawal.win:time_batch(1 sec)
```

The above statement collects events arriving during a one-second interval, at the end of which the engine posts the collected events as new events (insert stream) to each listener. The engine posts the events collected during the prior batch as old events (remove stream). The engine starts posting events to listeners one second after it receives the first event and thereon.

For statements containing aggregation functions and/or a `group by` clause, the engine posts consolidated aggregation results for an event batch. For example, consider the following statement:

```
select sum(amount) as mysum from Withdrawal.win:time_batch(1 sec)
```

Note that output rate limiting also generates batches of events following the output model as discussed here.

3.7. Aggregation and Grouping

3.7.1. Insert and Remove Stream

Statements that aggregate events via aggregation functions also post remove stream events as aggregated values change.

Consider the following statement that alerts when 2 Withdrawal events have been received:

```
select count(*) as mycount from Withdrawal having count(*) = 2
```

When the engine encounters the second withdrawal event, the engine posts a new event to update listeners. The value of the "mycount" property on that new event is 2. Additionally, when the engine encounters the third Withdrawal event, it posts an old event to update listeners containing the prior value of the count, if specifying the `rstream` keyword in the select clause to select the remove stream. The value of the "mycount" property on that old event is also 2.

Note the statement above does not specify a data window and thereby counts all arriving events since statement start. The statement above retains no events and its memory allocation is only the aggregation state, i.e. a single long value to represent `count(*)`.

The `istream` or `rstream` keyword can be used to eliminate either new events or old events posted to listeners. The next statement uses the `istream` keyword causing the engine to call the listener only once when the second Withdrawal event is received:

```
select istream count(*) as mycount from Withdrawal having count(*) = 2
```

3.7.2. Output for Aggregation and Group-By

Following SQL (Standard Query Language) standards for queries against relational databases, the presence or absence of aggregation functions and the presence or absence of the `group by` clause dictates the number of rows posted by the engine to listeners. The next sections outline the output model for batched events under aggregation and grouping. The examples also apply to data windows that don't batch events and post results continuously as events arrive or leave data windows. The examples also apply to patterns providing events when a complete pattern matches.

In summary, as in SQL, if your query only selects aggregation values, the engine provides one row of aggregated values. It provides that row every time the aggregation is updated (insert stream), which is when events arrive or a batch of events gets processed, and when the events leave a data window or a new batch of events arrives. The remove stream then consists of prior aggregation values.

Also as in SQL, if your query selects non-aggregated values along with aggregation values in the select clause, the engine provides a row per event. The insert stream then consists of the aggregation values at the time the event arrives, while the remove stream is the aggregation value at the time the event leaves a data window, if any is defined in your query.

The documentation provides output examples for query types in Appendix A, *Output Reference and Samples*, and the next sections outlines each query type.

Un-aggregated and Un-grouped

An example statement for the un-aggregated and un-grouped case is as follows:

```
select * from Withdrawal.win:time_batch(1 sec)
```

At the end of a time interval, the engine posts to listeners one row for each event arriving during the time interval.

The appendix provides a complete example including input and output events over time at Section A.2, “Output for Un-aggregated and Un-grouped Queries”

Fully Aggregated and Un-grouped

If your statement only selects aggregation values and does not group, your statement may look as the example below:

```
select sum(amount)
from Withdrawal.win:time_batch(1 sec)
```

At the end of a time interval, the engine posts to listeners a single row indicating the aggregation result. The aggregation result aggregates all events collected during the time interval.

The appendix provides a complete example including input and output events over time at Section A.3, “Output for Fully-aggregated and Un-grouped Queries”

Aggregated and Un-Grouped

If your statement selects non-aggregated properties and aggregation values, and does not group, your statement may be similar to this statement:

```
select account, sum(amount)
from Withdrawal.win:time_batch(1 sec)
```

At the end of a time interval, the engine posts to listeners one row per event. The aggregation result aggregates all events collected during the time interval.

The appendix provides a complete example including input and output events over time at Section A.4, “Output for Aggregated and Un-grouped Queries”

Fully Aggregated and Grouped

If your statement selects aggregation values and all non-aggregated properties in the `select` clause are listed in the `group by` clause, then your statement may look similar to this example:

```
select account, sum(amount)
from Withdrawal.win:time_batch(1 sec)
group by account
```

At the end of a time interval, the engine posts to listeners one row per unique account number. The aggregation result aggregates per unique account.

The appendix provides a complete example including input and output events over time at Section A.5, “Output for Fully-aggregated and Grouped Queries”

Aggregated and Grouped

If your statement selects non-aggregated properties and aggregation values, and groups only some properties using the `group by` clause, your statement may look as below:

```
select account, accountName, sum(amount)
from Withdrawal.win:time_batch(1 sec)
group by account
```

At the end of a time interval, the engine posts to listeners one row per event. The aggregation result aggregates per unique account.

The appendix provides a complete example including input and output events over time at Section A.6, “Output for Aggregated and Grouped Queries”

3.8. Event Visibility and Current Time

An event sent by your application or generated by statements is visible to all other statements in the same engine instance. Similarly, current time (the time horizon) moves forward for all statements in the same engine instance. Please see the Chapter 12, *API Reference* chapter for how to send events and how time moves forward through system time or via simulated time, and the possible threading models.

Within an Esper engine instance you can additionally control event visibility and current time on a statement level, under the term *isolated service* as described in Section 12.9, “Service Isolation”.

An isolated service provides a dedicated execution environment for one or more statements. Events sent to an isolated service are visible only within that isolated service. In the isolated service you can move time forward at the pace and resolution desired without impacting other statements that reside in the engine runtime or other isolated services. You can move statements between the engine and an isolated service.

Chapter 4. EPL Reference: Clauses

4.1. EPL Introduction

The Event Processing Language (EPL) is a SQL-like language with `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING` and `ORDER BY` clauses. Streams replace tables as the source of data with events replacing rows as the basic unit of data. Since events are composed of data, the SQL concepts of correlation through joins, filtering and aggregation through grouping can be effectively leveraged.

The `INSERT INTO` clause is recast as a means of forwarding events to other streams for further downstream processing. External data accessible through JDBC may be queried and joined with the stream data. Additional clauses such as the `PATTERN` and `OUTPUT` clauses are also available to provide the missing SQL language constructs specific to event processing.

The purpose of the `UPDATE` clause is to update event properties. Update takes place before an event applies to any selecting statements or pattern statements.

EPL statements are used to derive and aggregate information from one or more streams of events, and to join or merge event streams. This section outlines EPL syntax. It also outlines the built-in views, which are the building blocks for deriving and aggregating information from event streams.

EPL statements contain definitions of one or more views. Similar to tables in a SQL statement, views define the data available for querying and filtering. Some views represent windows over a stream of events. Other views derive statistics from event properties, group events or handle unique event property values. Views can be staggered onto each other to build a chain of views. The Esper engine makes sure that views are reused among EPL statements for efficiency.

The built-in set of views is:

1. Data window views: `win:length`, `win:length_batch`, `win:time`, `win:time_batch`, `win:time_length_batch`, `win:time_accum`, `win:ext_timed`, `ext:sort_window`, `ext:time_order`, `std:unique`, `std:groupwin`, `std:lastevent`, `std:firstevent`, `std:firstunique`, `win:firstlength`, `win:firsttime`.
2. Views that derive statistics: `std:size`, `stat:uni`, `stat:linest`, `stat:correl`, `stat:weighted_avg`.

EPL provides the concept of *named window*. Named windows are data windows that can be inserted-into and deleted-from by one or more statements, and that can queried by one or more statements. Named windows have a global character, being visible and shared across an engine instance beyond a single statement. Use the `CREATE WINDOW` clause to create named windows. Use the `ON MERGE` clause to atomically merge events into named window state, the `INSERT INTO` clause to insert data into a named window, the `ON DELETE` clause to remove events from a named window, the `ON UPDATE` clause to update events held by a named window and the `ON SELECT` clause to perform a query triggered by a pattern or arriving event on a named window. Finally, the name of the named window can occur in a statement's `FROM` clause to query a named window or include the named window in a join or subquery.

EPL allows execution of on-demand (fire-and-forget, non-continuous, triggered by API) queries against named windows through the runtime API. The query engine automatically indexes named window data for fast access by `ON SELECT/UPDATE/INSERT/DELETE` without the need to create an index explicitly. For fast on-demand query execution via runtime API use the `CREATE INDEX` syntax to create an explicit index.

Use `CREATE SCHEMA` to declare an event type.

Variables can come in handy to parameterize statements and change parameters on-the-fly and in response to events. Variables can be used in an expression anywhere in a statement as well as in the output clause for dynamic control of output rates.

Esper can be extended by plugging-in custom developed views and aggregation functions.

4.2. EPL Syntax

EPL queries are created and stored in the engine, and publish results to listeners as events are received by the engine or timer events occur that match the criteria specified in the query. Events can also be obtained from running EPL queries via the `GetSafeEnumerator` and `GetEnumerator` methods that provide a pull-data API.

The `select` clause in an EPL query specifies the event properties or events to retrieve. The `from` clause in an EPL query specifies the event stream definitions and stream names to use. The `where` clause in an EPL query specifies search conditions that specify which event or event combination to search for. For example, the following statement returns the average price for IBM stock ticks in the last 30 seconds.

```
select avg(price) from StockTick.win:time(30 sec) where symbol='IBM'
```

EPL queries follow the below syntax. EPL queries can be simple queries or more complex queries. A simple select contains only a `select` clause and a single stream definition. Complex EPL queries can be build that feature a more elaborate select list utilizing expressions, may join multiple streams, may contain a `where` clause with search conditions and so on.

```
[annotations]
[expression_declarations]
[insert into insert_into_def]
select select_list
from stream_def [as name] [, stream_def [as name]] [,...]
[where search_conditions]
[group by grouping_expression_list]
[having grouping_search_conditions]
[output output_specification]
[order by order_by_expression_list]
[limit num_rows]
```

4.2.1. Specifying Time Periods

Time-based windows as well as pattern observers and guards take a time period as a parameter. Time periods follow the syntax below.

```
time-period : [year-part] [month-part] [week-part] [day-part] [hour-part]
             [minute-part] [seconds-part] [milliseconds-part]

year-part : (number/variable_name) ("years" | "year")
month-part : (number/variable_name) ("months" | "month")
week-part : (number/variable_name) ("weeks" | "week")
day-part : (number/variable_name) ("days" | "day")
hour-part : (number/variable_name) ("hours" | "hour")
minute-part : (number/variable_name) ("minutes" | "minute" | "min")
seconds-part : (number/variable_name) ("seconds" | "second" | "sec")
milliseconds-part : (number/variable_name) ("milliseconds" | "millisecond" | "msec")
```

Some examples of time periods are:

```
10 seconds
10 minutes 30 seconds
```

```
20 sec 100 msec
1 day 2 hours 20 minutes 15 seconds 110 milliseconds
0.5 minutes
1 year
1 year 1 month
```

Variable names and substitution parameters '?' for prepared statements are also allowed as part of a time period expression.

A unit in the month part is equivalent to 30 days.

4.2.2. Using Comments

Comments can appear anywhere in the EPL or pattern statement text where whitespace is allowed. Comments can be written in two ways: slash-slash (`// ...`) comments and slash-star (`/* ... */`) comments.

Slash-slash comments extend to the end of the line:

```
// This comment extends to the end of the line.
// Two forward slashes with no whitespace between them begin such comments.

select * from MyEvent  // this is a slash-slash comment

// All of this text together is a valid statement.
```

Slash-star comments can span multiple lines:

```
/* This comment is a "slash-star" comment that spans multiple lines.
 * It begins with the slash-star sequence with no space between the '/' and '*' characters.
 * By convention, subsequent lines can begin with a star and are aligned, but this is
 * not required.
 */
select * from MyEvent  /* this also works */
```

Comments styles can also be mixed:

```
select field1, // first comment
/* second comment*/ field2
from MyEvent
```

4.2.3. Reserved Keywords

Certain words such as `select`, `delete` or `set` are reserved and may not be used as identifiers. Please consult Appendix B, *Reserved Keywords* for the list of reserved keywords and permitted keywords.

Names of built-in functions and certain auxiliary keywords are permitted as event property names and in the `rename` syntax of the `select` clause. For example, `count` is acceptable.

Consider the example below, which assumes that `'last'` is an event property of `MyEvent`:

```
// valid
select last, count(*) as count from MyEvent
```

This example shows an incorrect use of a reserved keyword:

```
// invalid
select insert from MyEvent
```

EPL offers an escape syntax for reserved keywords: Event properties as well as event or stream names may be escaped via the backwards apostrophe ` (ASCII 96) character.

The next example queries an event type by name `Order` (a reserved keyword) that provides a property by name `insert` (a reserved keyword):

```
// valid
select `insert` from `Order`
```

4.2.4. Escaping Strings

You may surround string values by either double-quotes (") or single-quotes ('). When your string constant in an EPL statement itself contains double quotes or single quotes, you must escape the quotes.

Double and single quotes may be escaped by the backslash (\) character or by unicode notation. Unicode 0027 is a single quote (') and 0022 is a double quote (").

The sample EPL below escapes the single quote in the string constant `John's`, and filters out order events where the name value matches:

```
select * from OrderEvent(name='John\'s')
// ...equivalent to...
select * from OrderEvent(name='John\u0027s')
```

The next EPL escapes the string constant `Quote "Hello"`:

```
select * from OrderEvent(description like "Quote \"Hello\"")
// is equivalent to
select * from OrderEvent(description like "Quote \u0022Hello\u0022")
```

When building an escape string via the API, escape the backslash, as shown in below code snippet:

```
epService.getEPAdministrator().createEPL("select * from OrderEvent(name='John\\'s')");
// ... and for double quotes...
epService.getEPAdministrator().createEPL("select * from OrderEvent(
    description like \"Quote \\\"Hello\\\"\"");
```

4.2.5. Data Types

EPL honors all built-in primitive and boxed types.

EPL also follows CLR standards in terms of widening, performing widening automatically in cases where widening type conversion is allowed without loss of precision, for both boxed and primitive types:

1. byte to short, int, long, float, double, decimal
2. short to int, long, float, or double, decimal
3. char to int, long, float, or double, decimal
4. int to long, float, or double, decimal
5. long to float or double, decimal
6. float to double or decimal
7. double to decimal

In cases where loss of precision is possible because of narrowing requirements, EPL compilation outputs a compilation error.

EPL supports casting via the `cast` function.

EPL returns double-type values for division regardless of operand type. EPL can also be configured to follow CLR rules for integer arithmetic instead as described in Section 13.4.18, “Engine Settings related to Expression Evaluation”.

Division by zero returns positive or negative infinity. Division by zero can be configured to return null instead.

Data Type of Constants

An EPL constant is a number or a character string that indicates a fixed value. Constants can be used as expressions in many EPL statements, including variable assignment and case-when statements. They can also be used as parameter values for many built-in objects and clauses. Constants are also called literals.

EPL supports the standard SQL constant notation as well as data type literals.

The following are types of EPL constants:

Table 4.1. Types of EPL constants

Type	Description	Examples
string	A single character to an unlimited number of characters. Valid delimiters are the single quote (') or double quote (").	<pre>select 'volume' as field1, "sleep" as field2, "\u0041" as unicodeA</pre>
boolean	A boolean value.	<pre>select true as field1, false as field2</pre>
integer	An integer value (4 byte).	<pre>select 1 as field1, -1 as field2, 1e2 as field3</pre>
long	A long value (8 byte). Use the "L" or "l" (lowercase L) suffix.	<pre>select 1L as field1, 1l as field2</pre>
double	A double-precision 64-bit IEEE 754 floating point.	<pre>select 1.67 as field1, 167e-2 as field2, 1.67d as field3</pre>
float	A single-precision 32-bit IEEE 754 floating point. Use the "f" suffix.	<pre>select 1.2f as field1, 1.2F as field2</pre>
byte	A 8-bit signed two's complement integer.	<pre>select 0x10 as field1</pre>

EPL does not have a single-byte character data type for its literals. Single character literals are treated as string.

Internal byte representation and boundary values of constants follow the CLR standard.

4.2.6. Annotation

An annotation is an addition made to information in a statement. Esper provides certain built-in annotations for defining statement name, adding a statement description or for tagging statements such as for managing statements or directing statement output. Other than the built-in annotations, applications can provide their own annotation classes that the EPL compiler can populate.

An annotation is part of the statement text and precedes the EPL select or pattern statement. Annotations are therefore part of the EPL grammar. The syntax for annotations follows the host language (Java, .NET) annotation syntax:

```
@annotation_name [(annotation_parameters)]
```

An annotation consists of the annotation name and optional annotation parameters. The *annotation_name* is the simple class name or fully-qualified class name of the annotation class. The optional *annotation_parameters* are a list of key-value pairs following the syntax:

```
@annotation_name (attribute_name = attribute_value, [name=value, ...])
```

The *attribute_name* is an identifier that must match the attributes defined by the annotation class. An *attribute_value* is a constant of any of the primitive types or string, an array, an enumeration value or another (nested) annotation. Null values are not allowed as annotation attribute values. Enumeration values are supported in EPL statements and not support in statements created via the `CreatePattern` method.

Use the `Attributes` property of `EPStatement` to obtain annotations provided via statement text.

Application-Provided Annotations

Your application may provide its own annotation classes. The engine detects and populates annotation instances for application annotation classes.

To enable the engine to recognize application annotation classes, your annotation name must include the package name (i.e. be fully-qualified) or your engine configuration must import the annotation class or package via the configuration API.

For example, assume that your application defines an annotation in its application code as follows:

```
public @interface ProcessMonitor {
    String processName();
    boolean isLongRunning default false;
    int[] subProcessIds;
}
```

Shown next is an EPL statement text that utilizes the annotation class defined earlier:

```
@ProcessMonitor(processName='CreditApproval',
    isLongRunning=true, subProcessIds = {1, 2, 3} )
select count(*) from ProcessEvent(processId in (1, 2, 3).win:time(30))
```

Above example assumes the `ProcessMonitor` annotation class is imported via configuration XML or API. Here is an example API call to import annotations provided by a package `com.mycompany.myannotations`:

```
epService.getEPAdministrator().getConfiguration().addImport("com.mycompany.myannotations.*");
```

Built-In Annotations

The list of built-in EPL annotations is:

Table 4.2. Built-In EPL Annotations

Name	Purpose and Attributes	Example
Name	Provides a statement name. Attributes are: value : Statement name.	<pre>@Name("MyStatementName")</pre>
Description	Provides a statement textual description. Attributes are: value : Statement description.	<pre>@Description("A statement description is placed here.")</pre>
Tag	For tagging a statement with additional information. Attributes are: name : Tag name. value : Tag value.	<pre>@Tag(name="MyTagName" , value="MyTagValue")</pre>
Priority	Applicable when an event (or schedule) matches filter criteria for multiple statements: Defines the order of statement processing (requires an engine-level setting). Attributes are: value : priority value.	<pre>@Priority(10)</pre>
Drop	Applicable when an event (or schedule) matches filter criteria for multiple statements, drops the event after processing the statement (requires an engine-level setting). No attributes.	<pre>@Drop</pre>
Hint	For providing one or more hints towards how the engine should execute a statement. Attributes are: value : A comma-separated list of one or more case-insensitive keywords.	<pre>@Hint('ITERATE_ONLY')</pre>
Hook	Use this annotation to register one or more statement-specific hooks providing a hook type for each individual hook, such as for SQL parameter, column or row conversion.	<pre>@Hook(type=HookType.SQLCOL, hook='MyDB'</pre>

Name	Purpose and Attributes	Example
	Attributes are the hook <code>type</code> and the hook itself (usually a import or class name):	
Audit	Causes the engine to output detailed processing information for a statement. optional value : A comma-separated list of one or more case-insensitive keywords.	<code>@Audit</code>

The following example statement text specifies some of the built-in annotations in combination:

```
@Name("RevenuePerCustomer")
@Description("Outputs revenue per customer considering all events encountered so far.")
@Tag(name="grouping", value="customer")

select customerId, sum(revenue) from CustomerRevenueEvent
```

@Name

Use the `@Name` EPL annotation to specify a statement name within the EPL statement itself, as an alternative to specifying the statement name via API.

If your application is also providing a statement name through the API, the statement name provided through the API overrides the annotation-provided statement name.

Example:

```
@Name("SecurityFilter1") select * from SecurityFilter(ip="127.0.0.1")
```

@Description

Use the `@Description` EPL annotation to add a statement textual description.

Example:

```
@Description('This statement filters localhost.') select * from SecurityFilter(ip="127.0.0.1")
```

@Tag

Use the `@Tag` EPL annotation to tag statements with name-value pairs, effectively adding a property to the statement. The attributes `name` and `value` are of type string.

Example:

```
@Tag(name='ip_sensitive', value='Y')
@Tag(name='author', value='Jim')
select * from SecurityFilter(ip="127.0.0.1")
```

@Priority

This annotation only takes effect if the engine-level setting for prioritized execution is set via configuration, as

described in Section 13.4.19, “Engine Settings related to Execution of Statements”.

Use the `@Priority` EPL annotation to tag statements with a priority value. The default priority value is zero (0) for all statements. When an event (or single timer execution) requires processing the event for multiple statements, processing begins with the highest priority statement and ends with the lowest-priority statement.

Example:

```
@Priority(10) select * from SecurityFilter(ip="127.0.0.1")
```

@Drop

This annotation only takes effect if the engine-level setting for prioritized execution is set via configuration, as described in Section 13.4.19, “Engine Settings related to Execution of Statements”.

Use the `@Drop` EPL annotation to tag statements that preempt all other same or lower-priority statements. When an event (or single timer execution) requires processing the event for multiple statements, processing begins with the highest priority statement and ends with the first statement marked with `@Drop`, which becomes the last statement to process that event.

Unless a different priority is specified, the statement with the `@Drop` EPL annotation executes at priority 1. Thereby `@Drop` alone is an effective means to remove events from a stream.

Example:

```
@Drop select * from SecurityFilter(ip="127.0.0.1")
```

@Hint

A hint can be used to provide tips for the engine to affect statement execution. Hints change performance or memory-use of a statement but generally do not change its output.

The string value of a `Hint` annotation contains a keyword or a comma-separated list of multiple keywords. Hint keywords are case-insensitive. A list of hints is available in Section 17.2.22, “Consider using Hints”.

Example:

```
@Hint('disable_reclaim_group')  
select ipaddress, count(*) from SecurityFilter.win:time(60 sec) group by ipaddress
```

@Hook

A hook is for attaching a callback to a statement.

The type value of a `Hook` annotation defines the type of hook and the `hook` value is an imported or fully-qualified class name providing the callback implementation.

@Audit

Causes the engine to output detailed information about the statements processing. Described in more detail at Section 14.3.1, “@Audit Annotation”.

4.2.7. Expression Declaration

An EPL statement can contain expression declarations. Expressions that are common to multiple places in an EPL statement can be moved to a named expression declaration and reused within the statement without duplicating the expression itself.

An expression declaration follows the lambda-style expression syntax. This syntax was chosen as it typically allows for a shorter and more concise expression body that can be easier to read than most procedural code.

The syntax for an expression declaration is:

```
expression expression_name { expression_body }
```

An expression declaration consists of the expression name and an expression body. The *expression_name* is any identifier. The *expression_body* contains optional parameters and the expression. The parameter types and the return type of the expression is determined by the engine and do not need to be specified.

Please note the following limitation: Parameters to a declared expression can only be a stream name. For limitations as to the type of expression that can be used in the expression body please see below.

In the expression body the `=>` lambda operator reads as "goes to". The left side of the lambda operator specifies the input parameters (if any) and the right side holds the expression. The lambda expression `x => x * x` is read "x goes to x times x".

In the expression body, if your expression takes no parameters, you may simply specify the expression and do not need the `=>` lambda operator.

If your expression takes one parameters, specify the input parameter name followed by the `=>` lambda operator and followed by the expression. The synopsis for use with a single input parameter is:

```
expression_body:    input_param_name => expression
```

If your expression takes two or more parameters, specify the input parameter names in parenthesis followed by the `=>` lambda operator followed by the expression. The synopsis for use with a multiple input parameter is:

```
expression_body:    (input_param [,input_param [...]]) => expression
```

The following example declares an expression that returns two times PI (ratio of the circumference of a circle to its diameter) and demonstrates its use in a select-clause:

```
expression twoPI { Math.PI * 2 } select twoPI() from SampleEvent
```

The next example declares an expression that accepts one parameter: a `MarketData` event. The expression computes a new "mid" price based on the buy and sell price:

```
expression midPrice { x => x.buy + x.sell / 2 }  
select midPrice(md) from MarketDataEvent as md
```

A further example that demonstrates two parameters is listed next. The example joins two streams and uses the price value from `MarketDataEvent` and the sentiment value of `NewsEvent` to compute a weighted sentiment:

```
expression weightedSentiment { (x, y) => x.price * y.sentiment }  
select weightedSentiment(md, news)  
from MarketDataEvent.std:lastevent() as md, NewsEvent.std:lastevent() news
```

Any expression can be used in the expression body including aggregations, variables, subqueries or further declared expressions. Sub-queries, when used without `in` or `exists`, must be placed within parenthesis.

An example subquery within a declared expression is shown next:

```
expression newsSubq(md) {
  (select sentiment from NewsEvent.std:unique(symbol) where symbol = md.symbol)
}
select newsSubq(mdstream)
from MarketDataEvent mdstream
```

When using declared expressions please note these limitations:

1. Parameters to a declared expression can currently only be a stream name.

The following scope rules apply for declared expressions:

1. The scope of the expression body of a declared expression only includes the parameters explicitly listed.

4.3. Choosing Event Properties And Events: the *Select* Clause

The `select` clause is required in all EPL statements. The `select` clause can be used to select all properties via the wildcard `*`, or to specify a list of event properties and expressions. The `select` clause defines the event type (event property names and types) of the resulting events published by the statement, or pulled from the statement via the iterator methods.

The `select` clause also offers optional `istream`, `irstream` and `rstream` keywords to control whether input stream, remove stream or input and remove stream events are posted to `UpdateEventHandler` instances and observers to a statement. By default, the engine provides only the insert stream to listener and observers. See Section 13.4.14, “Engine Settings related to Stream Selection” on how to change the default.

The syntax for the `select` clause is summarized below.

```
select [istream | irstream | rstream] [distinct] * | expression_list ...
```

The `istream` keyword is the default, and indicates that the engine only delivers insert stream events to listeners and observers. The `irstream` keyword indicates that the engine delivers both insert and remove stream. Finally, the `rstream` keyword tells the engine to deliver only the remove stream.

The `distinct` keyword outputs only unique rows depending on the column list you have specified after it. It must occur after the `select` and after the optional stream keywords, as described in more detail below.

4.3.1. Choosing all event properties: `select *`

The syntax for selecting all event properties in a stream is:

```
select * from stream_def
```

The following statement selects `StockTick` events for the last 30 seconds of IBM stock ticks.

```
select * from StockTick(symbol='IBM').win:time(30 sec)
```

You may well be asking: Why does the statement specify a time window here? First, the statement is meant to demonstrate the use of `*` wildcard. When the engine pushes statement results to your listener and as the statement does not select remove stream events via `rstream` keyword, the listener receives only new events and the time window could be left off. By adding the time window the pull API (iterator API or JDBC driver) returns

the last 30 seconds of events.

The `*` wildcard and expressions can also be combined in a `select` clause. The combination selects all event properties and in addition the computed values as specified by any additional expressions that are part of the `select` clause. Here is an example that selects all properties of stock tick events plus a computed product of price and volume that the statement names 'pricevolume':

```
select *, price * volume as pricevolume from StockTick
```

When using wildcard (`*`), Esper does not actually copy your event properties out of your event or events. It simply wraps your native type in an `EventBean` interface. Your application has access to the underlying event object through the `Underlying` property and has access to the property values through the `Get` method.

In a join statement, using the `select *` syntax selects one event property per stream to hold the event for that stream. The property name is the stream name in the `from` clause.

4.3.2. Choosing specific event properties

To choose the particular event properties to return:

```
select event_property [, event_property] [, ...] from stream_def
```

The following statement simply selects the symbol and price properties of stock ticks, and the total volume for stock tick events in a 60-second time window.

```
select symbol, price, sum(volume) from StockTick(symbol='IBM').win:time(60 sec)
```

The following statement declares a further view onto the event stream of stock ticks: the univariate statistics view (`stat:uni`). The statement selects the properties that this view derives from the stream, for the last 100 events of IBM stock ticks in the length window.

```
select datapoints, total, average, variance, stddev, stddevpa  
from StockTick(symbol='IBM').win:length(100).stat:uni(volume)
```

4.3.3. Expressions

The `select` clause can contain one or more expressions.

```
select expression [, expression] [, ...] from stream_def
```

The following statement selects the volume multiplied by price for a time batch of the last 30 seconds of stock tick events.

```
select volume * price from StockTick.win:time_batch(30 sec)
```

4.3.4. Renaming event properties

Event properties and expressions can be renamed using below syntax.

```
select [event_property | expression] as identifier [, ...]
```

The following statement selects volume multiplied by price and specifies the name *volPrice* for the resulting column.

```
select volume * price as volPrice from StockTick
```

Identifiers cannot contain the "." (dot) character, i.e. "vol.price" is not a valid identifier for the rename syntax.

4.3.5. Choosing event properties and events in a join

If your statement is joining multiple streams, you may specify property names that are unique among the joined streams, or use wildcard (*) as explained earlier.

In case the property name in your `select` or other clauses is not unique considering all joined streams, you will need to use the name of the stream as a prefix to the property.

This example is a join between the two streams `StockTick` and `News`, respectively named as 'tick' and 'news'. The example selects from the `StockTick` event the `symbol` value using the 'tick' stream name as a prefix:

```
select tick.symbol from StockTick.win:time(10) as tick, News.win:time(10) as news
where news.symbol = tick.symbol
```

Use the wildcard (*) selector in a join to generate a property for each stream, with the property value being the event itself. The output events of the statement below have two properties: the 'tick' property holds the `StockTick` event and the 'news' property holds the `News` event:

```
select * from StockTick.win:time(10) as tick, News.win:time(10) as news
```

The following syntax can also be used to specify what stream's properties to select:

```
select stream_name.* [as name] from ...
```

The selection of `tick.*` selects the `StockTick` stream events only:

```
select tick.* from StockTick.win:time(10) as tick, News.win:time(10) as news
where tick.symbol = news.symbol
```

The next example uses the `as` keyword to name each stream's joined events. This instructs the engine to create a property for each named event:

```
select tick.* as stocktick, news.* as news
from StockTick.win:time(10) as tick, News.win:time(10) as news
where stock.symbol = news.symbol
```

The output events of the above example have two properties 'stocktick' and 'news' that are the `StockTick` and `News` events.

The stream name itself, as further described in Section 4.4.5, "Using the Stream Name", may be used within expressions or alone.

This example passes events to a user-defined function named `compute` and also shows `insert-into` to populate an event stream of combined events:

```
insert into TickNewStream select tick, news, MyLib.compute(news, tick) as result
from StockTick.win:time(10) as tick, News.win:time(10) as news
where tick.symbol = news.symbol
```

```
// second statement that uses the TickNewStream stream
select tick.price, news.text, result from TickNewStream
```

In summary, the *stream_name.** streamname wildcard syntax can be used to select a stream as the underlying event or as a property, but cannot appear within an expression. While the *stream_name* syntax (without wildcard) always selects a property (and not as an underlying event), and can occur anywhere within an expression.

4.3.6. Choosing event properties and events from a pattern

If your statement employs pattern expressions, then your pattern expression tags events with a tag name. Each tag name becomes available for use as a property in the `select` clause and all other clauses.

For example, here is a very simple pattern that matches on every `StockTick` event received within 30 seconds after start of the statement. The sample selects the symbol and price properties of the matching events:

```
select tick.symbol as symbol, tick.price as price
from pattern[every tick=StockTick where timer:within(10 sec)]
```

The use of the wildcard selector, as shown in the next statement, creates a property for each tagged event in the output. The next statement outputs events that hold a single 'tick' property whose value is the event itself:

```
select * from pattern[every tick=StockTick where timer:within(10 sec)]
```

You may also select the matching event itself using the `tick.*` syntax. The engine outputs the `StockTick` event itself to listeners:

```
select tick.* from pattern[every tick=StockTick where timer:within(10 sec)]
```

A tag name as specified in a pattern is a valid expression itself. This example uses the `insert into` clause to make available the events matched by a pattern to further statements:

```
// make a new stream of ticks and news available
insert into StockTickAndNews
select tick, news from pattern [every tick=StockTick -> news=News(symbol=tick.symbol)]
```

```
// second statement to select from the stream of ticks and news
select tick.symbol, tick.price, news.text from StockTickAndNews
```

4.3.7. Selecting `insert` and `remove` stream events

The optional `istream`, `istream` and `rstream` keywords in the `select` clause control the event streams posted to listeners and observers to a statement.

If neither keyword is specified, and in the default engine configuration, the engine posts only `insert` stream events via the `NewEvents` property of the `UpdateEventArgs` delivered to `UpdateEventHandler` instances listening to the statement. The engine does not post `remove` stream events, by default.

The `insert` stream consists of the events entering the respective window(s) or stream(s) or aggregations, while the `remove` stream consists of the events leaving the respective window(s) or the changed aggregation result. See Chapter 3, *Processing Model* for more information on `insert` and `remove` streams.

The engine posts `remove` stream events to the `OldEvents` property of the `UpdateEventArgs` delivered to `UpdateEventHandler` instances only if the `istream` keyword occurs in the `select` clause. This behavior can be changed via engine-wide configuration as described in Section 13.4.14, “Engine Settings related to Stream Selection”.

By specifying the `istream` keyword you can instruct the engine to only post insert stream events via the `NewEvents` property to the `UpdateEventHandler`. The engine will then not post any remove stream events, and the `OldEvents` property is always a null value.

By specifying the `irstream` keyword you can instruct the engine to post both insert stream and remove stream events.

By specifying the `rstream` keyword you can instruct the engine to only post remove stream events via the `NewEvents` property to the `UpdateEventHandler`. The engine will then not post any insert stream events, and the `OldEvents` property is also always a null value.

The following statement selects only the events that are leaving the 30 second time window.

```
select rstream * from StockTick.win:time(30 sec)
```

The `istream` and `rstream` keywords in the `select` clause are matched by same-name keywords available in the `insert into` clause. While the keywords in the `select` clause control the event stream posted to listeners to the statement, the same keywords in the `insert into` clause specify the event stream that the engine makes available to other statements.

4.3.8. Qualifying property names and stream names

Property or column names can optionally be qualified by a stream name and the provider URI. The syntax is:

```
[provider_URI.]stream_name.property_name
```

The *provider_URI* is the URI supplied to the `EPServiceProviderManager` class, or the string `default` for the default provider.

This example assumes the provider is the default provider:

```
select MyEvent.myProperty from MyEvent
// ... equivalent to ...
select default.MyEvent.myProperty from MyEvent
```

Stream names can also be qualified by the provider URI. The syntax is:

```
[provider_URI.]stream_name
```

The next example assumes a provider URI by name of `Processor`:

```
select Processor.MyEvent.myProperty from Processor.MyEvent
```

4.3.9. Select `Distinct`

The optional `distinct` keyword removes duplicate output events from output. The keyword must occur after the `select` keyword and after the optional `irstream` keyword.

The `distinct` keyword in your `select` instructs the engine to consolidate, at time of output, the output event(s) and remove output events with identical property values. Duplicate removal only takes place when two or more events are output together at any one time, therefore `distinct` is typically used with a batch data window, output rate limiting, on-demand queries, on-select or iterator pull API.

If two or more output event objects have same property values for all properties of the event, the `distinct` removes all but one duplicated event before outputting events to listeners. Indexed, nested and mapped properties are considered in the comparison, if present in the output event.

The next example outputs sensor ids of temperature sensor events, but only every 10 seconds and only unique sensor id values during the 10 seconds:

```
select distinct sensorId from TemperatureSensorEvent output every 10 seconds
```

Use `distinct` with wildcard (*) to remove duplicate output events considering all properties of an event.

This example statement outputs all distinct events either when 100 events arrive or when 10 seconds passed, whichever occurs first:

```
select distinct * from TemperatureSensorEvent.win:time_length_batch(10, 100)
```

When selecting nested, indexed, mapped or dynamic properties in a `select` clause with `distinct`, it is relevant to know that the comparison uses hash code and the `Equals` semantics.

4.4. Specifying Event Streams: the *From* Clause

The `from` clause is required in all EPL statements. It specifies one or more event streams or named windows. Each event stream or named window can optionally be given a name by means of the `as` keyword.

```
from stream_def [as name] [unidirectional] [retain-union | retain-intersection]
    [, stream_def [as stream_name]] [, ...]
```

The event stream definition *stream_def* as shown in the syntax above can consists of either a filter-based event stream definition or a pattern-based event stream definition.

For joins and outer joins, specify two or more event streams. Joins between pattern-based and filter-based event streams are also supported. Joins and the `unidirectional` keyword are described in more detail in Section 4.12, “Joining Event Streams”.

Esper supports joins against relational databases for access to historical or reference data as explained in Section 4.13, “Accessing Relational Data via SQL”. Esper can also join results returned by an arbitrary method invocation, as discussed in Section 4.14, “Accessing Non-Relational Data via Method Invocation”.

The *stream_name* is an optional identifier assigned to the stream. The stream name can itself occur in any expression and provides access to the event itself from the named stream. Also, a stream name may be combined with a method name to invoke instance methods on events of that stream.

For all streams with the exception of historical sources your query may employ data window views as outlined below. The `retain-intersection` (the default) and `retain-union` keywords build a union or intersection of two or more data windows as described in Section 4.4.4, “Multiple Data Window Views”.

4.4.1. Filter-based Event Streams

The *stream_def* syntax for a filter-based event stream is as below:

```
event_stream_name [(filter_criteria)] [contained_selection] [.view_spec] [.view_spec] [...]
```


The *event_stream_name* is either the name of an event type or name of an event stream populated by an `insert into` statement or the name of a named window.

The *filter_criteria* is optional and consists of a list of expressions filtering the events of the event stream, within parenthesis after the event stream name.

The *contained_selection* is optional and is for use with coarse-grained events that have properties that are themselves one or more events, see Section 4.19, “Contained-Event Selection” for the synopsis and examples.

The *view_spec* are optional view specifications, which are combinable definitions for retaining events and for deriving information from events.

The following EPL statement shows event type, filter criteria and views combined in one statement. It selects all event properties for the last 100 events of IBM stock ticks for volume. In the example, the event type is the fully qualified type name `org.esper.example.StockTick`. The expression filters for events where the property `symbol` has a value of "IBM". The optional view specifications for deriving data from the `StockTick` events are a length window and a view for computing statistics on volume. The name for the event stream is "volumeStats".

```
select * from
  org.esper.example.StockTick(symbol='IBM').win:length(100).stat:uni(volume) as volumeStats
```

Esper filters out events in an event stream as defined by filter criteria before it sends events to subsequent views. Thus, compared to search conditions in a `where` clause, filter criteria remove unneeded events early. In the above example, events with a symbol other than IBM do not enter the time window.

Specifying an Event Type

The simplest form of filter is a filter for events of a given type without any conditions on the event property values. This filter matches any event of that type regardless of the event's properties. The example below is such a filter.

```
select * from com.mypackage.myevents.RfidEvent
```

Instead of the fully-qualified type name any other event name can be mapped via Configuration to a type, making the resulting statement more readable:

```
select * from RfidEvent
```

Interfaces and superclasses are also supported as event types. In the below example `IRfidReadable` is an interface class.

```
select * from org.myorg.rfid.IRfidReadable
```

Specifying Filter Criteria

The filtering criteria to filter for events with certain event property values are placed within parenthesis after the event type name:

```
select * from RfidEvent(category="Perishable")
```

All expressions can be used in filters, including static methods that return a boolean value:

```
select * from com.mycompany.RfidEvent(MyRFIDLib.isInRange(x, y) or (x < 0 and y < 0))
```

Filter expressions can be separated via a single comma ','. The comma represents a logical AND between filter expressions:

```
select * from RfidEvent(zone=1, category=10)
...is equivalent to...
select * from RfidEvent(zone=1 and category=10)
```

The following operators are highly optimized through indexing and are the preferred means of filtering in high-volume event streams:

- equals =
- not equals !=
- comparison operators < , > , >= , <=
- ranges
 - use the `between` keyword for a closed range where both endpoints are included
 - use the `in` keyword and round () or square brackets [] to control how endpoints are included
 - for inverted ranges use the `not` keyword and the `between` or `in` keywords
- list-of-values checks using the `in` keyword or the `not in` keywords followed by a comma-separated list of values

At compile time as well as at run time, the engine scans new filter expressions for sub-expressions that can be indexed. Indexing filter values to match event properties of incoming events enables the engine to match incoming events faster. The above list of operators represents the set of operators that the engine can best convert into indexes. The use of comma or logical `and` in filter expressions does not impact optimizations by the engine.

Filtering Ranges

Ranges come in the following 4 varieties. The use of round () or square [] bracket dictates whether an endpoint is included or excluded. The low point and the high-point of the range are separated by the colon : character.

- Open ranges that contain neither endpoint (low:high)
- Closed ranges that contain both endpoints [low:high]. The equivalent 'between' keyword also defines a closed range.
- Half-open ranges that contain the low endpoint but not the high endpoint [low:high)
- Half-closed ranges that contain the high endpoint but not the low endpoint (low:high]

The next statement shows a filter specifying a range for `x` and `y` values of RFID events. The range includes both endpoints therefore uses [] hard brackets.

```
mypackage.RfidEvent(x in [100:200], y in [0:100])
```

The `between` keyword is equivalent for closed ranges. The same filter using the `between` keyword is:

```
mypackage.RfidEvent(x between 100 and 200, y between 0 and 50)
```

The `not` keyword can be used to determine if a value falls outside a given range:

```
mypackage.RfidEvent(x not in [0:100])
```

The equivalent statement using the `between` keyword is:

```
mypackage.RfidEvent(x not between 0 and 100)
```

Filtering Sets of Values

The `in` keyword for filter criteria determines if a given value matches any value in a list of values.

In this example we are interested in RFID events where the category matches any of the given values:

```
mypackage.RfidEvent(category in ('Perishable', 'Container'))
```

By using the `not in` keywords we can filter events with a property value that does not match any of the values in a list of values:

```
mypackage.RfidEvent(category not in ('Household', 'Electrical'))
```

Filter Limitations

The following restrictions apply to filter criteria:

- Range and comparison operators require the event property to be of a numeric or string type.
- Aggregation functions are not allowed within filter expressions.
- The `prev` previous event function and the `prior` prior event function cannot be used in filter expressions.

4.4.2. Pattern-based Event Streams

Event pattern expressions can also be used to specify one or more event streams in an EPL statement. For pattern-based event streams, the event stream definition *stream_def* consists of the keyword `pattern` and a pattern expression in brackets `[]`. The syntax for an event stream definition using a pattern expression is below. As in filter-based event streams, an optional list of views that derive data from the stream can be supplied.

```
pattern [pattern_expression] [.view_spec] [.view_spec] [...]
```

The next statement specifies an event stream that consists of both stock tick events and trade events. The example tags stock tick events with the name "tick" and trade events with the name "trade".

```
select * from pattern [every tick=StockTickEvent or every trade=TradeEvent]
```

This statement generates an event every time the engine receives either one of the event types. The generated events resemble a map with "tick" and "trade" keys. For stock tick events, the "tick" key value is the underlying stock tick event, and the "trade" key value is a null value. For trade events, the "trade" key value is the underlying trade event, and the "tick" key value is a null value.

Lets further refine this statement adding a view the gives us the last 30 seconds of either stock tick or trade events. Lets also select prices and a price total.

```
select tick.price as tickPrice, trade.price as tradePrice,
       sum(tick.price) + sum(trade.price) as total
from pattern [every tick=StockTickEvent or every trade=TradeEvent].win:time(30 sec)
```

Note that in the statement above `tickPrice` and `tradePrice` can each be null values depending on the event processed. Therefore, an aggregation function such as `sum(tick.price + trade.price)` would always return null values as either of the two price properties are always a null value for any event matching the pattern. Use the `coalesce` function to handle null values, for example: `sum(coalesce(tick.price, 0) + coalesce(trade.price, 0))`.

4.4.3. Specifying Views

Views are used to specify an expiry policy for events (data window views) and also to derive data. Views can be staggered onto each other. See the section Chapter 11, *EPL Reference: Views* on the views available that also outlines the different types of views: Data Window views and Derived-Value views.

Views can optionally take one or more parameters. These parameters are expressions themselves that may consist of any combination of variables, arithmetic, user-defined function or substitution parameters for prepared statements, for example.

The example statement below outputs a count per expressway for car location events (contains information about the location of a car on a highway) of the last 60 seconds:

```
select expressway, count(*) from CarLocEvent.win:time(60)
group by expressway
```

The next example serves to show staggering of views. It uses the `std:groupwin` view to create a separate length window per car id:

```
select cardId, expressway, direction, segment, count(*)
from CarLocEvent.std:groupwin(cardId).win:length(4)
group by cardId, expressway, direction, segment
```

The first view `std:groupwin(cardId)` groups car location events by car id. The second view `win:length(4)` keeps a length window of the 4 last events, with one separate length window for each car id. The example reports the number of events per car id and per expressway, direction and segment considering the last 4 events for each car id only.

Note that the `group by` syntax is generally preferable over `std:groupwin` for grouping information as it is SQL-compliant, easier to read and does not create a separate data window per group. The `std:groupwin` in above example creates a separate data window (length window in the example) per group, demonstrating staggering views.

When views are staggered onto each other as a chain of views, then the insert and remove stream received by each view is the insert and remove stream made available by the view (or stream) earlier in the chain.

The special keep-all view keeps all events: It does not provide a remove stream, i.e. events are not removed from the keep-all view unless by means of the `on-delete` syntax or by revision events.

4.4.4. Multiple Data Window Views

Data window views provide an expiry policy that indicates when to remove events from the data window, with the exception of the keep-all data window which has no expiry policy and the `std:groupwin` grouped-window view for allocating a new data window per group.

EPL allows the freedom to use multiple data window views onto a stream and thus combine expiry policies. Combining data windows into an intersection (the default) or a union can achieve a useful strategy for retaining events and expiring events that are no longer of interest. Named windows and the `on-delete` syntax provide an additional degree of freedom.

In order to combine two or more data window views there is no keyword required. The *retain-intersection* keyword is the default and the *retain-union* keyword may instead be provided for a stream.

The concept of union and intersection come from Set mathematics. In the language of Set mathematics, two

sets A and B can be "added" together: The intersection of A and B is the set of all things which are members of both A and B, i.e. the members two sets have "in common". The union of A and B is the set of all things which are members of either A or B.

Use the *retain-intersection* (the default) keyword to retain an intersection of all events as defined by two or more data windows. All events removed from any of the intersected data windows are entered into the remove stream. This is the default behavior if neither retain keyword is specified.

Use the *retain-union* keyword to retain a union of all events as defined by two or more data windows. Only events removed from all data windows are entered into the remove stream.

As you can see, it is the remove stream that the combined multiple data windows provide which differs when retaining an intersection and retaining a union, the insert stream is the same to all data windows and their staggered views. Therefore, when combining batching data windows with further data windows, the insert stream still remains the insert stream of the set overall (not batched). Consider using `output snapshot` to obtain regular updates instead of combining batch and other data windows.

The next example statement totals the price of `OrderEvent` events in a union of the last 30 seconds and unique by product name:

```
select sum(price) from OrderEvent.win:time(30 sec).std:unique(productName) retain-union
```

In the above statement, all `OrderEvent` events that are either less than 30 seconds old or that are the last event for the product name are considered.

Here is an example statement totals the price of `OrderEvent` events in an intersection of the last 30 seconds and unique by product name:

```
select sum(price) from OrderEvent.win:time(30 sec).std:unique(productName) retain-intersection
```

In the above statement, only those `OrderEvent` events that are both less than 30 seconds old and are the last event for the product name are considered.

For advanced users and for backward compatibility, it is possible to configure Esper to allow multiple data window views without either of the `retain` keywords, as described in Section 13.4.11.2, "Configuring Multi-Expiry Policy Defaults".

4.4.5. Using the Stream Name

Your `from` clause may assign a name to each stream. This assigned stream name can serve any of the following purposes.

First, the stream name can be used to disambiguate property names. The `stream_name.property_name` syntax uniquely identifies which property to select if property names overlap between streams. Here is an example:

```
select prod.productId, ord.productId from ProductEvent as prod, OrderEvent as ord
```

Second, the stream name can be used with a wildcard (*) character to select events in a join, or assign new names to the streams in a join:

```
// Select ProductEvent only
select prod.* from ProductEvent as prod, OrderEvent
```

```
// Assign column names 'product' and 'order' to each event
```

```
select prod.* as product, ord.* as order from ProductEvent as prod, OrderEvent as ord
```

Further, the stream name by itself can occur in any expression: The engine passes the event itself to that expression. For example, the engine passes the ProductEvent and the OrderEvent to the user-defined function 'check-Order':

```
select prod.productId, MyFunc.checkOrder(prod, ord)
from ProductEvent as prod, OrderEvent as ord
```

Last, you may invoke an instance method on each event of a stream, and pass parameters to the instance method as well. Instance method calls are allowed anywhere in an expression.

The next statement demonstrates this capability by invoking a method 'computeTotal' on OrderEvent events and a method 'getMultiplier' on ProductEvent events:

```
select ord.computeTotal(prod.getMultiplier()) from ProductEvent as prod, OrderEvent as ord
```

Instance methods may also be chained: Your EPL may invoke a method on the result returned by a method invocation.

Assume that your product event exposes a method `getZone` which returns a zone object. Assume that the Zone class declares a method `checkZone`. This example statement invokes a method chain:

```
select prod.getZone().checkZone("zone 1") from ProductEvent as prod
```

4.5. Specifying Search Conditions: the *Where* Clause

The *where* clause is an optional clause in EPL statements. Via the *where* clause event streams can be joined and events can be filtered.

Comparison operators `=`, `<`, `>`, `>=`, `<=`, `!=`, `<>`, `is null`, `is not null` and logical combinations via `and` and `or` are supported in the *where* clause. The *where* clause can also introduce join conditions as outlined in Section 4.12, "Joining Event Streams". *where* clauses can also contain expressions. Some examples are listed below.

```
...where fraud.severity = 5 and amount > 500
...where (orderItem.orderId is null) or (orderItem.class != 10)
...where (orderItem.orderId = null) or (orderItem.class <> 10)
...where itemCount / packageCount > 10
```

4.6. Aggregates and grouping: the *Group-by* Clause and the *Having* Clause

4.6.1. Using aggregate functions

The aggregate functions are `sum`, `avg`, `count`, `max`, `min`, `median`, `stddev`, `avedev`. You can use aggregate functions to calculate and summarize data from event properties. For example, to find out the total price for all stock tick events in the last 30 seconds, type:

```
select sum(price) from StockTickEvent.win:time(30 sec)
```

Here is the syntax for aggregate functions:

```
aggregate_function( [all | distinct] expression)
```

You can apply aggregate functions to all events in an event stream window or other view, or to one or more groups of events. From each set of events to which an aggregate function is applied, Esper generates a single value.

Expression is usually an event property name. However it can also be a constant, function, or any combination of event property names, constants, and functions connected by arithmetic operators.

For example, to find out the average price for all stock tick events in the last 30 seconds if the price was doubled:

```
select avg(price * 2) from StockTickEvent.win:time(30 seconds)
```

You can use the optional keyword `distinct` with all aggregate functions to eliminate duplicate values before the aggregate function is applied. The optional keyword `all` which performs the operation on all events is the default.

You can use aggregation functions in a `select` clause and in a `having` clause. You cannot use aggregate functions in a `where` clause, but you can use the `where` clause to restrict the events to which the aggregate is applied. The next query computes the average and sum of the price of stock tick events for the symbol IBM only, for the last 10 stock tick events regardless of their symbol.

```
select 'IBM stats' as title, avg(price) as avgPrice, sum(price) as sumPrice
from StockTickEvent.win:length(10)
where symbol='IBM'
```

In the above example the length window of 10 elements is not affected by the `where` clause, i.e. all events enter and leave the length window regardless of their symbol. If we only care about the last 10 IBM events, we need to add filter criteria as below.

```
select 'IBM stats' as title, avg(price) as avgPrice, sum(price) as sumPrice
from StockTickEvent(symbol='IBM').win:length(10)
where symbol='IBM'
```

You can use aggregate functions with any type of event property or expression, with the following exceptions:

1. You can use `sum`, `avg`, `median`, `stddev`, `avedev` with numeric event properties only

Esper ignores any null values returned by the event property or expression on which the aggregate function is operating, except for the `count(*)` function, which counts null values as well. All aggregate functions return null if the data set contains no events, or if all events in the data set contain only null values for the aggregated expression.

4.6.2. Organizing statement results into groups: the *Group-by* clause

The `group by` clause is optional in all EPL statements. The `group by` clause divides the output of an EPL statement into groups. You can group by one or more event property names, or by the result of computed expressions. When used with aggregate functions, `group by` retrieves the calculations in each subgroup. You can use `group by` without aggregate functions, but generally that can produce confusing results.

For example, the below statement returns the total price per symbol for all stock tick events in the last 30

seconds:

```
select symbol, sum(price) from StockTickEvent.win:time(30 sec) group by symbol
```

The syntax of the `group by` clause is:

```
group by aggregate_free_expression [, aggregate_free_expression] [, ...]
```

Esper places the following restrictions on expressions in the `group by` clause:

1. Expressions in the `group by` cannot contain aggregate functions
2. Event properties that are used within aggregate functions in the `select` clause cannot also be used in a `group by` expression
3. When grouping an unbound stream, i.e. no data window is specified onto the stream providing groups, or when using output rate limiting with the `ALL` keyword, you should ensure your group-by expression does not return an unlimited number of values. If, for example, your group-by expression is a fine-grained timestamp, group state that accumulates for an unlimited number of groups potentially reduces available memory significantly. Use a `@Hint` as described below to instruct the engine when to discard group state.

You can list more than one expression in the `group by` clause to nest groups. Once the sets are established with `group by` the aggregation functions are applied. This statement posts the median volume for all stock tick events in the last 30 seconds per symbol and tick data feed. Esper posts one event for each group to statement listeners:

```
select symbol, tickDataFeed, median(volume)
from StockTickEvent.win:time(30 sec)
group by symbol, tickDataFeed
```

In the statement above the event properties in the `select` list (`symbol`, `tickDataFeed`) are also listed in the `group by` clause. The statement thus follows the SQL standard which prescribes that non-aggregated event properties in the `select` list must match the `group by` columns.

Esper also supports statements in which one or more event properties in the `select` list are not listed in the `group by` clause. The statement below demonstrates this case. It calculates the standard deviation for the last 30 seconds of stock ticks aggregating by symbol and posting for each event the symbol, `tickDataFeed` and the standard deviation on price.

```
select symbol, tickDataFeed, stddev(price) from StockTickEvent.win:time(30 sec) group by symbol
```

The above example still aggregates the `price` event property based on the `symbol`, but produces one event per incoming event, not one event per group.

Additionally, Esper supports statements in which one or more event properties in the `group by` clause are not listed in the `select` list. This is an example that calculates the mean deviation per symbol and `tickDataFeed` and posts one event per group with `symbol` and mean deviation of price in the generated events. Since `tickDataFeed` is not in the posted results, this can potentially be confusing.

```
select symbol, avedev(price)
from StockTickEvent.win:time(30 sec)
group by symbol, tickDataFeed
```

Expressions are also allowed in the `group by` list:

```
select symbol * price, count(*) from StockTickEvent.win:time(30 sec) group by symbol * price
```


If the `group by` expression resulted in a null value, the null value becomes its own group. All null values are aggregated into the same group. If you are using the `count(expression)` aggregate function which does not count null values, the count returns zero if only null values are encountered.

You can use a `where` clause in a statement with `group by`. Events that do not satisfy the conditions in the `where` clause are eliminated before any grouping is done. For example, the statement below posts the number of stock ticks in the last 30 seconds with a volume larger than 100, posting one event per group (symbol).

```
select symbol, count(*) from StockTickEvent.win:time(30 sec) where volume > 100 group by symbol
```

Hints Pertaining to Group-By

The Esper engine reclaims aggregation state aggressively when it determines that a group has no data points, based on the data in the data windows. When your application data creates a large number of groups with a small or zero number of data points then performance may suffer as state is reclaimed and created anew. Esper provides the `@Hint('disable_reclaim_group')` hint that you can specify as part of an EPL statement text to avoid group reclaim.

When aggregating values over an unbound stream (i.e. no data window is specified onto the stream) and when your group-by expression returns an unlimited number of values, for example when a timestamp expression is used, then please note the next hint.

A sample statement that aggregates stock tick events by timestamp, assuming the event type offers a property by name `timestamp` that, reflects time in high resolution, for example arrival or system time:

```
// Note the below statement could lead to an out-of-memory problem:
select symbol, sum(price) from StockTickEvent group by timestamp
```

As the engine has no means of detecting when aggregation state (sums per symbol) can be discarded, you may use the following hints to control aggregation state lifetime.

The `@Hint("reclaim_group_aged=age_in_seconds")` hint instructs the engine to discard aggregation state that has not been updated for `age_in_seconds` seconds.

The optional `@Hint("reclaim_group_freq=sweep_frequency_in_seconds")` can be used in addition to control the frequency at which the engine sweeps aggregation state to determine aggregation state age and remove state that is older than `age_in_seconds` seconds. If the hint is not specified, the frequency defaults to the same value as `age_in_seconds`.

The updated sample statement with both hints:

```
// Instruct engine to remove state older than 10 seconds and sweep every 5 seconds
@Hint('reclaim_group_aged=10,reclaim_group_freq=5')
select symbol, sum(price) from StockTickEvent group by timestamp
```

Variables may also be used to provide values for `age_in_seconds` and `sweep_frequency_in_seconds`.

This example statement uses a variable named `varAge` to control how long aggregation state remains in memory, and the engine defaults the sweep frequency to the same value as the variable provides:

```
@Hint('reclaim_group_aged=varAge')
select symbol, sum(price) from StockTickEvent group by timestamp
```

4.6.3. Selecting groups of events: the *Having* clause

Use the `having` clause to pass or reject events defined by the `group-by` clause. The `having` clause sets conditions for the `group by` clause in the same way `where` sets conditions for the `select` clause, except `where` cannot include aggregate functions, while `having` often does.

This statement is an example of a `having` clause with an aggregate function. It posts the total price per symbol for the last 30 seconds of stock tick events for only those symbols in which the total price exceeds 1000. The `having` clause eliminates all symbols where the total price is equal or less then 1000.

```
select symbol, sum(price)
from StockTickEvent.win:time(30 sec)
group by symbol
having sum(price) > 1000
```

To include more then one condition in the `having` clause combine the conditions with `and`, `or` or `not`. This is shown in the statement below which selects only groups with a total price greater then 1000 and an average volume less then 500.

```
select symbol, sum(price), avg(volume)
from StockTickEvent.win:time(30 sec)
group by symbol
having sum(price) > 1000 and avg(volume) < 500
```

A statement with the `having` clause should also have a `group by` clause. If you omit `group-by`, all the events not excluded by the `where` clause return as a single group. In that case `having` acts like a `where` except that `having` can have aggregate functions.

The `having` clause can also be used without `group by` clause as the below example shows. The example below posts events where the price is less then the current running average price of all stock tick events in the last 30 seconds.

```
select symbol, price, avg(price)
from StockTickEvent.win:time(30 sec)
having price < avg(price)
```

4.6.4. How the stream filter, *Where*, *Group By* and *Having* clauses interact

When you include filters, the `where` condition, the `group by` clause and the `having` condition in an EPL statement the sequence in which each clause affects events determines the final result:

1. The event stream's filter condition, if present, dictates which events enter a window (if one is used). The filter discards any events not meeting filter criteria.
2. The `where` clause excludes events that do not meet its search condition.
3. Aggregate functions in the select list calculate summary values for each group.
4. The `having` clause excludes events from the final results that do not meet its search condition.

The following query illustrates the use of `filter`, `where`, `group by` and `having` clauses in one statement with a `select` clause containing an aggregate function.

```
select tickDataFeed, stddev(price)
from StockTickEvent(symbol='IBM').win:length(10)
where volume > 1000
group by tickDataFeed
having stddev(price) > 0.8
```

Esper filters events using the filter criteria for the event stream `StockTickEvent`. In the example above only events with symbol `IBM` enter the length window over the last 10 events, all other events are simply discarded.

The `where` clause removes any events posted by the length window (events entering the window and event leaving the window) that do not match the condition of volume greater then 1000. Remaining events are applied to the `stddev` standard deviation aggregate function for each tick data feed as specified in the `group by` clause. Each `tickDataFeed` value generates one event. Esper applies the `having` clause and only lets events pass for `tickDataFeed` groups with a standard deviation of price greater then 0.8.

4.6.5. Comparing the *Group By* clause and the *std:groupwin* view

The *group by* clause as well as the built-in *std:groupwin* view are similar in their ability to group events. This section explains the key differences in their behavior and use.

The *group by* clause works together with aggregation functions in your statement to produce an aggregation result per group. In greater detail, this means that when a new event arrives, the engine applies the expressions in the *group by* clause to determine a grouping key. If the engine has not encountered that grouping key before (a new group), the engine creates a set of new aggregation results for that grouping key and performs the aggregation changing that new set of aggregation results. If the grouping key points to an existing set of prior aggregation results (an existing group), the engine performs the aggregation changing the prior set of aggregation results for that group.

The *std:groupwin* view is a built-in view that also groups events. The view is described in greater detail in Section 11.2.2, “Grouped Data Window (*std:groupwin*)”. Its primary use is to create a separate data window per group, or more generally to create separate instances of all its sub-views for each grouping key encountered.

The next example shows two queries that produce equivalent results. The query using the *group by* clause is generally preferable as is easier to read. The second form introduces the *stat:uni* view which computes univariate statistics for a given property:

```
select symbol, avg(price) from StockTickEvent group by symbol
// ... is equivalent to ...
select symbol, average from StockTickEvent.std:groupwin(symbol).stat:uni(price)
```

The next example shows two queries that are NOT equivalent as the length window is ungrouped in the first query, and grouped in the second query:

```
select symbol, sum(price) from StockTickEvent.win:length(10) group by symbol
// ... NOT equivalent to ...
select symbol, sum(price) from StockTickEvent.std:groupwin(symbol).win:length(10)
```

The key difference between the two statements is that in the first statement the length window is ungrouped and applies to all events regardless of group. While in the second query each group gets its own instance of a length window. For example, in the second query events arriving for symbol "ABC" get a length window of 10 events, and events arriving for symbol "DEF" get their own length window of 10 events.

4.7. Stabilizing and Controlling Output: the *Output* Clause

4.7.1. Output Clause Options

The *output* clause is optional in Esper and is used to control or stabilize the rate at which events are output and to suppress output events. The EPL language provides for several different ways to control output rate.

Here is the syntax for the *output* clause that specifies a rate in time interval or number of events:

```
output [after suppression_def]
```

```
[[all | first | last | snapshot] every output_rate [seconds | events]]
```

An alternate syntax specifies the time period between output as outlined in Section 4.2.1, “Specifying Time Periods”:

```
output [after suppression_def]
[[all | first | last | snapshot] every time_period]
```

A crontab-like schedule can also be specified. The schedule parameters follow the pattern observer parameters and are further described in Section 5.6.2.2, “timer:at”:

```
output [after suppression_def]
[[all | first | last | snapshot] at
(minutes, hours, days of month, months, days of week [, seconds])]
```

Last, output can be controlled by an expression that may contain variables, user-defined functions and information about the number of collected events. Output that is controlled by an expression is discussed in detail below.

The `after` keyword and `suppression_def` can appear alone or together with further output conditions and suppresses output events.

For example, the following statement outputs, every 60 seconds, the total price for all orders in the 30-minute time window:

```
select sum(price) from OrderEvent.win:time(30 min) output snapshot every 60 seconds
```

The `all` keyword is the default and specifies that all events in a batch should be output, each incoming row in the batch producing an output row. Note that for statements that group via the `group by` clause, the `all` keyword provides special behavior as below.

The `first` keyword specifies that only the first event in an output batch is to be output. Using the `first` keyword instructs the engine to output the first matching event as soon as it arrives, and then ignores matching events for the time interval or number of events specified. After the time interval elapsed, or the number of matching events has been reached, the next first matching event is output again and the following interval the engine again ignores matching events. For statements that group via the `group by` clause, the `first` keywords provides special behavior as below.

The `last` keyword specifies to only output the last event at the end of the given time interval or after the given number of matching events have been accumulated. Again, for statements that group via the `group by` clause the `last` keyword provides special behavior as below.

The `snapshot` keyword indicates that the engine output current computation results considering all events as per views specified and/or current aggregation results. While the other keywords control how a batch of events between output intervals is being considered, the `snapshot` keyword outputs all current state of a statement independent of the last batch. Its output is equivalent to the `GetEnumerator` method provided by a statement. The `snapshot` keyword requires a data window declaration if not used within a join and outputs only the last event if used without a data window.

The `output_rate` is the frequency at which the engine outputs events. It can be specified in terms of time or number of events. The value can be a number to denote a fixed output rate, or the name of a variable whose value is the output rate. By means of a variable the output rate can be controlled externally and changed dynamically at runtime.

Please consult the Appendix A, *Output Reference and Samples* for detailed information on insert and remove

stream output for the various `output` clause keywords.

The time interval can also be specified in terms of minutes; the following statement is identical to the first one.

```
select * from StockTickEvent.win:length(5) output every 1.5 minutes
```

A second way that output can be stabilized is by batching events until a certain number of events have been collected. The next statement only outputs when either 5 (or more) new or 5 (or more) old events have been batched.

```
select * from StockTickEvent.win:time(30 sec) output every 5 events
```

Additionally, event output can be further modified by the optional `last` keyword, which causes output of only the last event to arrive into an output batch.

```
select * from StockTickEvent.win:time(30 sec) output last every 5 events
```

Using the `first` keyword you can be notified at the start of the interval. This allows to watch for situations such as a rate falling below a threshold and only be informed every now and again after the specified output interval, but be informed the moment it first happens.

```
select * from TickRate.win:time(30 seconds) where rate<100 output first every 60 seconds
```

A sample statement using the Unix "crontab"-command schedule is shown next. See Section 5.6.2.2, "timer:at" for details on schedule syntax. Here, output occurs every 15 minutes from 8am to 5:45pm (hours 8 to 17 at 0, 15, 30 and 45 minutes past the hour):

```
select symbol, sum(price) from StockTickEvent group by symbol output at (* /15, 8:17, *, *, *)
```

Controlling Output Using an Expression

Output can also be controlled by an expression that may check variable values, use user-defined functions and query built-in properties that provide additional information. The synopsis is as follows:

```
output [after suppression_def]
[[all | first | last | snapshot] when trigger_expression
 [then set variable_name = assign_expression [, variable_name = assign_expression [...]]]
```

The `when` keyword must be followed by a trigger expression returning a boolean value of true or false, indicating whether to output. Use the optional `then` keyword to change variable values after the trigger expression evaluates to true. An assignment expression assigns a new value to variable(s).

Lets consider an example. The next statement assumes that your application has defined a variable by name `OutputTriggerVar` of boolean type. The statement outputs rows only when the `OutputTriggerVar` variable has a boolean value of true:

```
select sum(price) from StockTickEvent output when OutputTriggerVar = true
```

The engine evaluates the trigger expression when streams and data views post one or more insert or remove stream events after considering the `where` clause, if present. It also evaluates the trigger expression when any of the variables used in the trigger expression, if any, changes value. Thus output occurs as follows:

1. When there are insert or remove stream events and the `when` trigger expression evaluates to true, the engine outputs the resulting rows.
2. When any of the variables in the `when` trigger expression changes value, the engine evaluates the expres-

sion and outputs results. Result output occurs within the minimum time interval of timer resolution (100 milliseconds).

By adding a `then` part to the EPL, we can reset any variables after the trigger expression evaluated to true:

```
select sum(price) from StockTickEvent
  output when OutputTriggerVar = true
  then set OutputTriggerVar = false
```

Expressions in the `when` and `then` may, for example, use variables, user defined functions or any of the built-in named properties that are described in the below list.

The following built-in properties are available for use:

Table 4.3. Built-In Properties for Use with Output When

Built-In Property Name	Description
<code>last_output_timestamp</code>	Timestamp when the last output occurred for the statement; Initially set to time of statement creation
<code>count_insert</code>	Number of insert stream events
<code>count_remove</code>	Number of remove stream events

The values provided by `count_insert` and `count_remove` are non-continues: The number returned for these properties may 'jump' up rather than count up by 1. The counts reset to zero upon output.

The following restrictions apply to expressions used in the output rate clause:

- Event property names cannot be used in the output clause.
- Aggregation functions cannot be used in the output clause.
- The `prev` previous event function and the `prior` prior event function cannot be used in the output clause.

Suppressing Output With `After`

The `after` keyword and its time period or number of events parameters is optional and can occur after the `output` keyword, either alone or with output conditions as listed above.

The synopsis of `after` is as follows:

```
output after time_period | number events [...]
```

When using `after` either alone or together with further output conditions, the engine discards all output events until the time period passed as measured from the start of the statement, or until the number of output events are reached. The discarded events are not output and do not count towards any further output conditions if any are specified.

For example, the following statement outputs every minute the total price for all orders in the 30-minute time window but only after 30 minutes have passed:

```
select sum(price) from OrderEvent.win:time(30 min) output after 30 min snapshot every 1 min
```

An example in which `after` occur alone is below, in a statement that outputs total price for all orders in the last minute but only after 1 minute passed, each time an event arrives or leaves the data window:

```
select sum(price) from OrderEvent.win:time(1 min) output after 1 min
```

To demonstrate `after` when used with an event count, this statement find pairs of orders with the same id but suppresses output for the first 5 pairs:

```
select * from pattern[every o=OrderEvent->p=OrderEvent(id=o.id)] output after 5 events
```

4.7.2. Aggregation, Group By, Having and Output clause interaction

Remove stream events can also be useful in conjunction with aggregation and the `output` clause: When the engine posts remove stream events for fully-aggregated queries, it presents the aggregation state before the expiring event leaves the data window. Your application can thus easily obtain a delta between the new aggregation value and the prior aggregation value.

The engine evaluates the `having`-clause at the granularity of the data posted by views. That is, if you utilize a time window and output every 10 events, the `having` clause applies to each individual event or events entering and leaving the time window (and not once per batch of 10 events).

The `output` clause interacts in two ways with the `group by` and `having` clauses. First, in the `output every n events` case, the number `n` refers to the number of events arriving into the `group by` clause. That is, if the `group by` clause outputs only 1 event per group, or if the arriving events don't satisfy the `having` clause, then the actual number of events output by the statement could be fewer than `n`.

Second, the `last`, `all` and `first` keywords have special meanings when used in a statement with aggregate functions and the `group by` clause:

- When no keyword is specified, the engine produces an output row for each row in the batch or when using `group-by` then an output per group only for those groups present in the batch, following Section 3.7.2, “Output for Aggregation and Group-By”.
- The `all` keyword (the default) specifies that the most recent data for *all* groups seen so far should be output, whether or not these groups' aggregate values have just been updated
- The `last` keyword specifies that only groups whose aggregate values have been updated with the most recent batch of events should be output.
- The `first` keyword specifies that only groups whose aggregate values have been updated with the most recent batch of events should be output following the defined frequency, keeping frequency state for each group.

Please consult the Appendix A, *Output Reference and Samples* for detailed information on insert and remove stream output for aggregation and group-by.

By adding an output rate limiting clause to a statement that contains a `group by` clause we can control output of groups to obtain one row for each group, generating an event per group at the given output frequency.

The next statement outputs total price per symbol cumulatively (no data window was used here). As it specifies the `all` keyword, the statement outputs the current value for all groups seen so far, regardless of whether the group was updated in the last interval. Output occurs after an interval of 5 seconds passed and at the end of each subsequent interval:

```
select symbol, sum(price) from StockTickEvent group by symbol output all every 5 seconds
```

The below statement outputs total price per symbol considering events in the last 3 minutes. When events leave the 3-minute data window output also occurs as new aggregation values are computed. The `last` keyword instructs the engine to output only those groups that had changes. Output occurs after an interval of 10 seconds

passed and at the end of each subsequent interval:

```
select symbol, sum(price) from StockTickEvent.win:time(3 min)
group by symbol output last every 10 seconds
```

This statement also outputs total price per symbol considering events in the last 3 minutes. The `first` keyword instructs the engine to output as soon as there is a new value for a group. After output for a given group the engine suppresses output for the same group for 10 seconds and does not suppress output for other groups. Output occurs again for that group after the interval when the group has new value(s):

```
select symbol, sum(price) from StockTickEvent.win:time(3 min)
group by symbol output first every 10 seconds
```

4.7.3. Runtime Considerations

Output rate limiting provides output events to your application in regular intervals. Between intervals, the engine uses a buffer to hold events until the output condition is reached. If your application has high-volume streams, you may need to be mindful of the memory needs for output rates.

The `output` clause with the `snapshot` keyword does not require a buffer, all other output keywords do consume memory until the output condition is reached.

4.8. Sorting Output: the *Order By* Clause

The `order by` clause is optional. It is used for ordering output events by their properties, or by expressions involving those properties. .

For example, the following statement outputs batches of 5 or more stock tick events that are sorted first by price ascending and then by volume ascending:

```
select symbol from StockTickEvent.win:time(60 sec)
output every 5 events
order by price, volume
```

Here is the syntax for the `order by` clause:

```
order by expression [asc | desc] [, expression [asc | desc]] [, ...]
```

If the `order by` clause is absent then the engine still makes certain guarantees about the ordering of output:

- If the statement is not a join, does not group via `group by` clause and does not declare grouped data windows via `std:groupwin` view, the order in which events are delivered to listeners and through the `iterator pull` API is the order of event arrival.
- If the statement is a join or outer join, or groups, then the order in which events are delivered to listeners and through the `iterator pull` API is not well-defined. Use the `order by` clause if your application requires events to be delivered in a well-defined order.

Esper places the following restrictions on the expressions in the `order by` clause:

1. All aggregate functions that appear in the `order by` clause must also appear in the `select` expression.

Otherwise, any kind of expression that can appear in the `select` clause, as well as any name defined in the `select` clause, is also valid in the `order by` clause.

By default all sort operations on string values are performed via the `Compare` method and are thus not locale dependent. To account for differences in language or locale, see Section 13.4.17, “Engine Settings related to Language and Locale” to change this setting.

4.9. Limiting Row Count: the *Limit* Clause

The `limit` clause is typically used together with the `order by` and `output` clause to limit your query results to those that fall within a specified range. You can use it to receive the first given number of result rows, or to receive a range of result rows.

There are two syntaxes for the `limit` clause, each can be parameterized by integer constants or by variable names. The first syntax is shown below:

```
limit row_count [offset offset_count]
```

The required `row_count` parameter specifies the number of rows to output. The `row_count` can be an integer constant and can also be the name of the integer-type variable to evaluate at runtime.

The optional `offset_count` parameter specifies the number of rows that should be skipped (offset) at the beginning of the result set. A variable can also be used for this parameter.

The next sample EPL query outputs the top 10 counts per property 'uri' every 1 minute.

```
select uri, count(*) from WebEvent
group by uri
output snapshot every 1 minute
order by count(*) desc
limit 10
```

The next statement demonstrates the use of the `offset` keyword. It outputs ranks 3 to 10 per property 'uri' every 1 minute:

```
select uri, count(*) from WebEvent
group by uri
output snapshot every 1 minute
order by count(*) desc
limit 8 offset 2
```

The second syntax for the `limit` clause is for SQL standard compatibility and specifies the offset first, followed by the row count:

```
limit offset_count[, row_count]
```

The following are equivalent:

```
limit 8 offset 2
// ...equivalent to
limit 2, 8
```

A negative value for `row_count` returns an unlimited number of rows, and a zero value returns no rows. If variables are used, then the current variable value at the time of output dictates the row count and offset. A variable returning a null value for `row_count` also returns an unlimited number of rows.

A negative value for offset is not allowed. If your variable returns a negative or null value for offset then the value is assumed to be zero (i.e. no offset).

The `iterator pull` API also honors the `limit` clause, if present.

4.10. Merging Streams and Continuous Insertion: the *Insert Into* Clause

The `insert into` clause is optional in Esper. The clause can be specified to make the results of a statement available as an event stream for use in further statements, or to insert events into a named window. The clause can also be used to merge multiple event streams to form a single stream of events.

The syntax for the `insert into` clause is as follows:

```
insert [istream | rstream] into event_stream_name [ (property_name [, property_name] ) ]
```

The `istream` (default) and `rstream` keywords are optional. If no keyword or the `istream` keyword is specified, the engine supplies the insert stream events generated by the statement. The insert stream consists of the events entering the respective window(s) or stream(s). If the `rstream` keyword is specified, the engine supplies the remove stream events generated by the statement. The remove stream consists of the events leaving the respective window(s).

The `event_stream_name` is an identifier that names the event stream (and also implicitly names the types of events in the stream) generated by the engine. The identifier can be used in further statements to filter and process events of that event stream. The `insert into` clause can consist of just an event stream name, or an event stream name and one or more property names.

The engine also allows listeners to be attached to a statement that contain an `insert into` clause. Listeners receive all events posted to the event stream.

To merge event streams, simply use the same `event_stream_name` identifier in all EPL statements that merge their result event streams. Make sure to use the same number and names of event properties and event property types match up.

Esper places the following restrictions on the `insert into` clause:

1. The number of elements in the `select` clause must match the number of elements in the `insert into` clause if the clause specifies a list of event property names
2. If the event stream name has already been defined by a prior statement or configuration, and the event property names and/or event types do not match, an exception is thrown at statement creation time.

The following sample inserts into an event stream by name `CombinedEvent`:

```
insert into CombinedEvent
select A.customerId as custId, A.timestamp - B.timestamp as latency
  from EventA.win:time(30 min) A, EventB.win:time(30 min) B
 where A.txnId = B.txnId
```

Each event in the `CombinedEvent` event stream has two event properties named "custId" and "latency". The events generated by the above statement can be used in further statements, such as shown in the next statement:

```
select custId, sum(latency)
  from CombinedEvent.win:time(30 min)
 group by custId
```

The example statement below shows the alternative form of the `insert into` clause that explicitly defines the property names to use.

```
insert into CombinedEvent (custId, latency)
select A.customerId, A.timestamp - B.timestamp
...
```

The `rstream` keyword can be useful to indicate to the engine to generate only remove stream events. This can be useful if we want to trigger actions when events leave a window rather than when events enter a window. The statement below generates `CombinedEvent` events when `EventA` and `EventB` leave the window after 30 minutes.

```
insert rstream into CombinedEvent
select A.customerId as custId, A.timestamp - B.timestamp as latency
  from EventA.win:time(30 min) A, EventB.win:time(30 min) B
 where A.txnId = B.txnId
```

The `insert into` clause can be used in connection with patterns to provide pattern results to further statements for analysis:

```
insert into ReUpEvent
select linkUp.ip as ip
from pattern [every linkDown=LinkDownEvent -> linkUp=LinkUpEvent(ip=linkDown.ip)]
```

4.10.1. Transposing a Property To a Stream

Sometimes your events may carry properties that are themselves event objects. Therefore EPL offers a special syntax to insert the value of a property itself as an event into a stream:

```
insert into stream_name select property_name.* from ...
```

This feature is only supported for native events and is not supported for `Map` or `XML` events. Nested property names are also not supported.

In this example, the class `Summary` with properties `bid` and `ask` that are of type `Quote` is:

```
public class Summary {
  private Quote bid;
  private Quote ask;
  ...
}
```

The statement to populate a stream of `Quote` events is thus:

```
insert into MyBidStream select bid.* from Summary
```

4.10.2. Merging Streams By Event Type

The `insert into` clause allows to merge multiple event streams into a event single stream. The clause names an event stream to insert into by specifying an *event_stream_name*. The first statement that inserts into the named stream defines the stream's event types. Further statements that insert into the same event stream must match the type of events inserted into the stream as declared by the first statement.

One approach to merging event streams specifies individual column names either in the `select` clause or in the `insert into` clause of the statement. This approach has been shown in earlier examples.

Another approach to merging event streams specifies the wildcard (*) in the `select` clause (or the stream wildcard) to select the underlying event. The events in the event stream must then have the same event type as gen-

erated by the `from` clause.

Assume a statement creates an event stream named `MergedStream` by selecting `OrderEvent` events:

```
insert into MergedStream select * from OrderEvent
```

A statement can use the stream wildcard selector to select only `OrderEvent` events in a join:

```
insert into MergedStream select ord.* from ItemScanEvent, OrderEvent as ord
```

And a statement may also use an application-supplied user-defined function to convert events to `OrderEvent` instances:

```
insert into MergedStream select MyLib.convert(item) from ItemScanEvent as item
```

Esper specifically recognizes a conversion function: A conversion function must be the only selected column, and it must return either a CLR object or `System.Collections.Generic.IDictionary`.

4.10.3. Merging Disparate Types of Events: Variant Streams

A *variant stream* is a predefined stream into which events of multiple disparate event types can be inserted.

A variant stream name may appear anywhere in a pattern or `from` clause. In a pattern, a filter against a variant stream matches any events of any of the event types inserted into the variant stream. In a `from` clause including for named windows, views declared onto a variant stream may hold events of any of the event types inserted into the variant stream.

A variant stream is thus useful in problems that require different types of event to be treated the same.

Variant streams can be declared by means of `create variant schema` or can be predefined via runtime or initialization-time configuration as described in Section 13.4.23, “Variant Stream”. Your application may declare or predefine variant streams to carry events of a limited set of event types, or you may choose the variant stream to carry any and all types of events. This choice affects what event properties are available for consuming statements or patterns of the variant stream.

Assume that an application predefined a variant stream named `OrderStream` to carry only `ServiceOrder` and `ProductOrder` events. An `insert into` clause inserts events into the variant stream:

```
insert into OrderStream select * from ServiceOrder
```

```
insert into OrderStream select * from ProductOrder
```

Here is a sample statement that consumes the variant stream and outputs a total price per customer id for the last 30 seconds of `ServiceOrder` and `ProductOrder` events:

```
select customerId, sum(price) from OrderStream.win:time(30 sec) group by customerId
```

If your application predefines the variant stream to hold specific type of events, as the sample above did, then all event properties that are common to all specified types are visible on the variant stream, including nested, indexed and mapped properties. For access to properties that are only available on one of the types, the dynamic property syntax must be used. In the example above, the `customerId` and `price` were properties common to both `ServiceOrder` and `ProductOrder` events.

For example, here is a consuming statement that selects a `service duration` property that only `ServiceOrder`

events have, and that must therefore be casted to double and null values removed in order to aggregate:

```
select customerId, sum(coalesce(cast(serviceDuration?, double), 0))
from OrderStream.win:time(30 sec) group by customerId
```

If your application predefines a variant stream to hold any type of events (the `any` type variance), then all event properties of the variant stream are effectively dynamic properties.

For example, an application may define an `OutgoingEvents` variant stream to hold any type of event. The next statement is a sample consumer of the `OutgoingEvents` variant stream that looks for the `destination` property and fires for each event in which the property exists with a value of `'email'`:

```
select * from OutgoingEvents(destination = 'email')
```

4.10.4. Decorated Events

Your `select` clause may use the `'*'` wildcard together with further expressions to populate a stream of events. A sample statement is:

```
insert into OrderStream select *, price*units as linePrice from PurchaseOrder
```

When using wildcard and selecting additional expression results, the engine produces what is called *decorating* events for the resulting stream. Decorating events add additional property values to an underlying event.

In the above example the resulting `OrderStream` consists of underlying `PurchaseOrder` events *decorated* by a `linePrice` property that is a result of the `price*units` expression.

In order to use `insert into` to insert into an existing stream of decorated events, your underlying event type must match, and all additional decorating property names and types of the `select` clause must also match.

4.10.5. Event as a Property

Your `select` clause may use the stream name to populate a stream of events in which each event has properties that are itself an event. A sample statement is:

```
insert into CompositeStream select order, service, order.price+service.price as totalPrice
from PurchaseOrder.std:lastevent() as order, ServiceEvent.std:lastevent() as service
```

When using the stream name (or tag in patterns) in the `select` clause, the engine produces composite events: One or more of the properties of the composite event are events themselves.

In the above example the resulting `CompositeStream` consists of 3 columns: the `PurchaseOrder` event, the `ServiceEvent` event and the `totalPrice` property that is a result of the `order.price+service.price` expression.

In order to use `insert into` to insert into an existing stream of events in which properties are themselves events, each event column's event type must match, and all additional property names and types of the `select` clause must also match.

4.10.6. Populating an Underlying Event Object

Your `insert into` clause may also directly instantiate and populate application underlying event objects or `Map` event objects. This is described in greater detail in Section 2.11, “Event Objects Populated by Insert Into”.

4.11. Subqueries

A subquery is a `select` within another statement. Esper supports subqueries in the `select` clause, in the `where` clause and in stream and pattern filter expressions. Subqueries provide an alternative way to perform operations that would otherwise require complex joins. Subqueries can also make statements more readable than complex joins.

Esper supports both simple subqueries as well as correlated subqueries. In a simple subquery, the inner query is not correlated to the outer query. Here is an example simple subquery within a `select` clause:

```
select assetId, (select zone from ZoneClosed.std:lastevent()) as lastClosed from RFIDEvent
```

If the inner query is dependent on the outer query, we will have a correlated subquery. An example of a correlated subquery is shown below. Notice the `where` clause in the inner query, where the condition involves a stream from the outer query:

```
select * from RfidEvent as RFID where 'Dock 1' =  
  (select name from Zones.std:unique(zoneId) where zoneId = RFID.zoneId)
```

The example above shows a subquery in the `where` clause. The statement selects RFID events in which the zone name matches a string constant based on zone id. The statement uses the view `std:unique` to guarantee that only the last event per zone id is held from processing by the subquery.

The next example is a correlated subquery within a `select` clause. In this statement the `select` clause retrieves the zone name by means of a subquery against the Zones set of events correlated by zone id:

```
select zoneId, (select name from Zones.std:unique(zoneId)  
  where zoneId = RFID.zoneId) as name from RFIDEvent
```

Note that when a simple or correlated subquery returns multiple rows, the engine returns a `null` value as the subquery result. To limit the number of events returned by a subquery consider using one of the views `std:lastevent`, `std:unique` and `std:groupwin` or aggregation functions or the multi-row and multi-column selects as described below.

The `select` clause of a subquery also allows wildcard selects, which return as an event property the underlying event object of the event type as defined in the `from` clause. An example:

```
select (select * from MarketData.std:lastevent()) as md  
  from pattern [every timer:interval(10 sec)]
```

The output events to the statement above contain the underlying MarketData event in a property named "md". The statement populates the last MarketData event into a property named "md" every 10 seconds following the pattern definition, or populates a `null` value if no MarketData event has been encountered so far.

When your subquery returns multiple rows, you must use an aggregation function in the `select` clause of the subselect, as a subquery can only return a single row and single value object. To return multiple values from a subquery, consider writing a custom aggregation function that returns an array or collection of values.

Aggregation functions may be used in the `select` clause of the subselect as this example outlines:

```
select * from MarketData  
  where price > (select max(price) from MarketData(symbol='GOOG').std:lastevent())
```

As the sub-select expression is evaluated first (by default), the query above actually never fires for the GOOG

symbol, only for other symbols that have a price higher then the current maximum for GOOG. As a sidenote, the `insert into` clause can also be handy to compute aggregation results for use in multiple subqueries.

When using aggregation functions in a correlated subselect the engine computes the aggregation based on data window or named window contents matching the where-clause.

The following example compares the quantity value provided by the current order event against the total quantity of all order events in the last 1 hour for the same client.

```
select * from OrderEvent oe
where qty >
  (select sum(qty) from OrderEvent.win:time(1 hour) pd
   where pd.client = oe.client)
```

Filter expressions in a pattern or stream may also employ subqueries. Subqueries can be uncorrelated or can be correlated to properties of the stream or to properties of tagged events in a pattern. Subqueries may reference named windows as well.

The following example filters `BarData` events that have a close price less then the last moving average (field `movAgv`) as provided by stream `SMA20Stream` (an uncorrelated subquery):

```
select * from BarData(ticker='MSFT', closePrice <
  (select movAgv from SMA20Stream(ticker='MSFT').std:lastevent()))
```

A few generic examples follow to demonstrate the point. The examples use short event and property names so they are easy to read. Assume `A` and `B` are streams and `DNamedWindow` is a named window, and properties `a_id`, `b_id`, `d_id`, `a_val`, `b_val`, `d_val` respectively:

```
// Sample correlated subquery as part of stream filter criteria
select * from A(a_val in
  (select b_val from B.std:unique(b_val) as b where a.a_id = b.b_id)) as a
```

```
// Sample correlated subquery against a named window
select * from A(a_val in
  (select b_val from DNamedWindow as d where a.a_id = d.d_id)) as a
```

```
// Sample correlated subquery in the filter criteria as part of a pattern, querying a named window
select * from pattern [
  a=A -> b=B(bvalue =
    (select d_val from DNamedWindow as d where d.d_id = b.b_id and d.d_id = a.a_id))
]
```

Subquery state starts to accumulate as soon as a statement starts (and not only when a pattern-subexpression activates).

The following restrictions apply to subqueries:

1. The subquery stream definition must define a data window or other view to limit subquery results, reducing the number of events held for subquery execution
2. Subqueries can only consist of a `select` clause, a `from` clause and a `where` clause. The `group by` and `having` clauses, as well as joins, outer-joins and output rate limiting are not permitted within subqueries.
3. If using aggregation functions in a subquery, note these limitations:
 - a. None of the properties of the correlated stream(s) can be used within aggregation functions.
 - b. The properties of the subselect stream must all be within aggregation functions.

The order of evaluation of subqueries relative to the containing statement is guaranteed: If the containing state-

ment and its subqueries are reacting to the same type of event, the subquery will receive the event first before the containing statement's clauses are evaluated. This behavior can be changed via configuration. The order of evaluation of subqueries is not guaranteed between subqueries.

Performance of your statement containing one or more subqueries principally depends on two parameters. First, if your subquery correlates one or more columns in the subquery stream with the enclosing statement's streams, the engine automatically builds the appropriate indexes for fast row retrieval based on the key values correlated (joined). The second parameter is the number of rows found in the subquery stream and the complexity of the filter criteria (*where* clause), as each row in the subquery stream must evaluate against the *where* clause filter.

4.11.1. The 'exists' Keyword

The *exists* condition is considered "to be met" if the subquery returns at least one row. The *not exists* condition is considered true if the subquery returns no rows.

The synopsis for the *exists* keyword is as follows:

```
exists (subquery)
```

Let's take a look at a simple example. The following is an EPL statement that uses the *exists* condition:

```
select assetId from RFIDEvent as RFID
  where exists (select * from Asset.std:unique(assetId) where assetId = RFID.assetId)
```

This select statement will return all RFID events where there is at least one event in Assets unique by asset id with the same asset id.

4.11.2. The 'in' and 'not in' Keywords

The *in* subquery condition is true if the value of an expression matches one or more of the values returned by the subquery. Consequently, the *not in* condition is true if the value of an expression matches none of the values returned by the subquery.

The synopsis for the *in* keyword is as follows:

```
expression in (subquery)
```

The right-hand side subquery must return exactly one column.

The next statement demonstrates the use of the *in* subquery condition:

```
select assetId from RFIDEvent
  where zone in (select zone from ZoneUpdate(status = 'closed').win:time(10 min))
```

The above statement demonstrated the *in* subquery to select RFID events for which the zone status is in a closed state.

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand row yields null, the result of the *in* construct will be null, not false (or true for *not-in*). This is in accordance with SQL's normal rules for Boolean combinations of null values.

4.11.3. The 'any' and 'some' Keywords

The `any` subquery condition is true if the expression returns true for one or more of the values returned by the subquery.

The synopsis for the `any` keyword is as follows:

```
expression operator any (subquery)
expression operator some (subquery)
```

The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of `any` is "true" if any true result is obtained. The result is "false" if no true result is found (including the special case where the subquery returns no rows).

The *operator* can be any of the following values: `=`, `!=`, `<>`, `<`, `<=`, `>`, `>=`.

The `some` keyword is a synonym for `any`. The `in` construct is equivalent to `= any`.

The right-hand side subquery must return exactly one column.

The next statement demonstrates the use of the `any` subquery condition:

```
select * from ProductOrder as ord
  where quantity < any
    (select minimumQuantity from MinimumQuantity.win:keepall())
```

The above query compares `ProductOrder` event's quantity value with all rows from the `MinimumQuantity` stream of events and returns only those `ProductOrder` events that have a quantity that is less than any of the minimum quantity values of the `MinimumQuantity` events.

Note that if there are no successes and at least one right-hand row yields null for the operator's result, the result of the `any` construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

4.11.4. The 'all' Keyword

The `all` subquery condition is true if the expression returns true for all of the values returned by the subquery.

The synopsis for the `all` keyword is as follows:

```
expression operator all (subquery)
```

The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of `all` is "true" if all rows yield true (including the special case where the subquery returns no rows). The result is "false" if any false result is found. The result is `null` if the comparison does not return false for any row, and it returns `null` for at least one row.

The *operator* can be any of the following values: `=`, `!=`, `<>`, `<`, `<=`, `>`, `>=`.

The `not in` construct is equivalent to `!= all`.

The right-hand side subquery must return exactly one column.

The next statement demonstrates the use of the `all` subquery condition:

```
select * from ProductOrder as ord
  where quantity < all
    (select minimumQuantity from MinimumQuantity.win:keepall())
```

The above query compares `ProductOrder` event's quantity value with all rows from the `MinimumQuantity` stream of events and returns only those `ProductOrder` events that have a quantity that is less than all of the minimum quantity values of the `MinimumQuantity` events.

4.11.5. Multi-Column Selection

Your subquery may select multiple columns in the `select` clause including multiple aggregated values from a data window or named window.

The following example is a correlated subquery that selects wildcard and in addition selects the `bid` and `offer` properties of the last `MarketData` event for the same symbol as the arriving `OrderEvent`:

```
select *,
  (select bid, offer from MarketData.std:unique(symbol) as md
   where md.symbol = oe.symbol) as bidoffer
from OrderEvent oe
```

Output events for the above query contain all properties of the original `OrderEvent` event. In addition each output event contains a `bidoffer` nested property that itself contains the `bid` and `offer` properties. You may retrieve the `bid` and `offer` from output events directly via the `bidoffer.bid` property name syntax for nested properties.

The next example is similar to the above query but instead selects aggregations and selects from a named window by name `OrderNamedWindow` (creation not shown here). For each arriving `OrderEvent` it selects the total quantity and count of all order events for the same client, as currently held by the named window:

```
select *,
  (select sum(qty) as sumPrice, count(*) as countRows
   from OrderNamedWindow as onw
   where onw.client = oe.client) as pastOrderTotals
from OrderEvent as oe
```

Output events for the above query contain all properties of the original `OrderEvent` event. In addition each output event contains a `pastOrderTotals` nested property that itself contains the `sumPrice` and `countRows` properties.

4.11.6. Multi-Row Selection

While a subquery cannot change the cardinality of the selected stream, a subquery can return multiple values from the selected data window or named window. This section shows examples of the `window` aggregation function in subselects.

Consider using an inner join, outer join or unidirectional join instead to achieve a 1-to-many cardinality in the number of output events.

The next example is an uncorrelated subquery that selects all current `ZoneEvent` events considering the last `ZoneEvent` per zone for each arriving `RFIDEvent`.

```
select assetId,
  (select window(z.*) as winzones from ZoneEvent.std:unique(zone) as z) as zones
from RFIDEvent
```

Output events for the above query contain two properties: the `assetId` property and the `zones` property. The latter property is a nested property that contains the `winzones` property. You may retrieve the zones from output events directly via the `zones.winzones` property name syntax for nested properties.

In this example for a correlated subquery against a named window we assume that the `OrderNamedWindow` has been created and contains order events. The query returns for each `MarketData` event the list of order ids for orders with the same symbol:

```
select price,
  (select window(orderId) as winorders
   from OrderNamedWindow onw
   where onw.symbol = md.symbol) as orderIds
from MarketData md
```

Output events for the above query contain two properties: the `price` property and the `orderIds` property. The latter property is a nested property that contains the `winorders` property of type array.

4.11.7. Hints Related to Subqueries

The following hints are available to tune performance and memory use of subqueries.

Use the `@Hint('set_noindex')` hint for a statement that utilizes one or more subqueries. It instructs the engine to always perform a full table scan. The engine does not build an implicit index or use an explicitly-created index when this hint is provided. Use of the hint may result in reduced memory use but poor statement performance.

The following hints are available to tune performance and memory use of subqueries that select from named windows.

Named windows are globally-visible data windows. As such an application may create explicit indexes as discussed in Section 4.15.10, “Explicitly Indexing Named Windows”. The engine may also elect to create implicit indexes (no `create-index` EPL required) for index-based lookup of rows when executing `on-select`, `on-merge`, `on-update` and `on-delete` statements and for statements that subquery a named window.

By default and without specifying a hint, each statement that subqueries a named window also maintains its own index for looking up events held by the named window. The engine maintains the index by consuming the named window insert and remove stream. When the statement is destroyed it releases that index.

Specify the `@Hint('enable_window_subquery_indexshare')` hint to enable subquery index sharing for named windows. When using this hint, indexes for subqueries are maintained by the named window itself (and not each statement), are shared between one or more statements and may also utilize explicit indexes. Specify the hint once as part of the `create window` statement.

This sample EPL statement creates a named window with subquery index sharing enabled:

```
@Hint('enable_window_subquery_indexshare')
create window AllOrdersNamedWindow.win:keepall() as OrderMapEventType
```

When subquery index sharing is enabled, performance may increase as named window stream consumption is no longer needed. You may also expect reduced memory use especially if a large number of EPL statements perform similar subqueries against a named window. Subquery index sharing may require additional short-lived object creation and may slightly increase lock held time for named windows.

The following statement performs a correlated subquery against the named window above. When a settlement event arrives it select the order detail for the same order id as provided by the settlement event:

```
select
  (select * from AllOrdersNamedWindow as onw
   where onw.orderId = se.orderId) as orderDetail
from SettlementEvent as se
```

With subquery index sharing enabled the engine maintains an index of order events by order id for the named window, and shares that index between additional statements until the time all utilizing statements are destroyed.

You may disable subquery index sharing for a specific statement by specifying the `@Hint('disable_window_subquery_indexshare')` hint, as this example shows, causing the statement to maintain its own index:

```
@Hint('disable_window_subquery_indexshare')
select
  (select * from AllOrdersNamedWindow as onw
   where onw.orderId = se.orderId) as orderDetail
from SettlementEvent as se
```

4.12. Joining Event Streams

4.12.1. Introducing Joins

Two or more event streams can be part of the `from`-clause and thus both (all) streams determine the resulting events. This section summarizes the important concepts. The sections that follow present more detail on each topic.

The default join is an inner join which produces output events only when there is at least one match in all streams.

Consider the sample statement shown next:

```
select * from TickEvent.std:lastevent(), NewsEvent.std:lastevent()
```

The above statement outputs the last `TickEvent` and the last `NewsEvent` in one output event when either a `TickEvent` or a `NewsEvent` arrives. If no `TickEvent` was received before a `NewsEvent` arrives, no output occurs. Similarly when no `NewsEvent` was received before a `TickEvent` arrives, no output occurs.

The `where`-clause lists the join conditions that Esper uses to relate events in the two or more streams.

The next example statement retains the last `TickEvent` and last `NewsEvent` per symbol, and joins the two streams based on their symbol value:

```
select * from TickEvent.std:unique(symbol) as t, NewsEvent.std:unique(symbol) as n
where t.symbol = n.symbol
```

As before, when a `TickEvent` arrives for a symbol that has no matching `NewsEvent` then there is no output event.

An outer join does not require each event in either stream to have a matching event. The full outer join is useful when output is desired when no match is found. The different outer join types (full, left, right) are explained in more detail below.

This example statement is an outer-join and also returns the last `TickEvent` and last `NewsEvent` per symbol:

```
select * from TickEvent.std:unique(symbol) as t
full outer join NewsEvent.std:unique(symbol) as n on t.symbol = n.symbol
```

In the sample statement above, when a `TickEvent` arrives for a symbol that has no matching `NewsEvent`, or

when a `NewsEvent` arrives for a symbol that has no matching `TickEvent`, the statement still produces an output event with a null column value for the missing event.

Note that each of the sample queries above defines a data window. The sample queries above use the last-event data window (`std:lastevent`) or the unique data window (`std:unique`). A data window serves to indicate the subset of events to join from each stream and may be required depending on the join.

In above queries, when either a `TickEvent` arrives or when a `NewsEvent` arrives then the query evaluates and there is output. The same holds true if additional streams are added to the `from`-clause: Each of the streams in the `from`-clause trigger the join to evaluate.

The `unidirectional` keyword instructs the engine to evaluate the join only when an event arrives from the single stream that was marked with the `unidirectional` keyword. In this case no data window should be specified for the stream marked as `unidirectional` since the keyword implies that the current event of that stream triggers the join.

Here is the sample statement above with `unidirectional` keyword, so that output occurs only when a `TickEvent` arrives and not when a `NewsEvent` arrives:

```
select * from TickEvent as t unidirectional, NewsEvent.std:unique(symbol) as n
where t.symbol = n.symbol
```

It is oftentimes the case that an aggregation (count, sum, average) only needs to be calculated in the context of an arriving event or timer. Consider using the `unidirectional` keyword when aggregating over joined streams.

An EPL pattern is a normal citizen also providing a stream of data consisting of pattern matches. A time pattern, for example, can be useful to evaluate a join and produce output upon each interval.

This sample statement includes a pattern that fires every 5 seconds and thus triggers the join to evaluate and produce output, computing an aggregated total quantity per symbol every 5 seconds:

```
select symbol, sum(qty) from pattern[every timer:interval(5 sec)] unidirectional,
    TickEvent.std:unique(symbol) t, NewsEvent.std:unique(symbol) as n
where t.symbol = n.symbol group by symbol
```

Named windows as well as reference and historical data such as stored in your relational database, and data returned by a method invocation, can also be included in joins as discussed in Section 4.13, “Accessing Relational Data via SQL” and Section 4.14, “Accessing Non-Relational Data via Method Invocation”.

Related to joins are subqueries: A subquery is a `select` within another statement, see Section 4.11, “Subqueries”

The engine performs extensive query analysis and planning, building internal indexes and strategies as required to allow fast evaluation of many types of queries.

4.12.2. Inner (Default) Joins

Each point in time that an event arrives to one of the event streams, the two event streams are joined and output events are produced according to the `where` clause when matching events are found for all joined streams.

This example joins 2 event streams. The first event stream consists of fraud warning events for which we keep the last 30 minutes. The second stream is withdrawal events for which we consider the last 30 seconds. The streams are joined on account number.

```
select fraud.accountNumber as acctNum, fraud.warning as warn, withdraw.amount as amount,
```

```
max(fraud.timestamp, withdraw.timestamp) as timestamp, 'withdrawlFraud' as desc
from com.espertech.esper.example.atm.FraudWarningEvent.win:time(30 min) as fraud,
com.espertech.esper.example.atm.WithdrawalEvent.win:time(30 sec) as withdraw
where fraud.accountNumber = withdraw.accountNumber
```

Joins can also include one or more pattern statements as the next example shows:

```
select * from FraudWarningEvent.win:time(30 min) as fraud,
pattern [every w=WithdrawalEvent -> PINChangeEvent(acct=w.acct)].std:lastevent() as withdraw
where fraud.accountNumber = withdraw.w.accountNumber
```

The statement above joins the last 30 minutes of fraud warnings with a pattern. The pattern consists of every withdrawal event that is followed by a PIN change event for the same account number. It joins the two event streams on account number. The last-event view instructs the join to only consider the last pattern match.

In a join and outer join, your statement must declare a data window view or other view onto each stream. Streams that are marked as unidirectional and named windows as well as database or methods in a join are an exception and do not require a view to be specified. If you are joining an event to itself via contained-event selection, views also do not need to be specified.

The next example joins all FraudWarningEvent events that arrived since the statement was started, with the last 20 seconds of PINChangeEvent events:

```
select * from FraudWarningEvent.win:keepall() as fraud, PINChangeEvent.win:time(20 sec) as pin
where fraud.accountNumber = pin.accountNumber
```

The above example employed the special keep-all view that retains all events.

4.12.3. Outer, Left and Right Joins

Esper supports left outer joins, right outer joins, full outer joins and inner joins in any combination between an unlimited number of event streams. Outer and inner joins can also join reference and historical data as explained in Section 4.13, “Accessing Relational Data via SQL”, as well as join data returned by a method invocation as outlined in Section 4.14, “Accessing Non-Relational Data via Method Invocation”.

The keywords *left*, *right*, *full* and *inner* control the type of the join between two streams. The *on* clause specifies one or more properties that join each stream. The synopsis is as follows:

```
...from stream_def [as name]
((left|right|full outer) | inner) join stream_def
on property = property [and property = property ...]
[ ((left|right|full outer) | inner) join stream_def on ...]...
```

If the outer join is a left outer join, there will be an output event for each event of the stream on the left-hand side of the clause. For example, in the left outer join shown below we will get output for each event in the stream RfidEvent, even if the event does not match any event in the event stream OrderList.

```
select * from RfidEvent.win:time(30 sec) as rfid
left outer join
OrderList.win:length(10000) as orderlist
on rfid.itemId = orderlist.itemId
```

Similarly, if the join is a Right Outer Join, then there will be an output event for each event of the stream on the right-hand side of the clause. For example, in the right outer join shown below we will get output for each event in the stream OrderList, even if the event does not match any event in the event stream RfidEvent.

```
select * from RfidEvent.win:time(30 sec) as rfid
      right outer join
      OrderList.win:length(10000) as orderlist
      on rfid.itemId = orderList.itemId
```

For all types of outer joins, if the join condition is not met, the select list is computed with the event properties of the arrived event while all other event properties are considered to be null.

The next type of outer join is a full outer join. In a full outer join, each point in time that an event arrives to one of the event streams, one or more output events are produced. In the example below, when either an RfidEvent or an OrderList event arrive, one or more output event is produced. The next example shows a full outer join that joins on multiple properties:

```
select * from RfidEvent.win:time(30 sec) as rfid
      full outer join
      OrderList.win:length(10000) as orderlist
      on rfid.itemId = orderList.itemId and rfid.assetId = orderList.assetId
```

The last type of join is an inner join. In an inner join, the engine produces an output event for each event of the stream on the left-hand side that matches at least one event on the right hand side considering the join properties. For example, in the inner join shown below we will get output for each event in the RfidEvent stream that matches one or more events in the OrderList data window:

```
select * from RfidEvent.win:time(30 sec) as rfid
      inner join
      OrderList.win:length(10000) as orderlist
      on rfid.itemId = orderList.itemId and rfid.assetId = orderList.assetId
```

Patterns as streams in a join follow this rule: If no data window view is declared for the pattern then the pattern stream retains the last match. Thus a pattern must have matched at least once for the last row to become available in a join. Multiple rows from a pattern stream may be retained by declaring a data window view onto a pattern using the pattern `[...].view_specification` syntax.

Finally, this example outer joins multiple streams. Here the RfidEvent stream is outer joined to both ProductName and LocationDescription via left outer join:

```
select * from RfidEvent.win:time(30 sec) as rfid
      left outer join ProductName.win:keepall() as refprod
      on rfid.productId = refprod.prodId
      left outer join LocationDescription.win:keepall() as refdesc
      on rfid.location = refdesc.locId
```

The `on` clause may only employ the `=` equals operator and property names. Any other operators must be placed in the `where`-clause.

4.12.4. Unidirectional Joins

In a join or outer join your statement lists multiple event streams, views and/or patterns in the `from` clause. As events arrive into the engine, each of the streams (views, patterns) provides insert and remove stream events. The engine evaluates each insert and remove stream event provided by each stream, and joins or outer joins each event against data window contents of each stream, and thus generates insert and remove stream join results.

The direction of the join execution depends on which stream or streams are currently providing an insert or remove stream event for executing the join. A join is thus multidirectional, or bidirectional when only two streams are joined. A join can be made unidirectional if your application does not want new results when events

arrive on a given stream or streams.

The `unidirectional` keyword can be used in the `from` clause to identify a single stream that provides the events to execute the join. If the keyword is present for a stream, all other streams in the `from` clause become passive streams. When events arrive or leave a data window of a passive stream then the join does not generate join results.

For example, consider a use case that requires us to join stock tick events (`TickEvent`) and news events (`NewsEvent`). The `unidirectional` keyword allows to generate results only when `TickEvent` events arrive, and not when `NewsEvent` arrive or leave the 10-second time window:

```
select * from TickEvent unidirectional, NewsEvent.win:time(10 sec)
where tick.symbol = news.symbol
```

Aggregation functions in a `unidirectional` join aggregate within the context of each `unidirectional` event evaluation and are not cumulative. Thereby aggregation functions when used with `unidirectional` may evaluate faster as they do not need to consider a remove stream (data removed from data windows or named windows).

The count function in the next query returns, for each `TickEvent`, the number of matching `NewsEvent` events:

```
select count(*) from TickEvent unidirectional, NewsEvent.win:time(10 sec)
where tick.symbol = news.symbol
```

The following restrictions apply to `unidirectional` joins:

1. The `unidirectional` keyword can only be specified for a single stream in the `from` clause.
2. Receiving data from a `unidirectional` join via the pull API (`GetEnumerator` method) is not allowed. This is because the engine holds no state for the single stream that provides the events to execute the join.
3. The stream that declares the `unidirectional` keyword cannot declare a data window view or other view for that stream, since remove stream events are not processed for the single stream.

4.12.5. Hints Related to Joins

When joining 3 or more streams (including any relational or non-relational sources as below) it can sometimes help to provide the query planner instructions how to best execute the join. The engine compiles a query plan for the EPL statement at statement creation time. You can output the query plan to logging (see configuration).

An outer join that specifies only `inner` keywords for all streams is equivalent to an default (inner) join. The following two statements are equivalent:

```
select * from TickEvent.std:lastevent(),
       NewsEvent.std:lastevent() where tick.symbol = news.symbol
```

Equivalent to:

```
select * from TickEvent.std:lastevent()
       inner join NewsEvent.std:lastevent() on tick.symbol = news.symbol
```

For all types of joins, the query planner determines a query graph: The term is used here for all the information regarding what properties or expressions are used to join the streams. The query graph thus includes the where-clause expressions as well as outer-join on-clauses if this statement is an outer join. The query planner also computes a dependency graph which includes information about all historical data streams (relational and non-relational as below) and their input needs.

For default (inner) joins the query planner first attempts to find a path of execution as a nested iteration. For each stream the query planner selects the best order of streams available for the nested iteration considering the query graph and dependency graph. If the full depth of the join is achievable via nested iteration for all streams without full table scan then the query planner uses that nested iteration plan. If not, then the query planner re-plans considering a merge join (Cartesian) approach instead.

Specify the `@Hint('PREFER_MERGE_JOIN')` to instruct the query planner to prefer a merge join plan instead of a nested iteration plan. Specify the `@Hint('FORCE_NESTED_ITER')` to instruct the query planner to always use a nested iteration plan.

For example, consider the below statement. Depending on the number of matching rows in `OrderBookOne` and `OrderBookTwo` (named windows in this example, and assumed to be defined elsewhere) the performance of the join may be better using the merge join plan.

```
@Hint('PREFER_MERGE_JOIN')
select * from TickEvent.std:lastevent() t,
        OrderBookOne ob1, OrderBookOne ob2
where ob1.symbol = t.symbol and ob2.symbol = t.symbol
and ob1.price between t.buy and t.sell and ob2.price between t.buy and t.sell
```

For outer joins the query planner considers nested iteration and merge join (Cartesian) equally and above hints don't apply.

4.13. Accessing Relational Data via SQL

This chapter outlines how reference data and historical data that are stored in a relational database can be queried via SQL within EPL statements.

Esper can access via join and outer join as well as via enumerator (poll) API all types of event streams to stored data. In order for such data sources to become accessible to Esper, some configuration is required. The Section 13.4.8, “Relational Database Access” explains the required configuration for database access in greater detail, and includes information on configuring a query result cache.

Esper does not parse or otherwise inspect your SQL query. Therefore your SQL can make use of any database-specific SQL language extensions or features that your database provides.

If you have enabled query result caching in your Esper database configuration, Esper retains SQL query results in cache following the configured cache eviction policy.

Also if you have enabled query result caching in your Esper database configuration and provide EPL `where` clause and/or `on` clause (outer join) expressions, then Esper builds indexes on the SQL query results to enable fast lookup. This is especially useful if your queries return a large number of rows. For building the proper indexes, Esper inspects the expression found in your EPL query `where` clause, if present. For outer joins, Esper also inspects your EPL query `on` clause. Esper analyzes the EPL `on` clause and `where` clause expressions, if present, looking for property comparison with or without logical AND-relationships between properties. When a SQL query returns rows for caching, Esper builds and caches the appropriate index and lookup strategies for fast row matching against indexes.

Joins or outer joins in which only SQL statements or method invocations are listed in the `from` clause and no other event streams are termed *passive* joins. A passive join does not produce an insert or remove stream and therefore does not invoke statement listeners with results. A passive join can be iterated on (polled) using a statement's `GetSafeEnumerator` and `GetEnumerator` methods.

There are no restrictions to the number of SQL statements or types of streams joined. The following restrictions

currently apply:

- Sub-views on an SQL query are not allowed; That is, one cannot create a time or length window on an SQL query. However one can use the `insert into` syntax to make join results available to a further statement.
- Your database software must support JDBC prepared statements that provide statement meta data at compilation time. Most major databases provide this function. A workaround is available for databases that do not provide this function.
- JDBC drivers must support the `getMetadata` feature. A workaround is available as below for JDBC drivers that don't support getting metadata.

The next sections assume basic knowledge of SQL (Structured Query Language).

4.13.1. Joining SQL Query Results

To join an event stream against stored data, specify the `sql` keyword followed by the name of the database and a parameterized SQL query. The syntax to use in the `from` clause of an EPL statement is:

```
sql:database_name [ " parameterized_sql_query " ]
```

The engine uses the *database_name* identifier to obtain configuration information in order to establish a database connection, as well as settings that control connection creation and removal. Please see Section 13.4.8, “Relational Database Access” to configure an engine for database access.

Following the database name is the SQL query to execute. The SQL query can contain one or more substitution parameters. The SQL query string is placed in single brackets [and]. The SQL query can be placed in either single quotes (') or double quotes ("). The SQL query grammar is passed to your database software unchanged, allowing you to write any SQL query syntax that your database understands, including stored procedure calls.

Substitution parameters in the SQL query string take the form `${expression}`. The engine resolves *expression* at statement execution time to the actual expression result by evaluating the events in the joined event stream or current variable values, if any event property references or variables occur in the expression. An *expression* may not contain EPL substitution parameters.

The engine determines the type of the SQL query output columns by means of the result set metadata that your database software returns for the statement. The actual query results are obtained via `System.Data.DbDataReader`.

The sample EPL statement below joins an event stream consisting of `CustomerCallEvent` events with the results of an SQL query against the database named `MyCustomerDB` and table `Customer`:

```
select custId, cust_name from CustomerCallEvent,
sql:MyCustomerDB [ ' select cust_name from Customer where cust_id = ${custId} ' ]
```

The example above assumes that `CustomerCallEvent` supplies an event property named `custId`. The SQL query selects the customer name from the `Customer` table. The `where` clause in the SQL matches the `Customer` table column `cust_id` with the value of `custId` in each `CustomerCallEvent` event. The engine executes the SQL query for each new `CustomerCallEvent` encountered.

If the SQL query returns no rows for a given customer id, the engine generates no output event. Else the engine generates one output event for each row returned by the SQL query. An outer join as described in the next section can be used to control whether the engine should generate output events even when the SQL query returns no rows.

The next example adds a time window of 30 seconds to the event stream `CustomerCallEvent`. It also renames

the selected properties to `customerName` and `customerId` to demonstrate how the naming of columns in an SQL query can be used in the `select` clause in the EPL query. And the example uses explicit stream names via the `as` keyword.

```
select customerId, customerName from
  CustomerCallEvent.win:time(30 sec) as cce,
  sql:MyCustomerDB ["select cust_id as customerId, cust_name as customerName from Customer
                    where cust_id = ${cce.custId}"] as cq
```

Any window, such as the time window, generates insert stream (`istream`) events as events enter the window, and remove stream (`rstream`) events as events leave the window. The engine executes the given SQL query for each `CustomerCallEvent` in both the insert stream and the remove stream. As a performance optimization, the `istream` or `rstream` keywords in the `select` clause can be used to instruct the engine to only join insert stream or remove stream events, reducing the number of SQL query executions.

Since any expression may be placed within the `${...}` syntax, you may use variables or user-defined functions as well.

The next example assumes that a variable by name `varLowerLimit` is defined and that a user-defined function `getLimit` exists on the `MyLib` imported class that takes a `LimitEvent` as a parameter:

```
select * from LimitEvent le,
  sql:MyCustomerDB [' select cust_name from Customer where
                    amount > ${max(varLowerLimit, MyLib.getLimit(le))} ']
```

The example above takes the higher of the current variable value or the value returned by the user-defined function to return only those customer names where the amount exceeds the computed limit.

4.13.2. SQL Query and the EPL `where` Clause

Consider using the EPL `where` clause to join the SQL query result to your event stream. Similar to EPL joins and outer-joins that join event streams or patterns, the EPL `where` clause provides join criteria between the SQL query results and the event stream (as a side note, an SQL `where` clause is a filter of rows executed by your database on your database server before returning SQL query results).

Esper analyzes the expression in the EPL `where` clause and outer-join `on` clause, if present, and builds the appropriate indexes from that information at runtime, to ensure fast matching of event stream events to SQL query results, even if your SQL query returns a large number of rows. Your applications must ensure to configure a cache for your database using Esper configuration, as such indexes are held with regular data in a cache. If your application does not enable caching of SQL query results, the engine does not build indexes on cached data.

The sample EPL statement below joins an event stream consisting of `OrderEvent` events with the results of an SQL query against the database named `MyRefDB` and table `SymbolReference`:

```
select symbol, symbolDesc from OrderEvent as orders,
  sql:MyRefDB ['select symbolDesc from SymbolReference'] as reference
where reference.symbol = orders.symbol
```

Notice how the EPL `where` clause joins the `OrderEvent` stream to the `SymbolReference` table. In this example, the SQL query itself does not have a SQL `where` clause and therefore returns all rows from table `SymbolReference`.

If your application enables caching, the SQL query fires only at the arrival of the first `OrderEvent` event. When the second `OrderEvent` arrives, the join execution uses the cached query result. If the caching policy that you

specified in the Esper database configuration evicts the SQL query result from cache, then the engine fires the SQL query again to obtain a new result and places the result in cache.

If SQL result caching is enabled and your EPL `where` clause, as shown in the above example, provides the properties to join, then the engine indexes the SQL query results in cache and retains the index together with the query result in cache. Thus your application can benefit from high performance index-based lookups as long as the SQL query results are found in cache.

The SQL result caches operate on the level of all result rows for a given parameter set. For example, if your query returns 10 rows for a certain set of parameter values then the cache treats all 10 rows as a single entry keyed by the parameter values, and the expiry policy applies to all 10 rows and not to each individual row.

It is also possible to join multiple autonomous database systems in a single query, for example:

```
select symbol, symbolDesc from OrderEvent as orders,
       sql:My_Oracle_DB ['select symbolDesc from SymbolReference'] as reference,
       sql:My_MySQL_DB ['select orderList from orderHistory'] as history
where reference.symbol = orders.symbol
and history.symbol = orders.symbol
```

4.13.3. Outer Joins With SQL Queries

You can use outer joins to join data obtained from an SQL query and control when an event is produced. Use a left outer join, such as in the next statement, if you need an output event for each event regardless of whether or not the SQL query returns rows. If the SQL query returns no rows, the join result populates null values into the selected properties.

```
select custId, custName from
       CustomerCallEvent as cce
left outer join
       sql:MyCustomerDB ["select cust_id, cust_name as custName
                           from Customer where cust_id = ${cce.custId}"] as cq
on cce.custId = cq.cust_id
```

The statement above always generates at least one output event for each `CustomerCallEvent`, containing all columns selected by the SQL query, even if the SQL query does not return any rows. Note the `on` expression that is required for outer joins. The `on` acts as an additional filter to rows returned by the SQL query.

4.13.4. Using Patterns to Request (Poll) Data

Pattern statements and SQL queries can also be applied together in useful ways. One such use is to poll or request data from a database at regular intervals or following the schedule of the crontab-like `timer:at`.

The next statement is an example that shows a pattern that fires every 5 seconds to query the `NewOrder` table for new orders:

```
insert into NewOrders
select orderId, orderAmount from
       pattern [every timer:interval(5 sec)],
       sql:MyCustomerDB ['select orderId, orderAmount from NewOrders']
```

4.13.5. Polling SQL Queries via Enumerator

Usually your SQL query will take part in a join and thus be triggered by an event or pattern occurrence. Instead, your application may need to poll a SQL query and thus use Esper query execution and caching facilities and

obtain event data and metadata.

Your EPL statement can specify an SQL statement without a join. Such a stand-alone SQL statement does not post new events, and may only be queried via the `iterator` poll API. Your EPL and SQL statement may still use variables.

The next statement assumes that a `price_var` variable has been declared. It selects from the relational database table named `NewOrder` all rows in which the `price` column is greater than the current value of the `price_var` EPL variable:

```
select * from sql:MyCustomerDB ['select * from NewOrder where ${price_var} > price']
```

Use the `GetEnumerator` and `GetSafeEnumerator` methods on `EPStatement` to obtain results. The statement does not post events to listeners, it is strictly passive in that sense.

4.13.6. ADO.NET Implementation Overview

The engine translates SQL queries into JDBC `System.Data.DbCommand` statements by replacing `${name}` parameters with `'?'` placeholders. It obtains name and type of result columns from the compiled `PreparedStatement` meta data when the EPL statement is created.

The engine supplies parameters to the compiled statement via the `SetParameter` method on `DbCommand`. The engine uses the `DbDataReader` to obtain column values.

4.13.7. No-Metadata Workaround

Certain database drivers are known to not return metadata for precompiled prepared SQL statements. This can be a problem as metadata is required by Esper. Esper obtains SQL result set metadata to validate an EPL statement and to provide column types for output events. Drivers that do not provide metadata for precompiled SQL statements require a workaround. Such drivers do generally provide metadata for executed SQL statements, however do not provide the metadata for precompiled SQL statements.

Please consult the Chapter 13, *Configuration* for the configuration options available in relation to metadata retrieval.

To obtain metadata for an SQL statement, Esper can alternatively fire a SQL statement which returns the same column names and types as the actual SQL statement but without returning any rows. This kind of SQL statement is referred to as a *sample* statement in below workaround description. The engine can then use the sample SQL statement to retrieve metadata for the column names and types returned by the actual SQL statement.

Applications can provide a sample SQL statement to retrieve metadata via the `metadatasql` keyword:

```
sql:database_name ["parameterized_sql_query" metadatasql "sql_meta_query"]
```

The `sql_meta_query` must be an SQL statement that returns the same number of columns, the same type of columns and the same column names as the `parameterized_sql_query`, and does not return any rows.

Alternatively, applications can choose not to provide an explicit sample SQL statement. If the EPL statement does not use the `metadatasql` syntax, the engine applies lexical analysis to the SQL statement. From the lexical analysis Esper generates a sample SQL statement adding a restrictive clause `"where 1=0"` to the SQL statement.

Alternatively, you can add the following tag to the SQL statement: `${$ESPER-SAMPLE-WHERE}`. If the tag exists in the SQL statement, the engine does not perform lexical analysis and simply replaces the tag with the SQL

where clause "where 1=0". Therefore this workaround is applicable to SQL statements that cannot be correctly lexically analyzed. The SQL text after the placeholder is not part of the sample query. For example:

```
select mycol from sql:myDB [
  'select mycol from mytesttable ${ESPER-SAMPLE-WHERE} where ....'], ...
```

If your *parameterized_sql_query* SQL query contains vendor-specific SQL syntax, generation of the metadata query may fail to produce a valid SQL statement. If you experience an SQL error while fetching metadata, use any of the above workarounds with the Oracle JDBC driver.

4.13.8. SQL Input Parameter and Column Output Conversion

As part of database access configuration you may optionally specify SQL type mappings. These mappings apply to all queries against the same database identified by name.

If your application must perform SQL-query-specific or EPL-statement-specific mapping or conversion between types, the facility to register a conversion callback exists as follows.

Use the `@Hook` instruction and `HookType.SQLCOL` as part of your EPL statement text to register a statement SQL parameter or column conversion hook. Implement the interface `com.espertech.esper.client.hook.SQLColumnTypeConversion` to perform the input parameter or column value conversion.

A sample statement with annotation is shown:

```
@Hook(type=HookType.SQLCOL, hook='MyDBTypeConvertor')
select * from sql:MyDB ['select * from MyEventTable']
```

The engine expects `MyDBTypeConvertor` to resolve to a class (considering engine imports) and instantiates one instance of `MyDBTypeConvertor` for each statement.

4.13.9. SQL Row Object Conversion

Your application may also directly convert a SQL result row into a type which is an opportunity for your application to interrogate and transform the SQL row result data freely before packing the data into a type. Your application can additionally indicate to skip SQL result rows.

Use the `@Hook` instruction and `HookType.SQLROW` as part of your EPL statement text to register a statement SQL output row conversion hook. Implement the interface `com.espertech.esper.client.hook.SQLOutputRowConversion` to perform the output row conversion.

A sample statement with annotation is shown:

```
@Hook(type=HookType.SQLROW, hook='MyDBRowConvertor')
select * from sql:MyDB ['select * from MyEventTable']
```

The engine expects `MyDBRowConvertor` to resolve to a class (considering engine imports) and instantiates one `MyDBRowConvertor` instance for each statement.

4.14. Accessing Non-Relational Data via Method Invocation

Your application may need to join data that originates from a web service, a distributed cache, an object-oriented database or simply data held in memory by your application. Esper accommodates this need by allowing

a method invocation (or procedure call or function) in the `from` clause of a statement.

The results of such a method invocation in the `from` clause plays the same role as a relational database table in an inner and outer join in SQL. The role is thus dissimilar to the role of a user-defined function, which may occur in any expression such as in the `select` clause or the `where` clause. Both are backed by one or more static methods provided by your class library.

Esper can join and outer join an unlimited number and all types of event streams to the data returned by your method invocation. In addition, Esper can be configured to cache the data returned by your method invocations.

Joins or outer joins in which only SQL statements or method invocations are listed in the `from` clause and no other event streams are termed *passive* joins. A passive join does not produce an insert or remove stream and therefore does not invoke statement listeners with results. A passive join can be iterated on (polled) using a statement's `GetSafeEnumerator` and `GetEnumerator` methods.

The following restrictions currently apply:

- Sub-views on a method invocations are not allowed; That is, one cannot create a time or length window on a method invocation. However one can use the `insert into` syntax to make join results available to a further statement.

4.14.1. Joining Method Invocation Results

The syntax for a method invocation in the `from` clause of an EPL statement is:

```
method:class_name.method_name[(parameter_expressions)]
```

The `method` keyword denotes a method invocation. It is followed by a class name and a method name separated by a dot (.) character. If you have parameters to your method invocation, these are placed in round brackets after the method name. Any expression is allowed as a parameter, and individual parameter expressions are separated by a comma. Expressions may also use event properties of the joined stream.

In the sample join statement shown next, the method 'lookupAsset' provided by class 'MyLookupLib' returns one or more rows based on the asset id (a property of the `AssetMoveEvent`) that is passed to the method:

```
select * from AssetMoveEvent, method:MyLookupLib.lookupAsset(assetId)
```

The following statement demonstrates the use of the `where` clause to join events to the rows returned by a method invocation, which in this example does not take parameters:

```
select assetId, assetDesc from AssetMoveEvent as asset,  
       method:MyLookupLib.getAssetDescriptions() as desc  
where asset.assetid = desc.assetid
```

Your method invocation may return zero, one or many rows for each method invocation. If you have caching enabled through configuration, then Esper can avoid the method invocation and instead use cached results. Similar to SQL joins, Esper also indexes cached result rows such that join operations based on the `where` clause or outer-join `on` clause can be very efficient, especially if your method invocation returns a large number of rows.

If the time taken by method invocations is critical to your application, you may configure local caches as Section 13.4.6, “Cache Settings for From-Clause Method Invocations” describes.

Esper analyzes the expression in the EPL `where` clause and outer-join `on` clause, if present, and builds the ap-

appropriate indexes from that information at runtime, to ensure fast matching of event stream events to method invocation results, even if your method invocation returns a large number of rows. Your applications must ensure to configure a cache for your method invocation using Esper configuration, as such indexes are held with regular data in a cache. If your application does not enable caching of method invocation results, the engine does not build indexes on cached data.

4.14.2. Polling Method Invocation Results via Iterator

Usually your method invocation will take part in a join and thus be triggered by an event or pattern occurrence. Instead, your application may need to poll a method invocation and thus use Esper query execution and caching facilities and obtain event data and metadata.

Your EPL statement can specify a method invocation in the `from` clause without a join. Such a stand-alone method invocation does not post new events, and may only be queried via the `iterator` poll API. Your EPL statement may still use variables.

The next statement assumes that a `category_var` variable has been declared. It polls the `GetAssetDescriptions` method passing the current value of the `category_var` EPL variable:

```
select * from method:MyLookupLib.GetAssetDescriptions(category_var) ]
```

Use the `GetEnumerator` and `GetSafeEnumerator` methods on `EPStatement` to obtain results. The statement does not post events to listeners, it is strictly passive in that sense.

4.14.3. Providing the Method

Your application must provide a type that exposes a public static method. The method must accept the same number and type of parameters as listed in the parameter expression list.

If your method invocation returns either no row or only one row, then the return type of the method can be a native type or a `System.Collections.Generic.IDictionary`. If your method invocation can return more than one row, then the return type of the method must be an array of native type or an array of `Map`.

If you are using a native type or an array of native type as the return type, then the class must adhere to property conventions described in this documentation.

If you are using `System.Collections.Generic.IDictionary` as the return type or an array of `Map`, then the map should have `String`-type keys and object values (`Map<String, Object>`). When using `Map` as the return type, your application must provide a second method that returns property metadata, as the next section outlines.

Your application method must return either of the following:

1. A `null` value or an empty array to indicate an empty result (no rows).
2. An object or `Map` to indicate a one-row result, or an array that consists of a single object or `Map`.
3. An array of objects or `Map` instances to return multiple result rows.

As an example, consider the method 'getAssetDescriptions' provided by class 'MyLookupLib' as discussed earlier:

```
select AssetId, AssetDesc from AssetMoveEvent as asset,
       method:com.mypackage.MyLookupLib.GetAssetDescriptions() as desc
```



```
where asset.AssetId = desc.AssetId
```

The 'GetAssetDescriptions' method may return multiple rows and is therefore declared to return an array of the class 'AssetDesc'. The class AssetDesc is a type (not shown here):

```
public class MyLookupLib {
    ...
    public static AssetDesc[] getAssetDescriptions() {
        ...
        return new AssetDesc[] {...};
    }
}
```

The example above specifies the full type name of the type 'MyLookupLib' class in the EPL statement. The package name does not need to be part of the EPL if your application imports the package using the auto-import configuration through the API or XML, as outlined in Section 13.4.5, “Class and package imports”.

4.14.4. Using a Map Return Type

Your application may return `System.Collections.Generic.IDictionary` or an array of `Map` from method invocations. If doing so, your application must provide metadata about each row: it must declare the property name and property type of each `Map` entry of a row. This information allows the engine to perform type checking of expressions used within the statement.

You declare the property names and types of each row by providing a method that returns property metadata. The metadata method must follow these conventions:

1. The method name providing the property metadata must have same method name appended by the literal `Metadata`.
2. The method must have an empty parameter list and must be declared public and static.
3. The method providing the metadata must return a `IDictionary` of `String` property name keys and `System.Type` property name types (`IDictionary<String, Type>`).

In the following example, a class 'MyLookupLib' provides a method to return historical data based on asset id and asset code:

```
select assetId, location, x_coord, y_coord from AssetMoveEvent as asset,
       method:com.mypackage.MyLookupLib.getAssetHistory(assetId, assetCode) as history
```

A sample implementation of the class 'MyLookupLib' is shown below.

```
public class MyLookupLib {
    ...
    // For each column in a row, provide the property name and type
    //
    public static Map<String, Class> getAssetHistoryMetadata() {
        Map<String, Class> propertyNames = new HashMap<String, Class>();
        propertyNames.put("location", String.class);
        propertyNames.put("x_coord", Integer.class);
        propertyNames.put("y_coord", Integer.class);
        return propertyNames;
    }
    ...
    // Lookup rows based on assetId and assetCode
    //
    public static Map<String, Object>[] getAssetHistory(String assetId, String assetCode) {
        Map rows = new Map[2]; // this sample returns 2 rows
    }
}
```

```

for (int i = 0; i < 2; i++) {
    rows[i] = new HashMap();
    rows[i].put("location", "somevalue");
    rows[i].put("x_coord", 100);
    // ... set more values for each row
}
return rows;
}

```

In the example above, the 'getAssetHistoryMetadata' method provides the property metadata: the names and types of properties in each row. The engine calls this method once per statement to determine event typing information.

The 'getAssetHistory' method returns an array of `Map` objects that are two rows. The implementation shown above is a simple example. The parameters to the method are the `assetId` and `assetCode` properties of the `AssetMoveEvent` joined to the method. The engine calls this method for each insert and remove stream event in `AssetMoveEvent`.

To indicate that no rows are found in a join, your application method may return either a `null` value or an array of size zero.

4.15. Creating and Using Named Windows

A *named window* is a global data window that can take part in many statement queries, and that can be inserted-into and deleted-from by multiple statements. A named window holds events of the same type or supertype, unless used with a variant stream.

The `create window` clause declares a new named window. The named window starts up empty unless populated from an existing named window at time of creation. Events must be inserted into the named window using the `insert into` clause. Events can also be deleted from a named window via the `on delete` clause.

Events enter the named window by means of `insert into` clause of a `select` statement. Events leave a named window either because the expiry policy of the declared data window removes events from the named window, or through statements that use the `on delete` clause to explicitly delete from a named window.

To query a named window, simply use the window name in the `from` clause of your statement, including statements that contain subqueries, joins and outer-joins.

A named window may also decorate an event to preserve original events as described in Section 4.10.4, “Decorated Events” and Section 4.15.2.1, “Named Windows Holding Decorated Events”. In addition, columns of a named window are allowed to hold events themselves, as further explained in Section 4.10.5, “Event as a Property” and Section 4.15.2.2, “Named Windows Holding Events As Property”.

To tune subquery performance when the subquery selects from a named window, consider the hints discussed in Section 4.11.7, “Hints Related to Subqueries”.

4.15.1. Creating Named Windows: the `create window` clause

The `create window` statement creates a named window by specifying a window name and one or more data window views, as well as the type of event to hold in the named window.

There are two syntaxes for creating a named window: The first syntax allows to model a named window after an existing event type or an existing named window. The second syntax is similar to the SQL create-table syntax and provides a list of column names and column types.

A new named window starts up empty. It must be explicitly inserted into by one or more statements, as discussed below. A named window can also be populated at time of creation from an existing named window.

If your application stops or destroys the statement that creates the named window, any consuming statements no longer receive insert or remove stream events. The named window can also not be deleted from after it was stopped or destroyed.

The `create window` statement posts to listeners any events that are inserted into the named window as new data. The statement posts all deleted events or events that expire out of the data window to listeners as the remove stream (old data). The named window contents can also be iterated on via the pull API to obtain the current contents of a named window.

Creation by Modelling after an Existing Type

The benefit of modelling a named window after an existing event type is that event properties can be nested, indexed, mapped or other types that your event objects may provide as properties, including the type of the underlying event itself. Also, using the wildcard (*) operator means your EPL does not need to list each individual property explicitly.

The syntax for creating a named window by modelling the named window after an existing event type, is as follows:

```
create window window_name.view_specifications
  [as] [select list_of_properties from] event_type_or_windowname
  [insert [where filter_expression]]
```

The *window_name* you assign to the named window can be any identifier. The name should not already be in use as an event type or stream name.

The *view_specifications* are one or more data window views that define the expiry policy for removing events from the data window. Named windows must explicitly declare a data window view. This is required to ensure that the policy for retaining events in the data window is well defined. To keep all events, use the keep-all view: It indicates that the named window should keep all events and only remove events from the named window that are deleted via the `on delete` clause. The view specification can only list data window views, derived-value views are not allowed since these don't represent an expiry policy. Data window views are listed in Chapter 11, *EPL Reference: Views*. View parameterization and staggering are described in Section 4.4.3, “Specifying Views”.

The `select` clause and *list_of_properties* are optional. If present, they specify the column names and, implicitly by definition of the event type, the column types of events held by the named window. Expressions other than column names are not allowed in the `select` list of properties. Wildcards (*) and wildcards with additional properties can also be used.

The *event_type_or_windowname* is required if using the model-after syntax. It provides the name of the event type of events held in the data window, unless column names and types have been explicitly selected via `select`. The name of an (existing) other named window is also allowed here. Please find more details in Section 4.15.6, “Populating a Named Window from an Existing Named Window”.

Finally, the `insert` clause and optional *filter_expression* are used if the new named windows is modelled after an existing named window, and the data of the existing named window is to be populated, upon time of creation of the new window, from the existing named window. The optional *filter_expression* can be used to exclude events.

The next statement creates a named window 'AllOrdersNamedWindow' for which the expiry policy is simply to

keep all events. Assume that the event type 'OrderMapEventType' has been configured. The named window is to hold events of type 'OrderMapEventType':

```
create window AllOrdersNamedWindow.win:keepall() as OrderMapEventType
```

The below sample statement demonstrates the `select` syntax. It defines a named window in which each row has the three properties 'symbol', 'volume' and 'price'. This named window actively removes events from the window that are older then 30 seconds.

```
create window OrdersTimeWindow.win:time(30 sec) as
  select symbol, volume, price from OrderEvent
```

In an alternate form, the `as` keyword can be used to rename columns, and constants may occur in the select clause as well:

```
create window OrdersTimeWindow.win:time(30 sec) as
  select symbol as sym, volume as vol, price, 1 as alertId from OrderEvent
```

Creation By Defining Columns Names and Types

The second syntax for creating a named window is by supplying column names and types:

```
create window window_name.view_specifications [as] (column_name column_type
[,column_name column_type [...]])
```

The *column_name* is an identifier providing the event property name. The *column_type* is also required for each column. Valid column types are listed in Section 4.18.1, “Creating Variables: the Create Variable clause” and are the same as for variable types.

For attributes that are array-type append `[]` (left and right brackets).

The next statement creates a named window:

```
create window SecurityEvent.win:time(30 sec)
  (ipAddress string, userId String, numAttempts int, properties String[])
```

Named window columns can hold events by declaring the column type as the event type name. Array-type is combination with event-type is also supported.

The next two statements declare an event type and create a named window with a column of the defined event type:

```
create schema SecurityData (name String, roles String[])
```

```
create window SecurityEvent.win:time(30 sec)
  (ipAddress string, userId String, secData SecurityData, historySecData SecurityData[])
```

Dropping or Removing Named Windows

There is no syntax to drop or remove a named window.

The `Dispose` method on the `EPStatement` that created the named window removes the named window. However the implicit event type associated with the named window remains active since further statements may continue to use that type. Therefore a named window of the same name can only be created again if the type information matches the prior declaration for a named window.

4.15.2. Inserting Into Named Windows

The `insert into` clause inserts events into named windows. Your application must ensure that the column names and types match the declared column names and types of the named window to be inserted into.

For inserting into a named window and for simultaneously checking if the inserted row already exists in the named window or for atomic update-insert operation on a named window, consider using `on-merge` as described in Section 4.15.9, “Triggered Upsert using the On-Merge Clause”. `On-merge` is similar to the SQL `merge` clause and provides what is known as an “Upsert” operation: Update existing events or if no existing event(s) are found then insert a new event, all in one atomic operation provided by a single EPL statement.

In this example we first create a named window using some of the columns of an `OrderEvent` event type:

```
create window OrdersWindow.win:keepall() as select symbol, volume, price from OrderEvent
```

The `insert into` the named window selects individual columns to be inserted:

```
insert into OrdersWindow(symbol, volume, price) select name, count, price from FXOrderEvent
```

An alternative form is shown next:

```
insert into OrdersWindow select name as symbol, vol as volume, price from FXOrderEvent
```

Following above statement, the engine enters every `FXOrderEvent` arriving into the engine into the named window 'OrdersWindow'.

The following EPL statements create a named window for an event type backed by a type and insert into the window any 'OrderEvent' where the symbol value is IBM:

```
create window OrdersWindow.win:time(30) as com.mycompany.OrderEvent
```

```
insert into OrdersWindow select * from com.mycompany.OrderEvent(symbol='IBM')
```

The last example adds one column named 'derivedPrice' to the 'OrderEvent' type by specifying a wildcard, and uses a user-defined function to populate the column:

```
create window OrdersWindow.win:time(30) as select *, price as derivedPrice from OrderEvent
```

```
insert into OrdersWindow select *, MyFunc.func(price, percent) as derivedPrice from OrderEvent
```

Event representations based on base classes or interfaces, and subclasses or implementing classes, are compatible as these statements show:

```
// create a named window for the base class
create window OrdersWindow.std:unique(name) as select * from ProductBaseEvent
```

```
// The ServiceProductEvent class subclasses the ProductBaseEvent class
insert into OrdersWindow select * from ServiceProductEvent
```

```
// The MerchandiseProductEvent class subclasses the ProductBaseEvent class
insert into OrdersWindow select * from MerchandiseProductEvent
```

To avoid duplicate events inserted in a named window and atomically check if a row already exists, use `on-merge` as outlined in Section 4.15.9, “Triggered Upsert using the On-Merge Clause”. An example:

```
on ServiceProductEvent as spe merge OrdersWindow as win
where win.id = spe.id when not matched then insert select *
```

Named Windows Holding Decorated Events

Decorated events hold an underlying event and add additional properties to the underlying event, as described further in Section 4.10.4, “Decorated Events”.

Here we create a named window that decorates `OrderEvent` events by adding an additional property named `priceTotal` to each `OrderEvent`. A matching `insert into` statement is also part of the sample:

```
create window OrdersWindow.win:time(30) as select *, price as priceTotal from OrderEvent
```

```
insert into OrdersWindow select *, price * unit as priceTotal from ServiceOrderEvent
```

The property type of the additional `priceTotal` column is the property type of the existing `price` property of `OrderEvent`.

Named Windows Holding Events As Property

Columns in a named window may also hold an event itself. More information on the `insert into` clause providing event columns is in Section 4.10.5, “Event as a Property”.

The next sample creates a named window that specifies two columns: A column that holds an `OrderEvent`, and a column by name `priceTotal`. A matching `insert into` statement is also part of the sample:

```
create window OrdersWindow.win:time(30) as select this, price as priceTotal from OrderEvent
```

```
insert into OrdersWindow select order, price * unit as priceTotal
from ServiceOrderEvent as order
```

Note that the `this` property must exist on the event and must return the event type itself (native events only). The property type of the additional `priceTotal` column is the property type of the existing `price` property.

4.15.3. Selecting From Named Windows

A named window can be referred to by any statement in the `from` clause of the statement. Filter criteria can also be specified. Additional views may be used onto named windows however such views cannot include data window views.

A statement selecting all events from a named window 'AllOrdersNamedWindow' is shown next. The named window must first be created via the `create window` clause before use.

```
select * from AllOrdersNamedWindow
```

The statement as above simply receives the unfiltered insert stream of the named window and reports that stream to its listeners. The `GetEnumerator` method returns all events in the named window, if any.

If your application desires to obtain the events removed from the named window, use the `rstream` keyword as this statement shows:

```
select rstream * from AllOrdersNamedWindow
```

The next statement derives an average price per symbol from all events posted by a named window:

```
select symbol, avg(price) from AllOrdersNamedWindow group by symbol
```

Your application may create a consuming statement such as above on an empty named window, or your application may create the above statement on an already filled named window. The engine provides correct results in either case: At the time of statement creation the Esper engine internally initializes the consuming statement from the current named window, also taking your declared filters into consideration. Thus, your statement deriving data from a named window does not start empty if the named window already holds one or more events. A consuming statement also sees the remove stream of an already populated named window, if any.

If you require a subset of the data in the named window, you can specify one or more filter expressions onto the named window as shown here:

```
select symbol, avg(price) from AllOrdersNamedWindow(sector='energy') group by symbol
```

By adding a filter to the named window, the aggregation and grouping as well as any views that may be declared onto the named window receive a filtered insert and remove stream. The above statement thus outputs, continuously, the average price per symbol for all orders in the named window that belong to a certain sector.

A side note on variables in filters filtering events from named windows: The engine initializes consuming statements at statement creation time and changes aggregation state continuously as events arrive. If the filter criteria contain variables and variable values changes, then the engine does not re-evaluate or re-build aggregation state. In such a case you may want to place variables in the `having` clause which evaluates on already-built aggregation state.

The following example further declares a view into the named window. Such a view can be a plug-in view or one of the built-in views, but cannot be a data window view (with the exception of the `std:groupwin` grouped-window view which is allowed).

```
select * from AllOrdersNamedWindow(volume>0, price>0).mycompany:mypluginview()
```

Data window views cannot be used onto named windows since named windows post insert and remove streams for the events entering and leaving the named window, thus the expiry policy and batch behavior are well defined by the data window declared for the named window. For example, the following is not allowed and fails at time of statement creation:

```
// not a valid statement
select * from AllOrdersNamedWindow.win:time(30 sec)
```

4.15.4. Triggered Select on Named Windows: the `on select` clause

The `on select` clause performs a one-time, non-continuous query on a named window every time a triggering event arrives or a triggering pattern matches. The query can consider all events in the named window, or only events that match certain criteria, or events that correlate with an arriving event or a pattern of arriving events.

The syntax for the `on select` clause is as follows:

```
on event_type[(filter_criteria)] [as stream_name]
[insert into insert_into_def]
select select_list
from window_name [as stream_name]
[where criteria_expression]
[group by grouping_expression_list]
[having grouping_search_conditions]
```

```
[order by order_by_expression_list]
```

The *event_type* is the name of the type of events that trigger the query against the named window. It is optionally followed by *filter_criteria* which are filter expressions to apply to arriving events. The optional *as* keyword can be used to assign an stream name. Patterns or named windows can also be specified in the *on* clause, see the samples in Section 4.15.8.1, “Using Patterns in the On Delete Clause”.

The *insert into* clause works as described in Section 4.10, “Merging Streams and Continuous Insertion: the Insert Into Clause”. The *select* clause is described in Section 4.3, “Choosing Event Properties And Events: the Select Clause”. For all clauses the semantics are equivalent to a join operation: The properties of the triggering event or events are available in the *select* clause and all other clauses.

The *window_name* in the *from* clause is the name of the named window to select events from. The *as* keyword is also available to assign a stream name to the named window. The *as* keyword is helpful in conjunction with wildcard in the *select* clause to select named window events via the syntax `select streamname.*`.

The optional *where* clause contains a *criteria_expression* that correlates the arriving (triggering) event to the events to be considered from the named window. The *criteria_expression* may also simply filter for events in the named window to be considered by the query.

The *group by* clause, the *having* clause and the *order by* clause are all optional and work as described in earlier chapters.

The similarities and differences between an *on select* clause and a regular or outer join are as follows:

1. A join is evaluated when any of the streams participating in the join have new events (insert stream) or events leaving data windows (remove stream). A join is therefore bi-directional or multi-directional. However, the *on select* statement has one triggering event or pattern that causes the query to be evaluated and is thus uni-directional.
2. The query within the *on select* statement is not continuous: It executes only when a triggering event or pattern occurs. Aggregation and groups are computed anew considering the contents of the named window at the time the triggering event arrives.

The *iterator* of the *EPStatement* object representing the *on select* clause returns the last batch of selected events in response to the last triggering event, or null if the last triggering event did not select any rows.

For correlated queries that correlate triggering events with events held by a named window, Esper internally creates efficient indexes to enable high performance querying of events. It analyzes the *where* clause to build one or more indexes for fast lookup in the named window based on the properties of the triggering event.

The next statement demonstrates the concept. Upon arrival of a *QueryEvent* event the statement selects all events in the 'OrdersNamedWindow' named window:

```
on QueryEvent select win.* from OrdersNamedWindow as win
```

The engine executes the query on arrival of a triggering event, in this case a *QueryEvent*. It posts the query results to any listeners to the statement, in a single invocation, as the new data array. By prefixing the wildcard (*) selector with the stream name, the *select* clause returns only events of the named window and does not also return triggering events.

The *where* clause filters and correlates events in the named window with the triggering event, as shown next:

```
on QueryEvent(volume>0) as query
select query.symbol, query.volume, win.symbol from OrdersNamedWindow as win
```



```
where win.symbol = query.symbol
```

Upon arrival of a QueryEvent, if that event has a value for the volume property that is greater than zero, the engine executes the query. The query considers all events currently held by the 'OrdersNamedWindow' that match the symbol property value of the triggering QueryEvent event. The engine then posts query results to the statement's listeners.

Aggregation, grouping and ordering of results are possible as this example shows:

```
on QueryEvent as queryEvent
select symbol, sum(volume) from OrdersNamedWindow as win
group by symbol
having volume > 0
order by symbol
```

The above statement outputs the total volume per symbol for those groups where the sum of the volume is greater than zero, ordered by symbol ascending. The engine computes and posts the output based on the current contents of the 'OrdersNamedWindow' named window considering all events in the named window, since the query does not have a where clause.

When using wildcard (*) to select from streams in an on-select clause, each stream, that is the triggering stream and the selected-upon named window, are selected, similar to a join. Therefore your wildcard select returns two columns: the triggering event and the selection result event, for each row.

```
on QueryEvent as queryEvent
select * from OrdersNamedWindow as win
```

The query above returns a queryEvent column and a win column for each event. If only a single stream's event is desired in the result, use select win.* instead.

To trigger an on-select when an update to the selected named window occurs or when the triggering event is the same event that is being inserted into the named window, specify the named window name as the event type.

The next query fires the select for every change to the named window OrdersNamedWindow:

```
on OrdersNamedWindow as trig
select onw.symbol, sum(onw.volume)
from OrdersNamedWindow as onw
where onw.symbol = trig.symbol
```

4.15.5. Triggered Playback from Named Windows: the on Insert clause

The on insert clause is an on select clause as described in the prior chapter with the addition of an insert into clause.

Similar to the on select clause, the engine executes the query when a triggering event arrives. It then provides the query results as an event stream to further statements. It populates the event stream that is named in the insert into clause.

The statement below provides the query results to any consumers of the MyOrderStream, upon arrival of a QueryEvent event:

```
on QueryEvent as query
insert into MyOrderStream
select win.* from OrdersNamedWindow as win
```

Here is a sample consuming statement of the `MyOrderStream`. The statement further filters the events provided by the `on insert` statement by user id and reports a total of volume per symbol:

```
select symbol, sum(volume) from MyOrderStream(userId='user1') group by symbol
```

4.15.6. Populating a Named Window from an Existing Named Window

Your EPL statement may specify the name of an existing named window when creating a new named window, and may use the `insert` keyword to indicate that the new named window is to be populated from the events currently held by the existing named window.

For example, and assuming the named window `OrdersNamedWindow` already exists, this statement creates a new named window `ScratchOrders` and populates all orders in `OrdersNamedWindow` into the new named window:

```
create window ScratchOrders.win:keepall() as OrdersNamedWindow insert
```

The `where` keyword is also available to perform filtering, for example:

```
create window ScratchBuyOrders.win:time(10) as OrdersNamedWindow insert where side = 'buy'
```

4.15.7. Updating Named Windows: the `on update` clause

An `on update` clause updates events held by a named window. The clause can be used to update all events, or only events that match certain criteria, or events that correlate with an arriving event or a pattern of arriving events.

For updating a named window and for simultaneously checking if the updated row exists in the named window or for atomic update-insert operation on a named window, consider using `on-merge` as described in Section 4.15.9, “Triggered Upsert using the On-Merge Clause”. `On-merge` is similar to the SQL `merge` clause and provides what is known as an “Upsert” operation: Update existing events or if no existing event(s) are found then insert a new event, all in one atomic operation provided by a single EPL statement.

The syntax for the `on update` clause is as follows:

```
on event_type[(filter_criteria)] [as stream_name]
update window_name [as stream_name]
set property_name = expression [, property_name = expression [, ...]]
[where criteria_expression]
```

The *event_type* is the name of the type of events that trigger an update of rows in a named window. It is optionally followed by *filter_criteria* which are filter expressions to apply to arriving events. The optional `as` keyword can be used to assign a name for use in expressions and the `where` clause. Patterns and named windows can also be specified in the `on` clause.

The *window_name* is the name of the named window to update events. The `as` keyword is also available to assign a name to the named window.

The comma-separated list of property names and expressions set the value of one or more properties. Subqueries may be part of expressions however aggregation functions and the `prev` or `prior` function may not be used in expressions.

The optional `where` clause contains a *criteria_expression* that correlates the arriving (triggering) event to the events to be updated in the named window. The *criteria_expression* may also simply filter for events in the

named window to be updated.

The `iterator` of the `EPStatement` object representing the `on update` clause can also be helpful: It returns the last batch of updated events in response to the last triggering event, in any order, or null if the last triggering event did not update any rows.

Statements that reference the named window receive the new event in the insert stream and the event prior to the update in the remove stream.

Let's look at a couple of examples. In the simplest form, this statement updates all events in the named window 'AllOrdersNamedWindow' when any 'UpdateOrderEvent' event arrives, setting the price property to zero for all events currently held by the named window:

```
on UpdateOrderEvent update AllOrdersNamedWindow set price = 0
```

This example adds a `where` clause to the example above. Upon arrival of a triggering 'ZeroVolumeEvent', the statement updates prices on any orders that have a volume of zero or less:

```
on ZeroVolumeEvent update AllOrdersNamedWindow set price = 0 where volume <= 0
```

The next example shows a more complete use of the syntax, and correlates the triggering event with events held by the named window:

```
on NewOrderEvent(volume>0) as myNewOrders
update AllOrdersNamedWindow as myNamedWindow
set price = myNewOrders.price
where myNamedWindow.symbol = myNewOrders.symbol
```

In the above sample statement, only if a 'NewOrderEvent' event with a volume greater than zero arrives does the statement trigger. Upon triggering, all events in the named window that have the same value for the symbol property as the triggering 'NewOrderEvent' event are then updated (their price property is set to that of the arriving event). The statement also showcases the `as` keyword to assign a name for use in the `where` expression.

For correlated queries (as above) that correlate triggering events with events held by a named window, Esper internally creates efficient indexes to enable high performance update of events.

Your application can subscribe a listener to your `on update` statements to determine update events. The statement post any events that are updated to all listeners attached to the statement as new data, and the events prior to the update as old data. Upon iteration, the statement provides the last update event, if any.

The following example shows the use of tags and a pattern. It sets the price value of orders to that of either a 'FlushOrderEvent' or 'OrderUpdateEvent' depending on which arrived:

```
on pattern [every ord=OrderUpdateEvent(volume>0) or every flush=FlushOrderEvent]
update AllOrdersNamedWindow as win
set price = case when ord.price is null then flush.price else ord.price end
where ord.id = win.id or flush.id = win.id
```

The following restrictions apply:

1. Each property to be updated must be writable.
2. For underlying event representations that are CLR objects, a event object class must be `Serializable` as discussed in Section 4.20.1, "Immutability and Updates" and must provide setter methods for updated properties.
3. When using an XML underlying event type, event properties in the XML document representation are not available for update.
4. Nested, indexed and mapped properties are not supported for update. Revision event types and variant

streams may also not be updated.

4.15.8. Deleting From Named Windows: the `on delete` clause

An `on delete` clause removes events from a named window. The clause can be used to remove all events, or only events that match certain criteria, or events that correlate with an arriving event or a pattern of arriving events.

The syntax for the `on delete` clause is as follows:

```
on event_type[(filter_criteria)] [as stream_name]
delete from window_name [as stream_name]
[where criteria_expression]
```

The *event_type* is the name of the type of events that trigger removal from the named window. It is optionally followed by *filter_criteria* which are filter expressions to apply to arriving events. The optional `as` keyword can be used to assign a name for use in the `where` clause. Patterns and named windows can also be specified in the `on` clause as described in the next section.

The *window_name* is the name of the named window to delete events from. The `as` keyword is also available to assign a name to the named window.

The optional `where` clause contains a *criteria_expression* that correlates the arriving (triggering) event to the events to be removed from the named window. The *criteria_expression* may also simply filter for events in the named window to be removed from the named window.

The iterator of the `EPStatement` object representing the `on delete` clause can also be helpful: It returns the last batch of deleted events in response to the last triggering event, in any order, or null if the last triggering event did not remove any rows.

Let's look at a couple of examples. In the simplest form, this statement deletes all events from the named window 'AllOrdersNamedWindow' when any 'FlushOrderEvent' arrives:

```
on FlushOrderEvent delete from AllOrdersNamedWindow
```

This example adds a `where` clause to the example above. Upon arrival of a triggering 'ZeroVolumeEvent', the statement removes from the named window any orders that have a volume of zero or less:

```
on ZeroVolumeEvent delete from AllOrdersNamedWindow where volume <= 0
```

The next example shows a more complete use of the syntax, and correlates the triggering event with events held by the named window:

```
on NewOrderEvent(volume>0) as myNewOrders
delete from AllOrdersNamedWindow as myNamedWindow
where myNamedWindow.symbol = myNewOrders.symbol
```

In the above sample statement, only if a 'NewOrderEvent' event with a volume greater than zero arrives does the statement trigger. Upon triggering, all events in the named window that have the same value for the symbol property as the triggering 'NewOrderEvent' event are then removed from the named window. The statement also showcases the `as` keyword to assign a name for use in the `where` expression.

For correlated queries (as above) that correlate triggering events with events held by a named window, Esper internally creates efficient indexes to enable high performance removal of events especially from named windows that hold large numbers of events.

Your application can subscribe a listener to your `on delete` statements to determine removed events. The statement post any events that are deleted from a named window to all listeners attached to the statement as new data. Upon iteration, the statement provides the last deleted event, if any.

Using Patterns in the `On Delete` Clause

By means of patterns the `on delete` clause and `on select` clause (described below) can look for more complex conditions to occur, possibly involving multiple events or the passing of time. The syntax for `on delete` with a pattern expression is show next:

```
on pattern [pattern_expression] [as stream_name]
delete from window_name [as stream_name]
[where criteria_expression]
```

The *pattern_expression* is any pattern that matches zero or more arriving events. Tags can be used to name events in the pattern and can occur in the optional `where` clause to correlate to events to be removed from a named window.

In the next example the triggering pattern fires every 10 seconds. The effect is that every 10 seconds the statement removes from 'MyNamedWindow' all rows:

```
on pattern [every timer:interval(10 sec)] delete from MyNamedWindow
```

The following example shows the use of tags in a pattern:

```
on pattern [every ord=OrderEvent(volume>0) or every flush=FlushOrderEvent]
delete from OrderWindow as win
where ord.id = win.id or flush.id = win.id
```

The pattern above looks for `OrderEvent` events with a volume value greater then zero and tags such events as 'ord'. The pattern also looks for `FlushOrderEvent` events and tags such events as 'flush'. The `where` clause deletes from the 'OrderWindow' named window any events that match in the value of the 'id' property either of the arriving events.

4.15.9. Triggered Upsert using the `on-Merge` Clause

The `on merge` clause is similar to the SQL `merge` clause. It provides what is known as an "Upsert" operation: Update existing events or if no existing event(s) are found then insert a new event, all in an atomic operation provided by a single EPL statement.

The syntax for the `on merge` clause is as follows:

```
on event_type[(filter_criteria)] [as stream_name]
merge [into] window_name [as stream_name]
[where criteria_expression]
  when [not] matched [and search_condition]
  then [
    insert [into streamname]
      [ (property_name [, property_name] [,...]) ]
      select select_expression [, select_expression[,...]]
      [where filter_expression]
    |
    update set property_name = expression [, property_name = expression [,...]]
      [where filter_expression]
    |
    delete
      [where filter_expression]
  ]
```

```
[then [insert|update|delete]] [,then ...]
[when ... then ... [...]]
```

The *event_type* is the name of the type of events that trigger the merge. It is optionally followed by *filter_criteria* which are filter expressions to apply to arriving events. The optional *as* keyword can be used to assign a name for use in the *where* clause. Patterns and named windows can also be specified in the *on* clause as described in prior sections.

The *window_name* is the name of the named window to insert, update or delete events. The *as* keyword is also available to assign a name to the named window.

The optional *where* clause contains a *criteria_expression* that correlates the arriving (triggering) event to the events to be considered of the named window. We recommend specifying a criteria expression that is as specific as possible.

Following the *where* clause is one or more *when matched* or *when not matched* clauses in any order. Each may have an additional search condition associated.

After each *when [not] matched* follow one or more *then* clauses that each contain the action to take: Either an *insert*, *update* or *delete* keyword.

After *when not matched* only *insert* action(s) are available. After *when matched* any *insert*, *update* and *delete* action(s) are available.

After *insert* follows, optionally, the *into* keyword followed by the stream name or named window to insert-into. If no *into* and stream name is specified, the insert applies to the current named window. It follows an optional list of columns inserted. It follows the required *select* keyword and one or more select clause expressions. The wildcard (*) is available in the select clause as well. It follows an optional *where*-clause that may return Boolean false to indicate that the action should not be applied.

After *update* follows the *set* keyword and one or more assignment pairs. It follows an optional *where*-clause that may return Boolean false to indicate that the action should not be applied.

After *delete* follows an optional *where*-clause that may return Boolean false to indicate that the action should not be applied.

When according to the *where*-clause *criteria_expression* the engine finds no events in the named window that match the condition, the engine evaluates each *when not matched* clause. If the optional search condition returns true or no search condition was provided then the engine performs all of the actions listed after each *then*.

When according to the *where*-clause *criteria_expression* the engine finds one or more events in the named window that match the condition, the engine evaluates each *when matched* clause. If the optional search condition returns true or no search condition was provided the engine performs all of the actions listed after each *then*.

The engine executes *when matched* and *when not matched* in the order specified. If the optional search condition returns true or no search condition was specified then the engine takes the associated action (or multiple actions for multiple *then* keywords). When the block of actions completed the engine proceeds to the next matching event, if any. After completing all matching events the engine continues to the next triggering event if any.

In the first example we declare a schema that provides a product id and that holds a total price:

```
create schema ProductTotalRec as (productId string, totalPrice double)
```

We create a named window that holds a row for each unique product:

```
create window ProductWindow.std:unique(productId) as ProductTotalRec
```

The events for this example are order events that hold an order id, product id, price, quantity and deleted-flag declared by the next schema:

```
create schema OrderEvent as (orderId string, productId string, price double,
    quantity int, deletedFlag boolean)
```

The following EPL statement utilizes `on-merge` to total up the price for each product based on arriving order events:

```
on OrderEvent oe
  merge ProductWindow pw
  where pw.productId = oe.productId
  when matched
    then update set totalPrice = totalPrice + oe.price
  when not matched
    then insert select productId, price as totalPrice
```

In the above example, when an order event arrives, the engine looks up in the product named window the matching row or rows for the same product id as the arriving event. In this example the engine always finds no row or one row as the product named window is declared with a unique data window based on product id. If the engine finds a row in the named window, it performs the update action adding up the price as defined under `when matched`. If the engine does not find a row in the named window it performs the insert action as defined under `when not matched`, inserting a new row.

The `insert` keyword may be followed by a list of columns as shown in this EPL snippet:

```
// equivalent to the insert shown in the last 2 lines in above EPL
...when not matched
  then insert(productId, totalPrice) select productId, price
```

The second example demonstrates the use of a `select`-clause with wildcard, a search condition and the `delete` keyword. It creates a named window that holds order events and employs `on-merge` to insert order events for which no corresponding order id was found, update quantity to the quantity provided by the last arriving event and delete order events that are marked as deleted:

```
create window OrderWindow.win:keepall() as OrderEvent
```

```
on OrderEvent oe
  merge OrderWindow pw
  where pw.orderId = oe.orderId
  when not matched
    then insert select *
  when matched and oe.deletedFlag=true
    then delete
  when matched
    then update set pw.quantity = oe.quantity, pw.price = oe.price
```

In the above example the `oe.deletedFlag=true` search condition instructs the engine to take the delete action only if the deleted-flag is set.

You may specify multiple actions by providing multiple `then` keywords each followed by an action. Each of the `insert`, `update` and `delete` actions can itself have a `where`-clause as well. If a `where`-clause exists for an action, the engine evaluates the `where`-clause and applies the action only if the `where`-clause returns Boolean true.

This example specifies two update actions and uses the `where`-clause to trigger different update behavior de-

pending on whether the order event price is less than zero:

```
on OrderEvent oe
merge OrderWindow pw
where pw.orderId = oe.orderId
when matched
  then update set pw.quantity = 0, pw.price = 0 where oe.price < 0
  then update set pw.quantity = oe.quantity, pw.price = oe.price where oe.price >= 0
```

To insert events into another stream and not the named window, use `insert into streamname`.

In the next example each matched-clause contains two actions, one action to insert a log event and a second action to insert, delete or update:

```
on OrderEvent oe
merge OrderWindow pw
where pw.orderId = oe.orderId
when not matched
  then insert into LogEvent select 'this is an insert' as name
  then insert select *
when matched and oe.deletedFlag=true
  then insert into LogEvent select 'this is a delete' as name
  then delete
when matched
  then insert into LogEvent select 'this is a update' as name
  then update set pw.quantity = oe.quantity, pw.price = oe.price
```

While the engine evaluates and executes all actions listed under the same matched-clause in order, you may not rely on updated field values of an earlier action to trigger the where-clause of a later action. Similarly you should avoid simultaneous update and delete actions for the same match: the engine does not guarantee whether the update or the delete take final affect.

For correlated queries (as above) that correlate triggering events with events held by a named window, Esper internally creates efficient indexes to enable high performance update and removal of events especially from named windows that hold large numbers of events.

Your application can subscribe a listener to `on merge` statements to determine inserted, updated and removed events. Statements post any events that are inserted to, updated or deleted from a named window to all listeners attached to the statement as new data and removed data. Upon iteration, the statement provides the last inserted events, if any.

4.15.10. Explicitly Indexing Named Windows

You may explicitly create an index on a named window. When executing on-demand (fire-and-forget) queries as described in Section 12.4.3, “On-Demand Snapshot Query Execution” using prepared or unprepared queries, an explicit index can help speed query performance.

You do not need to explicitly create an index unless using on-demand (fire-and-forget) queries. Therefore you do not need to explicitly create an index for `on-select`, `on-merge`, `on-update`, `on-delete` as well as subqueries or consuming statements against named windows. However if the engine finds an explicitly-created index matching the requirements of such on-query or subqueries it uses that index instead.

Please use the following syntax to create an explicit index on a named window:

```
create index index_name on named_window_name (property [hash|btree]
  [, property] [hash|btree] [,...] )
```


The *index_name* is the name assigned to the index. The name uniquely identifies the index and is used in engine logging.

The *named_window_name* is the name of an existing named window. If the named window has data already, the engine builds an index for the data in the named window.

The list of *property* names are the properties of events within the named window to include in the index. Following each property name you may specify the optional `hash` or `btree` keyword.

If you specify no keyword or the `hash` keyword for a property, the index will be a hash-based (unsorted) index in respect to that property. If you specify the `btree` keyword, the index will be a binary-tree-based sorted index in respect to that property. You may combine `hash` and `btree` properties for the same index. Specify `btree` for a property if you expect to perform numerical or string comparison using relational operators (`<`, `>`, `>=`, `<=`), the `between` or the `in` keyword for ranges and inverted ranges.

We list a few example EPL statements next that create a named window and create a single index:

```
// create a named window
create window UserProfileWindow.win:time(1 hour) select * from UserProfile
```

```
// create an index for the user id property only
create index UserProfileIndex on UserProfileWindow(userId)
```

Next, execute an on-demand fire-and-forget query as shown below, herein we use the prepared version to demonstrate:

```
String query = "select * from UserProfileWindow where userId='Joe'";
EPOnDemandPreparedQuery prepared = epRuntime.prepareQuery(query);
// query performance excellent in the face of large number of rows
EPOnDemandQueryResult result = prepared.execute();
// ...later ...
prepared.execute(); // execute a second time
```

By default, the engine builds a hash code -based index useful for direct comparison via equals (`=`). Filter expressions that look for ranges or use `in`, `between` do not benefit from the hash-based index and should use the `btree` keyword.

The next example creates a composite index over two fields `symbol` and `buyPrice`:

```
// create a named window
create window TickEventWindow.win:time(1 hour) as (symbol string, buyPrice double)
```

```
// create an index
create index idx1 on TickEventWindow(symbol hash, buyPrice btree)
```

A sample fire-and-forget query is shown below (this time the API calls are not shown):

```
// query performance excellent in the face of large number of rows
select * from TickEventWindow where symbol='GE' and buyPrice between 10 and 20
```

4.15.11. Versioning and Revision Event Type Use with Named Windows

As outlined in Section 2.9, “Updating, Merging and Versioning Events”, revision event types process updates or new versions of events held by a named window.

A revision event type is simply one or more existing pre-configured event types whose events are related, as

configured by static configuration, by event properties that provide same key values. The purpose of key values is to indicate that arriving events are related: An event amends, updates or adds properties to an earlier event that shares the same key values. No additional EPL is needed when using revision event types for merging event data.

Revision event types can be useful in these situations:

1. Some of your events carry only partial information that is related to a prior event and must be merged together.
2. Events arrive that add additional properties or change existing properties of prior events.
3. Events may carry properties that have null values or properties that do not exist (for example events backed by Map or XML), and for such properties the earlier value must be used instead.

To better illustrate, consider a revision event type that represents events for creation and updates to user profiles. Lets assume the user profile creation events carry the user id and a full profile. The profile update events indicate only the user id and the individual properties that actually changed. The user id property shall serve as a key value relating profile creation events and update events.

A revision event type must be configured to instruct the engine which event types participate and what their key properties are. Configuration is described in Section 13.4.22, “Revision Event Type” and is not shown here.

Assume that an event type `UserProfileRevisions` has been configured to hold profile events, i.e. creation and update events related by user id. This statement creates a named window to hold the last 1 hour of current profiles per user id:

```
create window UserProfileWindow.win:time(1 hour) select * from UserProfileRevisions
```

```
insert into UserProfileWindow select * from UserProfileCreation
```

```
insert into UserProfileWindow select * from UserProfileUpdate
```

In revision event types, the term *base* event is used to describe events that are subject to update. Events that update, amend or add additional properties to base events are termed *delta* events. In the example, base events are profile creation events and delta events are profile update events.

Base events are expected to arrive before delta events. In the case where a delta event arrives and is not related by key value to a base event or a revision of the base event currently held by the named window the engine ignores the delta event. Thus, considering the example, profile update events for a user id that does not have an existing profile in the named window are not applied.

When a base or delta event arrives, the insert and remove stream output by the named window are the current and the prior version of the event. Let's come back to the example. As creation events arrive that are followed by update events or more creation events for the same user id, the engine posts the current version of the profile as insert stream (new data) and the prior version of the profile as remove stream (old data).

Base events are also implicitly delta events. That is, if multiple base events of the same key property values arrive, then each base event provides a new version. In the example, if multiple profile creation events arrive for the same user id then new versions of the current profile for that user id are output by the engine for each base event, as it does for delta events.

The expiry policy as specified by view definitions applies to each distinct key value, or multiple distinct key values for composite keys. An expiry policy re-evaluates when new versions arrive. In the example, user profile events expire from the time window when no creation or update event for a given user id has been received for 1 hour.

Several strategies are available for merging or overlaying events as the configuration chapter describes in greater detail.

Any of the Map, XML and native event representations as well as plug-in event representations may participate in a revision event type. For example, profile creation events could be native events, while profile update events could be `System.Collections.Generic.IDictionary` events.

Delta events may also add properties to the revision event type. For example, one could add a new event type with security information to the revision event type and such security-related properties become available on the resulting revision event type.

The following restrictions apply to revision event types:

- Nested properties are only supported for the native event representation. Nested properties are not individually versioned; they are instead versioned by the containing property.
- Dynamic, indexed and mapped properties are only supported for nested properties and not as properties of the revision event type itself.

4.16. Declaring an Event Type: *Create Schema*

EPL allows declaring an event type via the `create schema` clause and also by means of the static or runtime configuration API `AddEventType` functions. The term schema and event type has the same meaning in EPL.

Your application can declare an event type by providing the property names and types or by providing a class name. Your application may also declare a variant stream schema.

When using the `create schema` syntax to declare an event type, the engine automatically removes the event type when there are no started statements referencing the event type, including the statement that declared the event type. When using the configuration API, the event type stays cached even if there are no statements that refer to the event type and until explicitly removed via the runtime configuration API.

4.16.1. Declare an Event Type by Providing Names and Types

The synopsis of the `create schema` syntax providing property names and types is:

```
create schema schema_name [as] (property_name property_type [,property_name property_type [,...]]
    [inherits inherited_event_type [, inherited_event_type] [,...]])
```

The *property_name* is an identifier providing the event property name. The *property_type* is also required for each property. Valid property types are listed in Section 4.18.1, “Creating Variables: the Create Variable clause” and in addition include:

1. Any type name, fully-qualified or the simple class name if imports are configured.
2. Add left and right square brackets `[]` to any type to denote an array-type event property.
3. Use an event type name as a property type.

The optional `inherits` keywords may be used to provide a list of event types that are supertypes to the declared type.

A few example event type declarations follow:

```
// Declare type SecurityEvent
create schema SecurityEvent as (ipAddress string, userId String, numAttempts int)
```

```
// Declare type AuthorizationEvent with the roles property being an array of String
// and the hostinfo property being an object
create schema AuthorizationEvent(group String, roles String[], hostinfo com.mycompany.HostNameInfo)

// Declare type CompositeEvent in which the innerEvents property is an array of SecurityEvent
create schema CompositeEvent(group String, innerEvents SecurityEvent[])

// Declare type WebPageVisitEvent that inherits all properties from PageHitEvent
create schema WebPageVisitEvent(userId String) inherits PageHitEvent
```

4.16.2. Declare an Event Type by Providing a Class Name

When using native types as the underlying event representation your application may simply provide the class name:

```
create schema schema_name [as] class_name
```

The *class_name* must be a fully-qualified class name (including the package name) if imports are not configured. If your application configures imports then the simple class name suffices without package name.

The next example statements declare an event type based on a class:

```
// Shows the use of a fully-qualified class name to declare the LoginEvent event type
create schema LoginEvent as com.mycompany.LoginValue

// When the configuration includes imports, the declaration does not need a package name
create schema LogoutEvent as SignoffValue
```

4.16.3. Declare a Variant Stream

A variant stream is a predefined stream into which events of multiple disparate event types can be inserted. Please see Section 4.10.3, “Merging Disparate Types of Events: Variant Streams” for rules regarding property visibility and additional information.

The synopsis is:

```
create variant schema schema_name [as] eventtype_name/* [, eventtype_name/*] [, ...]
```

Provide the `variant` keyword to declare a variant stream.

The `'*'` wildcard character declares a variant stream that accepts any type of event inserted into the variant stream.

Provide *eventtype_name* if the variant stream should hold events of the given type only. When using `insert into` to insert into the variant stream the engine checks to ensure the inserted event type or its supertypes match the required event type.

A few examples are shown below:

```
// Create a variant stream that accepts only LoginEvent and LogoutEvent event types
create variant schema SecurityVariant as LoginEvent, LogoutEvent

// Create a variant stream that accepts any event type
create variant schema AnyEvent as *
```

4.17. Splitting and Duplicating Streams

EPL offers a convenient syntax to splitting and duplicating events into multiple streams, and for receiving unmatched events among a set of filter criteria.

You may define a triggering event or pattern in the `on`-part of the statement followed by multiple `insert into`, `select` and `where` clauses.

The synopsis is:

```
on event_type[(filter_criteria)] [as stream_name]
insert into insert_into_def select select_list [where condition]
[insert into insert_into_def select select_list [where condition]]
[insert into...]
[output first | all]
```

The *event_type* is the name of the type of events that trigger the split stream. It is optionally followed by *filter_criteria* which are filter expressions to apply to arriving events. The optional `as` keyword can be used to assign a stream name. Patterns and named windows can also be specified in the `on` clause.

Following the `on`-clause is one or more *insert into* clauses as described in Section 4.10, “Merging Streams and Continuous Insertion: the Insert Into Clause” and *select* clauses as described in Section 4.3, “Choosing Event Properties And Events: the Select Clause”.

Each `select` clause may be followed by a `where` clause containing a condition. If the condition is true for the event, the engine transforms the event according to the `select` clause and inserts it into the corresponding stream.

At the end of the statement can be an optional `output` clause. By default the engine inserts into the first stream for which the `where` clause condition matches if one was specified, starting from the top. If you specify the `output all` keywords, then the engine inserts into each stream (not only the first stream) for which the `where` clause condition matches or that do not have a `where` clause.

If, for a given event, none of the `where` clause conditions match, the statement listener receives the unmatched event. The statement listener only receives unmatched events and does not receive any transformed or inserted events. The `GetEnumerator` method to the statement returns no events.

In the below sample statement, the engine inserts each `OrderEvent` into the `LargeOrders` stream if the order quantity is 100 or larger, or into the `SmallOrders` stream if the order quantity is smaller than 100:

```
on OrderEvent
  insert into LargeOrders select * where orderQty >= 100
  insert into SmallOrders select *
```

The next example statement adds a new stream for medium-sized orders. The new stream receives orders that have an order quantity between 20 and 100:

```
on OrderEvent
  insert into LargeOrders select orderId, customer where orderQty >= 100
  insert into MediumOrders select orderId, customer where orderQty between 20 and 100
  insert into SmallOrders select orderId, customer where orderQty > 0
```

As you may have noticed in the above statement, orders that have an order quantity of zero don't match any of the conditions. The engine does not insert such order events into any stream and the listener to the statement receives these unmatched events.

By default the engine inserts into the first `insert into stream` without a `where` clause or for which the `where` clause condition matches. To change the default behavior and insert into all matching streams instead (including those without a `where` clause), the `output all` keywords may be added to the statement.

The sample statement below shows the use of the `output all` keywords. The statement populates both the `LargeOrders` stream with large orders as well as the `VIPCustomerOrders` stream with orders for certain customers based on customer id:

```
on OrderEvent
  insert into LargeOrders select * where orderQty >= 100
  insert into VIPCustomerOrders select * where customerId in (1001, 1002)
output all
```

Since the `output all` keywords are present, the above statement inserts each order event into either both streams or only one stream or none of the streams, depending on order quantity and customer id of the order event. The statement delivers order events not inserted into any of the streams to the listeners and/or subscriber to the statement.

The following limitations apply to split-stream statements:

1. Aggregation functions and the `prev` and `prior` operators are not available in conditions and the `select`-clause.

4.18. Variables

A *variable* is a scalar, object or event value that is available for use in all statements including patterns. Variables can be used in an expression anywhere in a statement as well as in the `output` clause for output rate limiting.

Variables must first be declared or configured before use, by defining each variable's type and name. Variables can be created via the `create variable` syntax or declared by runtime or static configuration. Variables can be assigned new values by using the `on set` syntax or via the `SetVariableValue` methods on `EPRuntime`. The `EPRuntime` also provides method to read variable values.

When declaring a class-type or an event type variable you may read or set individual properties within the same variable.

The engine guarantees consistency and atomicity of variable reads and writes on a statement-level (this is a soft guarantee, see below). Variables are optimized for fast read access and are also multithread-safe.

Variables can also be removed, at runtime, by destroying all referencing statements including the statement that created the variable, or by means of the runtime configuration API.

4.18.1. Creating Variables: the `Create Variable` clause

The `create variable` syntax creates a new variable by defining the variable type and name. In alternative to the syntax, variables can also be declared in the runtime and engine configuration options.

The synopsis for creating a variable is as follows:

```
create variable variable_type variable_name [ = assignment_expression ]
```

The *variable_type* can be any of the following:

```
variable_type
:   string
|   char
|   character
|   bool
|   boolean
|   byte
|   short
|   int
|   integer
|   long
|   double
|   float
|   object
|   class_name
|   event_type_name
```

All variable types accept null values. The `object` type is for an untyped variable that can be assigned any value. You can provide a fully-qualified class name to declare a variable of that class type. You can also supply the name of an event type to declare a variable that holds an event of that type.

The *variable_name* is an identifier that names the variable. The variable name should not already be in use by another variable.

The *assignment_expression* is optional. Without an assignment expression the initial value for the variable is `null`. If present, it supplies the initial value for the variable.

The `EPStatement` object of the `create variable` statement provides access to variable values. The pull API methods `GetEnumerator` and `GetSafeEnumerator` return the current variable value. Listeners to the `create variable` statement subscribe to changes in variable value: the engine posts new and old value of the variable to all listeners when the variable value is updated by an `on set` statement.

The example below creates a variable that provides a threshold value. The name of the variable is `var_threshold` and its type is `long`. The variable's initial value is `null` as no other value has been assigned:

```
create variable long var_threshold
```

This statement creates an integer-type variable named `var_output_rate` and initializes it to the value ten (10):

```
create variable integer var_output_rate = 10
```

In addition to creating a variable via the `create variable` syntax, the runtime and engine configuration API also allows adding variables. The next code snippet illustrates the use of the runtime configuration API to create a string-typed variable:

```
epService.getEPAdministrator().getConfiguration()
    .addVariable("myVar", String.class, "init value");
```

The engine removes the variable if the statement that created the variable is destroyed and all statements that reference the variable are also destroyed. The `GetVariableNameUsedBy` and the `RemoveVariable` methods, both part of the runtime `ConfigurationOperations` API, provide use information and can remove a variable. If the variable was added via configuration, it can only be removed via the configuration API.

4.18.2. Setting Variable Values: the `on set` clause

The `on set` statement assigns a new value to one or more variables when a triggering event arrives or a triggering pattern occurs. Use the `SetVariableValue` methods on `EPRuntime` to assign variable values programmatically.

ally.

The synopsis for setting variable values is:

```
on event_type[(filter_criteria)] [as stream_name]
  set variable_name = expression [, variable_name = expression [...]]
```

The *event_type* is the name of the type of events that trigger the variable assignments. It is optionally followed by *filter_criteria* which are filter expressions to apply to arriving events. The optional `as` keyword can be used to assign an stream name. Patterns and named windows can also be specified in the `on` clause.

The comma-separated list of variable names and expressions set the value of one or more variables. Subqueries may be part of expressions however aggregation functions and the `prev` or `prior` function may not be used in expressions.

All new variable values are applied atomically: the changes to variable values by the `on set` statement become visible to other statements all at the same time. No changes are visible to other processing threads until the `on set` statement completed processing, and at that time all changes become visible at once.

The `EPStatement` object provides access to variable values. The pull API methods `GetEnumerator` and `GetSafeEnumerator` return the current variable values for each of the variables set by the statement. Listeners to the statement subscribe to changes in variable values: the engine posts new variable values of all variables to any listeners.

In the following example, a variable by name `var_output_rate` has been declared previously. When a `NewOutputRateEvent` event arrives, the variable is updated to a new value supplied by the event property 'rate':

```
on NewOutputRateEvent set var_output_rate = rate
```

The next example shows two variables that are updated when a `ThresholdUpdateEvent` arrives:

```
on ThresholdUpdateEvent as t
  set var_threshold_lower = t.lower,
     var_threshold_higher = t.higher
```

The sample statement shown next counts the number of pattern matches using a variable. The pattern looks for `OrderEvent` events that are followed by `CancelEvent` events for the same order id within 10 seconds of the `OrderEvent`:

```
on pattern[every a=OrderEvent -> (CancelEvent(orderId=a.orderId) where timer:within(10 sec))]
  set var_counter = var_counter + 1
```

4.18.3. Using Variables

A variable name can be used in any expression and can also occur in an output rate limiting clause. This section presents examples and discusses performance, consistency and atomicity attributes of variables.

The next statement assumes that a variable named 'var_threshold' was created to hold a total price threshold value. The statement outputs an event when the total price for a symbol is greater then the current threshold value:

```
select symbol, sum(price) from TickEvent
group by symbol
having sum(price) > var_threshold
```

In this example we use a variable to dynamically change the output rate on-the-fly. The variable

'var_output_rate' holds the current rate at which the statement posts a current count to listeners:

```
select count(*) from TickEvent output every var_output_rate seconds
```

Variables are optimized towards high read frequency and lower write frequency. Variable reads do not incur locking overhead (99% of the time) while variable writes do incur locking overhead.

The engine softly guarantees consistency and atomicity of variables when your statement executes in response to an event or timer invocation. Variables acquire a stable value (implemented by versioning) when your statement starts executing in response to an event or timer invocation, and variables do not change value during execution. When one or more variable values are updated via `on set` statements, the changes to all updated variables become visible to statements as one unit and only when the `on set` statement completes successfully.

The atomicity and consistency guarantee is a soft guarantee. If any of your application statements, in response to an event or timer invocation, execute for a time interval longer than 15 seconds (default interval length), then the engine may use current variable values after 15 seconds passed, rather than then-current variable values at the time the statement started executing in response to an event or timer invocation.

The length of the time interval that variable values are held stable for the duration of execution of a given statement is by default 15 seconds, but can be configured via engine default settings.

4.18.4. Object-Type Variables

A variable of type `object` (or `System.Object` via the API) can be assigned any value including null. When using an object-type variable in an expression, your statement may need to cast the value to the desired type.

The following sample EPL creates a variable by name `varobj` of type `object`:

```
create variable object varobj
```

4.18.5. Class and Event-Type Variables

The `create variable` syntax and the API accept a fully-qualified class name or alternatively the name of an event type. This is useful when you want a single variable to have multiple property values to read or set.

The next statement assumes that the event type `PageHitEvent` is declared:

```
create variable PageHitEvent varPageHitZero
```

These example statements show two ways of assigning to the variable:

```
// You may assign the complete event
on PageHitEvent(ip='0.0.0.0') pagehit set varPageHitZero = pagehit
```

```
// Or assign individual properties of the event
on PageHitEvent(ip='0.0.0.0') pagehit set varPageHitZero.userId = pagehit.userId
```

Similarly statements may use properties of class or event-type variables as this example shows:

```
select * from FirewallEvent(userId=varPageHitZero.userId)
```

When using class or event-type variables, in order for the engine to assign property values, the underlying event type must allow writing property values. If using native event classes the class must have setter methods and a

default constructor.

4.19. Contained-Event Selection

Contained-event selection is for use when an event contains properties that are themselves events. For example when application events are coarse-grained structures and you need to perform bulk operations on the rows of the property graph in an event.

Use the contained-event selection syntax in a filter expression such as in a pattern, `from` clause, `subselect`, `on-select` and `on-delete`. This section provides the synopsis and examples.

To review, in the `from` clause a *contained_selection* may appear after the event stream name and filter criteria, and before any view specifications.

The synopsis for *contained_selection* is as follows:

```
[select select_expressions from] property_expression [as property_alias] [where filter_expression]
```

The `select` clause and *select_expressions* are optional and may be used to select specific properties.

The *property_expression* is required and must be a valid property name that returns an event fragment, i.e. a property that can itself be represented as an event by the underlying event representation. Simple values such as integer or string are not fragments.

The *property_alias* can be provided to assign a name to the property expression.

The `where` clause and *filter_expression* is optional and may be used to filter out properties.

As an example event, consider a media order. A media order consists of order items as well as product descriptions. A media order event can be represented as an object graph (event representation), or a structure of nested Maps (Map event representation) or a XML document (XML DOM) or other custom plug-in event representation.

To illustrate, a sample media order event in XML event representation is shown below. Also, a XML event type can optionally be strongly-typed with an explicit XML XSD schema that we don't show here. Note that Map and native object representation can be considered equivalent for the purpose of this example.

Let us now assume that we have declared the event type `MediaOrder` as being represented by the root node `<mediaorder>` of such XML snip:

```
<mediaorder>
  <orderId>P0200901</orderId>
  <items>
    <item>
      <itemId>100001</itemId>
      <productId>B001</productId>
      <amount>10</amount>
      <price>11.95</price>
    </item>
  </items>
  <books>
    <book>
      <bookId>B001</bookId>
      <author>Heinlein</author>
      <review>
        <reviewId>1</reviewId>
        <comment>best book ever</comment>
      </review>
    </book>
  </books>
</mediaorder>
```

```

</book>
<book>
  <bookId>B002</bookId>
  <author>Isaac Asimov</author>
</book>
</books>
</mediaorder>

```

The next query utilizes the contained-event selection syntax to return each book:

```
select * from MediaOrder[books.book]
```

The result of the above query is one event per book. Output events contain only the book properties and not any of the mediaorder-level properties.

Note that, when using listeners, the engine delivers multiple results in one invocation of each listener. Therefore listeners to the above statement can expect a single invocation passing all book events within one media order event as an array.

To better illustrate the position of the contained-event selection syntax in a statement, consider the next two queries:

```
select * from MediaOrder(orderId='PO200901')[books.book]
```

The above query returns each book only for media orders with a given order id. This query illustrates a contained-event selection and a view:

```
select count(*) from MediaOrder[books.book].std:unique(bookId)
```

The sample above counts each book unique by book id.

Contained-event selection can be staggered. When staggering multiple contained-event selections the staggered contained-event selection is relative to its parent.

This example demonstrates staggering contained-event selections by selecting each review of each book:

```
select * from MediaOrder[books.book][review]
```

Listeners to the query above receive a row for each review of each book. Output events contain only the review properties and not the book or media order properties.

The following is not valid:

```

// not valid
select * from MediaOrder[books.book.review]

```

The `book` property in an indexed property (an array or collection) and thereby requires an index in order to determine which book to use. The expression `books.book[1].review` is valid and means all reviews of the second (index 1) book.

The contained-event selection syntax is part of the filter expression and may therefore occur in patterns and anywhere a filter expression is valid.

A pattern example is below. The example assumes that a `Cancel` event type has been defined that also has an `orderId` property:

```
select * from pattern [c=Cancel -> books=MediaOrder(orderId = c.orderId)[books.book] ]
```

When used in a pattern, a filter with a contained-event selection returns an array of events, similar to the match-until clause in patterns. The above statement returns, in the `books` property, an array of book events.

4.19.1. Select Clause in a Contained-Event Selection

The optional `select` clause provides control over which fields are available in output events. The expressions in the `select` clause apply only to the properties available underneath the property in the `from` clause, and the properties of the enclosing event.

When no `select` is specified, only the properties underneath the selected property are available in output events.

In summary, the `select` clause may contain:

1. Any expressions, wherein properties are resolved relative to the property in the `from` clause.
2. Use the wildcard (*) to provide all properties that exist under the property in the `from` clause.
3. Use the *property_alias*. * syntax to provide all properties that exist under a property in the `from` clause.

The next query's `select` clause selects each review for each book, and the order id as well as the book id of each book:

```
select * from MediaOrder[select orderId, bookId from books.book][select * from review]
// ... equivalent to ...
select * from MediaOrder[select orderId, bookId from books.book][review]]
```

Listeners to the statement above receive an event for each review of each book. Each output event has all properties of the review row, and in addition the `bookId` of each book and the `orderId` of the order. Thus `bookId` and `orderId` are found in each result event, duplicated when there are multiple reviews per book and order.

The above query uses wildcard (*) to select all properties from reviews. As has been discussed as part of the `select` clause, the wildcard (*) and *property_alias*. * do not copy properties for performance reasons. The wildcard syntax instead specifies the underlying type, and additional properties are added onto that underlying type if required. Only one wildcard (*) and *property_alias*. * (unless used with a column rename) may therefore occur in the `select` clause list of expressions.

All the following queries produce an output event for each review of each book. The next sample queries illustrate the options available to control the fields of output events.

The output events produced by the next query have all properties of each review and no other properties available:

```
select * from MediaOrder[books.book][review]
```

The following query is not a valid query, since the order id and book id are not part of the contained-event selection:

```
// Invalid select clause: orderId and bookId not produced.
select orderId, bookId from MediaOrder[books.book][review]
```

This query is valid. Note that output events carry only the `orderId` and `bookId` properties and no other data:

```
select orderId, bookId from MediaOrder[books.book][select orderId, bookId from review]
//... equivalent to ...
select * from MediaOrder[select orderId, bookId from books.book][review]]
```

This variation produces output events that have all properties of each book and only `reviewId` and `comment` for each review:

```
select * from MediaOrder[select * from books.book][select reviewId, comment from review]
// ... equivalent to ...
select * from MediaOrder[books.book as book][select book.*, reviewId, comment from review]
```

The output events of the next EPL have all properties of the order and only `bookId` and `reviewId` for each review:

```
select * from MediaOrder[books.book as book]
      [select mediaOrder.*, bookId, reviewId from review] as mediaOrder
```

This EPL produces output events with 3 columns: a column named `mediaOrder` that is the order itself, a column named `book` for each book and a column named `review` that holds each review:

```
insert into ReviewStream select * from MediaOrder[books.book as book]
      [select mo.* as mediaOrder, book.* as book, review.* as review from review as mo] as mo

// .. and a sample consumer of ReviewStream...
select mediaOrder.orderId, book.bookId, review.reviewId from ReviewStream
```

Please note these limitations:

1. Sub-selects, aggregation functions and the `prev` and `prior` operators are not available in contained-event selection.
2. Expressions in the `select` and `where` clause of a contained-event selection can only reference properties relative to the current event and property.

4.19.2. Where Clause in a Contained-Event Selection

The optional `where` clause may be used to filter out properties at the same level that the `where`-clause occurs.

The properties in the filter expression must be relative to the property in the `from` clause or the enclosing event.

This query outputs all books with a given author:

```
select * from MediaOrder[books.book where author = 'Heinlein']
```

This query outputs each review of each book where a review comment contains the word 'good':

```
select * from MediaOrder[books.book][review where comment like 'good']
```

4.19.3. Contained-Event Selection and Joins

This section discusses contained-event selection in joins.

When joining within the same event it is not required that views are specified. Recall, in a join or outer join there must be views specified that hold the data to be joined. For self-joins, no views are required and the join executes against the data returned by the same event.

This query inner-joins items to books where book id matches the product id:

```
select book.bookId, item.itemId
from MediaOrder[books.book] as book,
```

```
MediaOrder[items.item] as item
where productId = bookId
```

Query results for the above query when sending the media order event as shown earlier are:

book.bookId	item.itemId
B001	100001

The next example query is a left outer join. It returns all books and their items, and for books without item it returns the book and a `null` value:

```
select book.bookId, item.itemId
from MediaOrder[books.book] as book
left outer join
    MediaOrder[items.item] as item
on productId = bookId
```

Query results for the above query when sending the media order event as shown earlier are:

book.bookId	item.itemId
B001	100001
B002	null

A full outer join combines the results of both left and right outer joins. The joined table will contain all records from both tables, and fill in `null` values for missing matches on either side.

This example query is a full outer join, returning all books as well as all items, and filling in `null` values for book id or item id if no match is found:

```
select orderId, book.bookId, item.itemId
from MediaOrder[books.book] as book
full outer join
    MediaOrder[select orderId, * from items.item] as item
on productId = bookId
order by bookId, item.itemId asc
```

As in all other continuous queries, aggregation results are cumulative from the time the statement was created.

The following query counts the cumulative number of items in which the product id matches a book id:

```
select count(*)
from MediaOrder[books.book] as book,
    MediaOrder[items.item] as item
where productId = bookId
```

The `unidirectional` keyword in a join indicates to the query engine that aggregation state is not cumulative. The next query counts the number of items in which the product id matches a book id for each event:

```
select count(*)
from MediaOrder[books.book] as book unidirectional,
    MediaOrder[items.item] as item
where productId = bookId
```

4.20. Updating an Insert Stream: the Update IStream Clause

The `update istream` statement allows declarative modification of event properties of events entering a stream. Update is a pre-processing step to each new event, modifying an event before the event applies to any statements.

The synopsis of `update istream` is as follows:

```
update istream event_type [as stream_name]
  set property_name = set_expression [, property_name = set_expression] [... ]
  [where where_expression]
```

The *event_type* is the name of the type of events that the `update` applies to. The optional `as` keyword can be used to assign a name to the event type for use with subqueries, for example. Following the `set` keyword is a comma-separated list of property names and expressions that provide the event properties to change and values to set.

The optional `where` clause and expression can be used to filter out events to which to apply updates.

Listeners to an `update` statement receive the updated event in the insert stream (new data) and the event prior to the update in the remove stream (old data). Note that if there are multiple update statements that all apply to the same event then the engine will ensure that the output events delivered to listeners or subscribers are consistent with the then-current updated properties of the event (if necessary making event copies, as described below, in the case that listeners are attached to update statements). Iterating over an update statement returns no events.

As an example, the below statement assumes an `AlertEvent` event type that has properties named `severity` and `reason`:

```
update istream AlertEvent
  set severity = 'High'
  where severity = 'Medium' and reason like '%withdrawal limit%'
```

The statement above changes the value of the `severity` property to "High" for `AlertEvent` events that have a medium severity and contain a specific reason text.

Update statements apply the changes to event properties before other statements receive the event(s) for processing, e.g. "`select * from AlertEvent`" receives the updated `AlertEvent`. This is true regardless of the order in which your application creates statements.

When multiple update statements apply to the same event, the engine executes updates in the order in which update statements are created. We recommend the `@Priority` EPL annotation to define a deterministic order of processing updates, especially in the case where update statements get created and destroyed dynamically or multiple update statements update the same fields. The update statement with the highest `@Priority` value applies last.

The `update` clause can be used on streams populated via `insert into`, as this example utilizing a pattern demonstrates:

```
insert into DoubleWithdrawalStream
  select a.id, b.id, a.account as account, 0 as minimum
  from pattern [a=Withdrawal -> b=Withdrawal(id = a.id)]
```

```
update istream DoubleWithdrawalStream set minimum = 1000 where account in (10002, 10003)
```

When using `update` with named windows, any changes to event properties apply before an event enters the

named window.

Consider the next example (shown here with statement names in @Name EPL annotation, multiple EPL statements):

```
@Name("CreateWindow") create window MyWindow.win:time(30 sec) as AlertEvent

@Name("UpdateStream") update istream MyWindow set severity = 'Low' where reason = '%out of paper%'

@Name("InsertWindow") insert into MyWindow select * from AlertEvent

@Name("SelectWindow") select * from MyWindow
```

The `UpdateStream` statement specifies an `update` clause that applies to all events entering the named window. Note that `update` does not apply to events already in the named window at the time an application creates the `UpdateStream` statement, it only applies to new events entering the named window (after an application created the update statement).

Therefore, in the above example listeners to the `SelectWindow` statement as well as the `CreateWindow` statement receive the updated event, while listeners to the `InsertWindow` statement receive the original `AlertEvent` event (and not the updated event).

Subqueries can also be used in all expressions including the optional `where` clause.

This example demonstrates a correlated subquery in an assignment expression and also demonstrates the optional `as` keyword. It assigns the `phone` property of an `AlertEvent` event a new value based on the lookup within all unique `PhoneEvent` events (according to an `empid` property) correlating the `AlertEvent` property `reporter` with the `empid` property of `PhoneEvent`:

```
update istream AlertEvent as ae
  set phone =
    (select phone from PhoneEvent.std:unique(empid) where empid = ae.reporter)
```

When using `update`, please note these limitations:

1. Expressions may not use aggregation functions.
2. The `prev` and `prior` functions may not be used.
3. For underlying event representations that are CLR objects, a event object class must implement the `Serializable` behavior as discussed below.
4. When using an XML underlying event type, event properties in the XML document representation are not available for update.
5. Nested, indexed and mapped properties are not supported for update. Revision event types and variant streams may also not be updated.

4.20.1. Immutability and Updates

When updating event objects the engine maintains consistency across statements. The engine ensures that an update to an event does not impact the results of statements that look for or retain the original un-updated event. As a result the engine may need to copy an event object to maintain consistency.

In the case your application utilizes objects as the underlying event representation and an `update` statement updates properties on an object, then in order to maintain consistency across statements it is necessary for the engine to copy the object before changing properties (and thus not change the original object).

For application objects, the copy operation is implemented by serialization. Your event object must therefore implement the `Serializable` behavior to become eligible for update. As an alternative to serialization, you

may instead configure a copy method as part of the event type configuration via `ConfigurationEventTypeLegacy`.

4.21. Controlling Event Delivery : The `FOR` Clause

The engine delivers all result events of a given statement to the statement's listeners and subscriber (if any) in a single invocation of each listener and subscriber's `Update` method passing an array of result events. For example, a statement using a time-batch view may provide many result events after a time period passes, a pattern may provide multiple matching events or in a join the join cardinality could be multiple rows.

For statements that typically post multiple result events to listeners the `for` keyword controls the number of invocations of the engine to listeners and subscribers and the subset of all result events delivered by each invocation. This can be useful when your application listener or subscriber code expects multiple invocations or expects that invocations only receive events that belong together by some additional criteria.

The `for` keyword is a reserved keyword. It is followed by either the `grouped_delivery` keyword for grouped delivery or the `discrete_delivery` keyword for discrete delivery. The `for` clause is valid after any EPL select statement.

The synopsis for grouped delivery is as follows:

```
... for grouped_delivery (group_expression [, group_expression] [...])
```

The `group_expression` expression list provides one or more expressions to apply to result events. The engine invokes listeners and subscribers once for each distinct set of values returned by `group_expression` expressions passing only the events for that group.

The synopsis for discrete delivery is as follows:

```
... for discrete_delivery
```

With discrete delivery the engine invokes listeners and subscribers once for each result event passing a single result event in each invocation.

Consider the following example without `for`-clause. The time batch data view collects `RFIDEvent` events for 10 seconds and posts an array of result events:

```
select * from RFIDEvent.win:time_batch(10 sec)
```

Let's consider an example event sequence as follows:

Table 4.4. Sample Sequence of Events for `FOR` Keyword

Event
RFIDEvent(id:1, zone:'A')
RFIDEvent(id:2, zone:'B')
RFIDEvent(id:3, zone:'A')

Without `for`-clause and after the 10-second time period passes, the engine delivers an array of 3 events in a single invocation to listeners and the subscriber.

The next example specifies the `for`-clause and grouped delivery by zone:

```
select * from RFIDEvent.win:time_batch(10 sec) for grouped_delivery (zone)
```

With grouped delivery and after the 10-second time period passes, the above statement delivers result events in two invocations to listeners and the subscriber: The first invocation delivers an array of two events that contains zone A events with id 1 and 3. The second invocation delivers an array of 1 event that contains a zone B event with id 2.

The next example specifies the `for`-clause and discrete delivery:

```
select * from RFIDEvent.win:time_batch(10 sec) for discrete_delivery
```

With discrete delivery and after the 10-second time period passes, the above statement delivers result events in three invocations to listeners and the subscriber: The first invocation delivers an array of 1 event that contains the event with id 1, the second invocation delivers an array of 1 event that contains the event with id 2 and the third invocation delivers an array of 1 event that contains the event with id 3.

Remove stream events are also delivered in multiple invocations, one for each group, if your statement selects remove stream events explicitly via `irstream` or `rstream` keywords.

The `insert into` for inserting events into a stream is not affected by the `for`-clause.

The delivery order respects the natural sort order or the explicit sort order as provided by the `order by` clause, if present.

The following are known limitations:

1. The engine validates *group_expression* expressions against the output event type, therefore all properties specified in *group_expression* expressions must occur in the `select` clause.

Chapter 5. EPL Reference: Patterns

5.1. Event Pattern Overview

Event patterns match when an event or multiple events occur that match the pattern's definition. Patterns can also be time-based.

Pattern expressions consist of pattern atoms and pattern operators:

1. Pattern *atoms* are the basic building blocks of patterns. Atoms are filter expressions, observers for time-based events and plug-in custom observers that observe external events not under the control of the engine.
2. Pattern *operators* control expression lifecycle and combine atoms logically or temporally.

The below table outlines the different pattern atoms available:

Table 5.1. Pattern Atoms

Pattern Atom	Example
Filter expressions specify an event to look for.	<pre>StockTick(symbol='ABC', price > 100)</pre>
Time-based event observers specify time intervals or time schedules.	<pre>timer:interval(10 seconds)</pre> <pre>timer:at(*, 16, *, *, *)</pre>
Custom plug-in observers can add pattern language syntax for observing application-specific events.	<pre>myapplication:myobserver("http://someResource")</pre>

There are 4 types of pattern operators:

1. Operators that control pattern sub-expression repetition: `every`, `every-distinct`, `[num]` and `until`
2. Logical operators: `and`, `or`, `not`
3. Temporal operators that operate on event order: `->` (followed-by)
4. Guards are where-conditions that control the lifecycle of subexpressions. Examples are `timer:within`, `timer:withinmax` and `while-expression`. Custom plug-in guards may also be used.

Pattern expressions can be nested arbitrarily deep by including the nested expression(s) in `()` round parenthesis.

Underlying the pattern matching is a state machine that transitions between states based on arriving events and based on time advancing. A single event or advancing time may cause a reaction in multiple parts of your active pattern state.

5.2. How to use Patterns

5.2.1. Pattern Syntax

This is an example pattern expression that matches on every `ServiceMeasurement` events in which the value of the `latency` event property is over 20 seconds, and on every `ServiceMeasurement` event in which the `success` property is false. Either one or the other condition must be true for this pattern to match.

```
every (spike=ServiceMeasurement(latency>20000) or error=ServiceMeasurement(success=false))
```

In the example above, the pattern expression starts with an `every` operator to indicate that the pattern should fire for every matching events and not just the first matching event. Within the `every` operator in round brackets is a nested pattern expression using the `or` operator. The left hand of the `or` operator is a filter expression that filters for events with a high latency value. The right hand of the operator contains a filter expression that filters for events with error status. Filter expressions are explained in Section 5.4, “Filter Expressions In Patterns”.

The example above assigned the tags `spike` and `error` to the events in the pattern. The tags are important since the engine only places tagged events into the output event(s) that a pattern generates, and that the engine supplies to listeners of the pattern statement. The tags can further be selected in the `select`-clause of an EPL statement as discussed in Section 4.4.2, “Pattern-based Event Streams”.

Patterns can also contain comments within the pattern as outlined in Section 4.2.2, “Using Comments”.

Pattern statements are created via the `EPAdministrator` interface. The `EPAdministrator` interface allows to create pattern statements in two ways: Pattern statements that want to make use of the EPL `select` clause or any other EPL constructs use the `CreateEPL` method to create a statement that specifies one or more pattern expressions. EPL statements that use patterns are described in more detail in Section 4.4.2, “Pattern-based Event Streams”. Use the syntax as shown in below example.

```
EPAdministrator admin = EPServiceProviderManager.GetDefaultProvider().EPAdministrator;
String eventName = ServiceMeasurement.Name;
EPStatement myTrigger = admin.CreateEPL("select * from pattern [" +
    "every (spike=" + eventName + "(latency>20000) or error=" + eventName + "(success=false))"]);
```

Pattern statements that do not need to make use of the EPL `select` clause or any other EPL constructs can use the `CreatePattern` method, as in below example.

```
EPStatement myTrigger = admin.CreatePattern(
    "every (spike=" + eventName + "(latency>20000) or error=" + eventName + "(success=false))");
```

5.2.2. Patterns in EPL

A pattern may appear anywhere in the `from` clause of an EPL statement including joins and subqueries. Patterns may therefore be used in combination with the `where` clause, `group by` clause, `having` clause as well as output rate limiting and `insert into`.

In addition, a data window view can be declared onto a pattern. A data window declared onto a pattern only serves to retain pattern matches. A data window declared onto a pattern does not limit, cancel, remove or delete intermediate pattern matches of the pattern when pattern matches leave the data window.

This example statement demonstrates the idea by selecting a total price per customer over pairs of events (`ServiceOrder` followed by a `ProductOrder` event for the same customer id within 1 minute), occurring in the last 2 hours, in which the sum of price is greater than 100, and using a `where` clause to filter on name:

```
select a.custId, sum(a.price + b.price)
from pattern [every a=ServiceOrder ->
    b=ProductOrder(custId = a.custId) where timer:within(1 min)].win:time(2 hour)
where a.name in ('Repair', b.name)
group by a.custId
having sum(a.price + b.price) > 100
```

5.2.3. Subscribing to Pattern Events

When a pattern fires it publishes one or more events to any event handlers attached to the pattern statement. Event handlers are required to implement the `com.espertech.esper.client.UpdateEventHandler` delegate.

The example below shows an implementation of the `com.espertech.esper.client.UpdateEventHandler` delegate. We add the anonymous delegate to the `myPattern` statement created earlier. The listener code simply extracts the underlying event class.

```
myPattern.Events +=
    delegate(Object sender, UpdateEventArgs e) {
        ServiceMeasurement spike = (ServiceMeasurement) e.NewEvents[0].Get("spike");
        ServiceMeasurement error = (ServiceMeasurement) e.NewEvents[0].Get("error");
        ... // either spike or error can be null, depending on which occurred
        ... // add more logic here
    };
```

Event handlers receive the sender of the event (the statement) and an event that indicates details about the event that has occurred. The event includes an array of `EventBean` instances in the `NewEvents` property. There is one `EventBean` instance passed to the event handler for each combination of events that matches the pattern expression. At least one `EventBean` instance is always passed to the event handler.

The properties of each `EventBean` instance contain the underlying events that caused the pattern to fire, if events have been named in the filter expression via the `name=eventType` syntax. The property name is thus the name supplied in the pattern expression, while the property type is the type of the underlying class, in this example `ServiceMeasurement`.

5.2.4. Pulling Data from Patterns

Data can also be obtained from pattern statements via the `GetSafeEnumerator()` and `GetEnumerator()` methods on `EPStatement` (the pull API). If the pattern had fired at least once, then the iterator returns the last event for which it fired. The `MoveNext()` method can be used to determine if the pattern had fired. Additionally, extension methods such as `HasFirst` and `FirstOrDefault` can be applied to the statement because it implements the `IEnumerable` interface.

```
if (myPattern.HasFirst()) {
    ServiceMeasurement event = (ServiceMeasurement) view.FirstOrDefault().Get("alert");
    ... // some more code here to process the event
}
else {
    ... // no matching events at this time
}
```

Further, if a data window is defined onto a pattern, the iterator returns the pattern matches according to the data window expiry policy.

This pattern specifies a length window of 10 elements that retains the last 10 matches of A and B events, for use via iterator or for use in a join or subquery:

```
select * from pattern [every (A or B).win:length(10)]
```

5.2.5. Pattern Error Reporting

While the pattern compiler analyzes your pattern and verifies its integrity, it may not detect certain pattern errors that may occur at runtime. Sections of this pattern documentation point out common cases where the pattern engine will log a runtime error. We recommend turning on the log warning level at project development time to inspect and report on warnings logged. If a statement name is assigned to a statement then the statement name is logged as well.

5.3. Operator Precedence

The operators at the top of this table take precedence over operators lower on the table.

Table 5.2. Pattern Operator Precedence

Precedence	Operator	Description	Example
1	unary	every, not, every-distinct	<pre>every MyEvent timer:interval(5 min) and not MyEvent</pre>
2	guard post-fix	where timer:within and while (expression) (incl. withinmax and plug-in pattern guard)	<pre>MyEvent where timer:within(1 sec) a=MyEvent while (a.price between 1 and 10)</pre>
3	repeat	[num], until	<pre>[5] MyEvent [1..3] MyEvent until MyOtherEvent</pre>
4	and	and	<pre>every (MyEvent and MyOtherEvent)</pre>
5	or	or	<pre>every (MyEvent or MyOtherEvent)</pre>
6	followed-by	->	<pre>every (MyEvent -> MyOtherEvent)</pre>

If you are not sure about the precedence, please consider placing parenthesis () around your subexpressions. Parenthesis can also help make expressions easier to read and understand.

Note that we are also providing the EPL grammar as a HTML file as part of the documentation set on the project website.

The following table outlines sample equivalent expressions, with and without the use of parenthesis for subexpressions.

Table 5.3. Equivalent Pattern Expressions

Expression	Equivalent	Reason
every A or B	(every A) or B	The <code>every</code> operator has higher precedence then the <code>or</code> operator.
every A -> B or C	(every A) -> (B or C)	The <code>or</code> operator has higher precedence then the <code>followed-by</code> operator.
A -> B or B -> A	A -> (B or B) -> A	The <code>or</code> operator has higher precedence then the <code>followed-by</code> operator, specify as (A -> B) or (B -> A) instead.
A and B or C	(A and B) or C	The <code>and</code> operator has higher precedence then the <code>or</code> operator.
every A where timer:within(5)	every (A where timer:within(5))	The <code>every</code> operator has higher precedence then the <code>timer:within</code> guard postfix.
A -> B until C -> D	A -> (B until C) -> D	The <code>until</code> operator has higher precedence then the <code>followed-by</code> operator.
[5] A or B	([5] A) or B	The <code>[num]</code> repeat operator has higher precedence then the <code>or</code> operator.

5.4. Filter Expressions In Patterns

The simplest form of filter is a filter for events of a given type without any conditions on the event property values. This filter matches any event of that type regardless of the event's properties. The example below is such a filter. Note that this event pattern would stop firing as soon as the first `RfidEvent` is encountered.

```
com.mypackage.myevents.RfidEvent
```

To make the event pattern fire for every `RfidEvent` and not just the first event, use the `every` keyword.

```
every com.mypackage.myevents.RfidEvent
```

The example above specifies the fully-qualified type name as the event type. Via configuration, the event pattern above can be simplified by using the name that has been defined for the event type.

```
every RfidEvent
```

Interfaces and superclasses are also supported as event types. In the below example `IRfidReadable` is an interface class, and the statement matches any event that implements this interface:

```
every org.myorg.rfid.IRfidReadable
```

The filtering criteria to filter for events with certain event property values are placed within parenthesis after the event type name:

```
RfidEvent(category="Perishable")
```

All expressions can be used in filters, including static method invocations that return a boolean value:

```
RfidEvent(com.mycompany.MyRFIDLib.isInRange(x, y) or (x<0 and y < 0))
```

Filter expressions can be separated via a single comma ','. The comma represents a logical AND between expressions:

```
RfidEvent(zone=1, category=10)
...is equivalent to...
RfidEvent(zone=1 and category=10)
```

The following set of operators are highly optimized through indexing and are the preferred means of filtering high-volume event streams:

- equals =
- not equals !=
- comparison operators < , > , >= , <=
- ranges
 - use the `between` keyword for a closed range where both endpoints are included
 - use the `in` keyword and round () or square brackets [] to control how endpoints are included
 - for inverted ranges use the `not` keyword and the `between` or `in` keywords
- list-of-values checks using the `in` keyword or the `not in` keywords followed by a comma-separated list of values

At compile time as well as at run time, the engine scans new filter expressions for subexpressions that can be indexed. Indexing filter values to match event properties of incoming events enables the engine to match incoming events faster. The above list of operators represents the set of operators that the engine can best convert into indexes. The use of comma or logical `and` in filter expressions does not impact optimizations by the engine.

For more information on filters please see Section 4.4.1, “Filter-based Event Streams”. Contained-event selection on filters in patterns is further described in Section 4.19, “Contained-Event Selection”.

Filter criteria can also refer to events matching prior named events in the same expression. Below pattern is an example in which the pattern matches once for every `RfidEvent` that is preceded by an `RfidEvent` with the same asset id.

```
every e1=RfidEvent -> e2=RfidEvent(assetId=e1.assetId)
```

The syntax shown above allows filter criteria to reference prior results by specifying the event name tag of the prior event, and the event property name. The tag names in the above example were `e1` and `e2`. This syntax can be used in all filter operators or expressions including ranges and the `in` set-of-values check:

```
every e1=RfidEvent ->
  e2=RfidEvent(MyLib.isInRadius(e1.x, e1.y, x, y) and zone in (1, e1.zone))
```

An arriving event changes the truth value of all expressions that look for the event. Consider the pattern as follows:

```
every (RfidEvent(zone > 1) and RfidEvent(zone < 10))
```

The pattern above is satisfied as soon as only one event with zone in the interval [2, 9] is received.

5.5. Pattern Operators

5.5.1. Every

The `every` operator indicates that the pattern sub-expression should restart when the subexpression qualified by the `every` keyword evaluates to true or false. Without the `every` operator the pattern sub-expression stops when the pattern sub-expression evaluates to true or false.

As a side note, please be aware that a single invocation to the `UpdateEventHandler` may deliver multiple events in one invocation, since the `UpdateEventArgs` contain an array of values.

Thus the `every` operator works like a factory for the pattern sub-expression contained within. When the pattern sub-expression within it fires and thus quits checking for events, the `every` causes the start of a new pattern sub-expression listening for more occurrences of the same event or set of events.

Every time a pattern sub-expression within an `every` operator turns true the engine starts a new active sub-expression looking for more event(s) or timing conditions that match the pattern sub-expression. If the `every` operator is not specified for a subexpression, the subexpression stops after the first match was found.

This pattern fires when encountering an A event and then stops looking.

```
A
```

This pattern keeps firing when encountering A events, and doesn't stop looking.

```
every A
```

When using `every` operator with the `->` followed-by operator, each time the `every` operator restarts it also starts a new subexpression instance looking for events in the followed-by subexpression.

Let's consider an example event sequence as follows.

A₁ B₁ C₁ B₂ A₂ D₁ A₃ B₃ E₁ A₄ F₁ B₄

Table 5.4. 'Every' operator examples

Example	Description
<pre>every (A -> B)</pre>	<p>Detect an A event followed by a B event. At the time when B occurs the pattern matches, then the pattern matcher restarts and looks for the next A event.</p> <ol style="list-style-type: none"> Matches on B₁ for combination {A₁, B₁} Matches on B₃ for combination {A₂, B₃} Matches on B₄ for combination {A₄, B₄}
<pre>every A -> B</pre>	<p>The pattern fires for every A event followed by a B event.</p> <ol style="list-style-type: none"> Matches on B₁ for combination {A₁, B₁} Matches on B₃ for combination {A₂, B₃} and {A₃, B₃} Matches on B₄ for combination {A₄, B₄}
<pre>A -> every B</pre>	<p>The pattern fires for an A event followed by every B event.</p> <ol style="list-style-type: none"> Matches on B₁ for combination {A₁, B₁}. Matches on B₂ for combination {A₁, B₂}. Matches on B₃ for combination {A₁, B₃} Matches on B₄ for combination {A₁, B₄}

Example	Description
<pre>every A -> every B</pre>	<p>The pattern fires for every A event followed by every B event.</p> <ol style="list-style-type: none"> 1. Matches on B₁ for combination {A₁, B₁}. 2. Matches on B₂ for combination {A₁, B₂}. 3. Matches on B₃ for combination {A₁, B₃} and {A₂, B₃} and {A₃, B₃} 4. Matches on B₄ for combination {A₁, B₄} and {A₂, B₄} and {A₃, B₄} and {A₄, B₄}

The examples show that it is possible that a pattern fires for multiple combinations of events that match a pattern expression. Each combination is posted as an `EventBean` instance to the `UpdateEventHandler` implementation.

Let's consider the `every` operator in conjunction with a subexpression that matches 3 events that follow each other:

```
every (A -> B -> C)
```

The pattern first looks for A events. When an A event arrives, it looks for a B event. After the B event arrives, the pattern looks for a C event. Finally, when the C event arrives the pattern fires. The engine then starts looking for an A event again.

Assume that between the B event and the C event a second A₂ event arrives. The pattern would ignore the A₂ event entirely since it's then looking for a C event. As observed in the prior example, the `every` operator restarts the subexpression `A -> B -> C` only when the subexpression fires.

In the next statement the `every` operator applies only to the A event, not the whole subexpression:

```
every A -> B -> C
```

This pattern now matches for each A event that is followed by a B event and then a C event, regardless of when the A event arrives. Note that for each A event that arrives the pattern engine starts a new subexpression looking for a B event and then a C event, outputting each combination of matching events.

Limiting Subexpression Lifetime

As the introduction of the `every` operator states, the operator starts new subexpression instances and can cause multiple matches to occur for a single arriving event.

New subexpressions also take a very small amount of system resources and thereby your application should carefully consider when subexpressions must end when designing patterns. Use the `timer:within` construct and the `and not` constructs to end active subexpressions. The data window onto a pattern stream does not serve to limit pattern sub-expression lifetime.

Lets look at a concrete example. Consider the following sequence of events arriving:

A₁ A₂ B₁

This pattern matches on arrival of B₁ and outputs two events (an array of length 2 if using a listener). The two events are the combinations {A₁, B₁} and {A₂, B₁}:

```
every a=A -> b=B
```

The `and` `not` operators are used to end an active subexpression.

The next pattern matches on arrival of B_1 and outputs only the last A event which is the combination $\{A_2, B_1\}$:

```
every a=A -> (b=B and not A)
```

The `and` `not` operators cause the subexpression looking for $\{A_1, B?\}$ to end when A_2 arrives.

Similarly, in the pattern below the engine starts a new subexpression looking for a B event every 1 second. After 5 seconds there are 5 subexpressions active looking for a B event and 5 matches occur at once if a B event arrives after 5 seconds.

```
every timer:interval(1 sec) -> b=B
```

Again the `and` `not` operators can end subexpressions that are not intended to match any longer:

```
every timer:interval(1 sec) -> (b=B and not timer:interval(1 sec))
// equivalent to
every timer:interval(1 sec) -> (b=B where timer:within(1 sec))
```

Every Operator Example

In this example we consider a generic pattern in which the pattern must match for each A event followed by a B event and followed by a C event, in which both the B event and the C event must arrive within 1 hour of the A event. The first approach to the pattern is as follows:

```
every A -> (B -> C) where timer:within(1 hour)
```

Consider the following sequence of events arriving:

$A_1 \ A_2 \ B_1 \ C_1 \ B_2 \ C_2$

First, the pattern as above never stops looking for A events since the `every` operator instructs the pattern to keep looking for A events.

When A_1 arrives, the pattern starts a new subexpression that keeps A_1 in memory and looks for any B event. At the same time, it also keeps looking for more A events.

When A_2 arrives, the pattern starts a new subexpression that keeps A_2 in memory and looks for any B event. At the same time, it also keeps looking for more A events.

After the arrival of A_2 , there are 3 subexpressions active:

1. The first active subexpression with A_1 in memory, looking for any B event.
2. The second active subexpression with A_2 in memory, looking for any B event.
3. A third active subexpression, looking for the next A event.

In the pattern above, we have specified a 1-hour lifetime for subexpressions looking for B and C events. Thus, if no B and no C event arrive within 1 hour after A_1 , the first subexpression goes away. If no B and no C event arrive within 1 hour after A_2 , the second subexpression goes away. The third subexpression however stays around looking for more A events.

The pattern as shown above thus matches on arrival of C_1 for combination $\{A_1, B_1, C_1\}$ and for combination $\{A_2, B_1, C_1\}$, provided that B_1 and C_1 arrive within an hour of A_1 and A_2 .

You may now ask how to match on $\{A_1, B_1, C_1\}$ and $\{A_2, B_2, C_2\}$ instead, since you may need to correlate on a given property.

The pattern as discussed above matches every A event followed by the first B event followed by the next C event, and doesn't specifically qualify the B or C events to look for based on the A event. To look for specific B and C events in relation to a given A event, the correlation must use one or more of the properties of the A event, such as the "id" property:

```
every a=A -> (B(id=a.id -> C(id=a.id)) where timer:within(1 hour)
```

The pattern as shown above thus matches on arrival of C_1 for combination $\{A_1, B_1, C_1\}$ and on arrival of C_2 for combination $\{A_2, B_2, C_2\}$.

Sensor Example

This example looks at temperature sensor events named Sample. The pattern detects when 3 sensor events indicate a temperature of more then 50 degrees uninterrupted within 90 seconds of the first event, considering events for the same sensor only.

```
every sample=Sample(temp > 50) ->
( (Sample(sensor=sample.sensor, temp > 50) and not Sample(sensor=sample.sensor, temp <= 50))
->
  (Sample(sensor=sample.sensor, temp > 50) and not Sample(sensor=sample.sensor, temp <= 50))
) where timer:within(90 seconds))
```

The pattern starts a new subexpression in the round braces after the first followed-by operator for each time a sensor indicated more then 50 degrees. Each subexpression then lives a maximum of 90 seconds. Each subexpression ends if a temperature of 50 degree or less is encountered for the same sensor. Only if 3 temperature events in a row indicate more then 50 degrees, and within 90 seconds of the first event, and for the same sensor, does this pattern fire.

5.5.2. Every-Distinct

Similar to the `every` operator in most aspects, the `every-distinct` operator indicates that the pattern subexpression should restart when the subexpression qualified by the `every-distinct` keyword evaluates to true or false. In addition, the `every-distinct` eliminates duplicate results received from an active subexpression according to its distinct-value expressions.

The synopsis for the `every-distinct` pattern operator is:

```
every-distinct(distinct_value_expr [, distinct_value_exp[...][, expiry_time_period])
```

Within parenthesis are one or more *distinct_value_expr* expressions that return the values by which to remove duplicates.

You may optionally specify an *expiry_time_period* time period. If present, the pattern engine expires and removes distinct key values that are older then the time period, removing their associated memory and allowing such distinct values to match again. When your distinct value expressions return an unlimited number of values, for example when your distinct value is a timestamp or auto-increment column, you should always specify an expiry time period.

When specifying properties in the distinct-value expression list, you must ensure that the event types providing properties are tagged. Only properties of event types within filter expressions that are sub-expressions to the `every-distinct` may be specified.

For example, this pattern keeps firing for every A event with a distinct value for its `aprop` property:

```
every-distinct(a.aprop) a=A
```

Note that the pattern above assigns the `a` tag to the A event and uses `a.aprop` to identify the `prop` property as a value of the `a` event A.

A pattern that returns the first Sample event for each sensor, assuming `sensor` is a field that returns a unique id identifying the sensor that originated the Sample event, is:

```
every-distinct(s.sensor) s=Sample
```

The next pattern looks for pairs of A and B events and returns only the first pair for each combination of `aprop` of an A event and `bprop` of a B event:

```
every-distinct(a.aprop, b.bprop) (a=A and b=B)
```

The following pattern looks for A events followed by B events for which the value of the `aprop` of an A event is the same value of the `bprop` of a B event but only for each distinct value of `aprop` of an A event:

```
every-distinct(a.aprop) a=A -> b=B(bprop = a.aprop)
```

When specifying properties as part of distinct-value expressions, properties must be available from tagged event types in sub-expressions to the `every-distinct`.

The following patterns are not valid:

```
// Invalid: event type in filter not tagged
every-distinct(aprop) A

// Invalid: property not from a sub-expression of every-distinct
a=A -> every-distinct(a.aprop) b=B
```

When an active subexpression to `every-distinct` becomes permanently false, the distinct-values seen from the active subexpression are removed and the sub-expression within is restarted.

For example, the below pattern detects each A event distinct by the value of `aprop`.

```
every-distinct(a.aprop) (a=A and not B)
```

In the pattern above, when a B event arrives, the subexpression becomes permanently false and is restarted anew, detecting each A event distinct by the value of `aprop` without considering prior values.

When your distinct key is a timestamp or other non-unique property, specify an expiry time period.

The following example returns every distinct A event according to the timestamp property on the A event, retaining each timestamp value for 10 seconds:

```
every-distinct(a.timestamp, 10 seconds) a=A
```

In the example above, if for a given A event and its timestamp value the same timestamp value occurs again for another A event before 10 seconds passed, the A event is not a match. If 10 seconds passed the pattern indicates a second match.

5.5.3. Repeat

The repeat operator fires when a pattern sub-expression evaluates to true a given number of times. The synopsis is as follows:

```
[match_count] repeating_subexpr
```

The repeat operator is very similar to the `every` operator in that it restarts the *repeating_subexpr* pattern sub-expression up to a given number of times.

match_count is a positive number that specifies how often the *repeating_subexpr* pattern sub-expression must evaluate to true before the repeat expression itself evaluates to true, after which the engine may indicate a match.

For example, this pattern fires when the last of five A events arrives:

```
[5] A
```

Parenthesis must be used for nested pattern sub-expressions. This pattern fires when the last of a total of any five A or B events arrives:

```
[5] (A or B)
```

Without parenthesis the pattern semantics change, according to the operator precedence described earlier. This pattern fires when the last of a total of five A events arrives or a single B event arrives, whichever happens first:

```
[5] A or B
```

Tags can be used to name events in filter expression of pattern sub-expressions. The next pattern looks for an A event followed by a B event, and a second A event followed by a second B event. The output event provides indexed and array properties of the same name:

```
[2] (a=A -> b=B)
```

Using tags with repeat is further described in Section 5.5.4.6, “Tags and the Repeat Operator”.

Consider the following pattern that demonstrates the behavior when a pattern sub-expression becomes permanently false:

```
[2] (a=A and not C)
```

In the case where a C event arrives before 2 A events arrive, the pattern above becomes permanently false.

Lets add an `every` operator to restart the pattern and thus keep matching for all pairs of A events that arrive without a C event in between each pair:

```
every [2] (a=A and not C)
```

Since pattern matches return multiple A events, your select clause should use tag `a` as an array, for example:

```
select a[0].id, a[1].id from pattern [every [2] (a=A and not C)]
```

5.5.4. Repeat-Until

The repeat `until` operator provides additional control over repeated matching.

The repeat until operator takes an optional range, a pattern sub-expression to repeat, the `until` keyword and a second pattern sub-expression that ends the repetition. The synopsis is as follows:

```
[range] repeated_pattern_expr until end_pattern_expr
```

Without a *range*, the engine matches the *repeated_pattern_expr* pattern sub-expression until the *end_pattern_expr* evaluates to true, at which time the expression turns true.

An optional *range* can be used to indicate the minimum number of times that the *repeated_pattern_expr* pattern sub-expression must become true.

The optional *range* can also specify a maximum number of times that *repeated_pattern_expr* pattern sub-expression evaluates to true and retains tagged events. When this number is reached, the engine stops the *repeated_pattern_expr* pattern sub-expression.

Unbound Repeat

In the unbound repeat, without a *range*, the engine matches the *repeated_pattern_expr* pattern sub-expression until the *end_pattern_expr* evaluates to true, at which time the expression turns true. The synopsis is:

```
repeated_pattern_expr until end_pattern_expr
```

This is a pattern that keeps looking for A events until a B event arrives:

```
A until B
```

Nested pattern sub-expressions must be placed in parenthesis since the `until` operator has precedence over most operators. This example collects all A or B events for 10 seconds and places events received in indexed properties 'a' and 'b':

```
(a=A or b=B) until timer:interval(10 sec)
```

Bound Repeat Overview

The synopsis for the optional *range* qualifier is:

```
[ [low_endpoint] : [high_endpoint] ]
```

low_endpoint is an optional number that appears on the left of a colon (:), after which follows an optional *high_endpoint* number.

A range thus consists of a *low_endpoint* and a *high_endpoint* in square brackets and separated by a colon (:) characters. Both endpoint values are optional but either one or both must be supplied. The *low_endpoint* can be omitted to denote a range that starts at zero. The *high_endpoint* can be omitted to denote an open-ended range.

Some examples for valid ranges might be:

```
[3 : 10]
[:3]      // range starts at zero
[2:]      // open-ended range
```

The *low_endpoint*, if specified, defines the minimum number of times that the *repeated_pattern_expr* pattern sub-expression must become true in order for the expression to become true.

The *high_endpoint*, if specified, is the maximum number of times that the *repeated_pattern_expr* pattern sub-expression becomes true. If the number is reached, the engine stops the *repeated_pattern_expr* pattern sub-expression.

In all cases, only at the time that the *end_pattern_expr* pattern sub-expression evaluates to true does the expression become true. If *end_pattern_expr* pattern sub-expression evaluates to false, then the expression evaluates to false.

Bound Repeat - Open Ended Range

An open-ended range specifies only a low endpoint and not a high endpoint.

Consider the following pattern which requires at least three A events to match:

```
[3:] A until B
```

In the pattern above, if a B event arrives before 3 A events occurred, the expression ends and evaluates to false.

Bound Repeat - High Endpoint Range

A high-endpoint range specifies only a high endpoint and not a low endpoint.

In this sample pattern the engine will be looking for a maximum of 3 A events. The expression turns true as soon as a single B event arrives regardless of the number of A events received:

```
[ :3] A until B
```

The next pattern matches when a C or D event arrives, regardless of the number of A or B events that occurred:

```
[ :3] (a=A or b=B) until (c=C or d=D)
```

In the pattern above, if more than 3 A or B events arrive, the pattern stops looking for additional A or B events. The 'a' and 'b' tags retain only the first 3 (combined) matches among A and B events. The output event contains these tagged events as indexed properties.

Bound Repeat - Bounded Range

A bounded range specifies a low endpoint and a high endpoint.

The next pattern matches after at least one A event arrives upon the arrival of a single B event:

```
[1:3] a=A until B
```

If a B event arrives before the first A event, then the pattern does not match. Only the first 3 A events are returned by the pattern.

Tags and the Repeat Operator

The tags assigned to events in filter subexpressions within a repeat operator are available for use in filter expressions and also in any EPL clause.

This sample pattern matches 2 A events followed by a B event. Note the filter on B events: only a B event that has a value for the "beta" property that equals any of the "id" property values of the two A events is considered:


```
[2] A -> B(beta in (a[0].id, a[1].id))
```

The next EPL statement returns pairs of A events:

```
select a, a[0], a[0].id, a[1], a[1].id
from pattern [ every [2] a=A ]
```

The `select` clause of the statement above showcases different ways of accessing tagged events:

- The tag itself can be used to select an array of underlying events. For example, the 'a' expression above returns an array of underlying events of event type A.
- The tag as an indexed property returns the underlying event at that index. For instance, the 'a[0]' expression returns the first underlying A event, or null if no such A event was matched by the repeat operator.
- The tag as a nested, indexed property returns a property of the underlying event at that index. For example, the 'a[1].id' expression returns the 'id' property value of the second A event, or null if no such second A event was matched by the repeat operator.

You may not use indexed tags defined in the sub-expression to the repeat operator in the same subexpression. For example, in the following pattern the subexpression to the repeat operator is `(a=A() -> b=B(id=a[0].id))` and the tag `a` cannot be used in its indexed form in the filter for event B:

```
// invalid
every [2] (a=A() -> b=B(id=a[0].id))
```

You can use tags without an index:

```
// valid
every [2] (a=A() -> b=B(id=a.id))
```

5.5.5. And

Similar to the C# `&&` operator the `and` operator requires both nested pattern expressions to turn true before the whole expression turns true (a join pattern).

This pattern matches when both an A event and a B event arrive, at the time the last of the two events arrive:

```
A and B
```

This pattern matches on any sequence of an A event followed by a B event and then a C event followed by a D event, or a C event followed by a D and an A event followed by a B event:

```
(A -> B) and (C -> D)
```

Note that in an `and` pattern expression it is not possible to correlate events based on event property values. For example, this is an invalid pattern:

```
// This is NOT valid
a=A and B(id = a.id)
```

The above expression is invalid as it relies on the order of arrival of events, however in an `and` expression the order of events is not specified and events fulfill an `and` condition in any order. The above expression can be changed to use the followed-by operator:

```
// This is valid
a=A -> B(id = a.id)
// another example using 'and'...
a=A -> (B(id = a.id) and C(id = a.id))
```

Consider a pattern that looks for the same event:

```
A and A
```

The pattern above fires when a single A event arrives. The first arriving A event triggers a state transition in both the left and the right hand side expression.

In order to match after two A events arrive in any order, there are two options to express this pattern. The followed-by operator is one option and the repeat operator is the second option, as the next two patterns show:

```
A -> A
// ... or ...
[2] A
```

5.5.6. Or

Similar to the C# “||” operator the `or` operator requires either one of the expressions to turn true before the whole expression turns true.

Look for either an A event or a B event. As always, A and B can itself be nested expressions as well.

```
A or B
```

Detect all stock ticks that are either above or below a threshold.

```
every (StockTick(symbol='IBM', price < 100) or StockTick(symbol='IBM', price > 105))
```

5.5.7. Not

The `not` operator negates the truth value of an expression. Pattern expressions prefixed with `not` are automatically defaulted to true upon start, and turn permanently false when the expression within turns true.

The `not` operator is generally used in conjunction with the `and` operator or subexpressions as the below examples show.

This pattern matches only when an A event is encountered followed by a B event but only if no C event was encountered before either an A event and a B event, counting from the time the pattern is started:

```
(A -> B) and not C
```

Assume we'd like to detect when an A event is followed by a D event, without any B or C events between the A and D events:

```
A -> (D and not (B or C))
```

It may help your understanding to discuss a pattern that uses the `or` operator and the `not` operator together:

```
a=A -> (b=B or not C)
```

In the pattern above, when an A event arrives then the engine starts the subexpression B or not C. As soon as the subexpression starts, the not operator turns to true. The or expression turns true and thus your listener receives an invocation providing the A event in the property 'a'. The subexpression does not end and continues listening for B and C events. Upon arrival of a B event your listener receives a second invocation. If instead a C event arrives, the not turns permanently false however that does not affect the or operator (but would end an and operator).

To test for absence of an event, use timer:interval together with and not operators. The sample statement reports each 10-second interval during which no A event occurred:

```
every (timer:interval(10 sec) and not A)
```

In many cases the not operator, when used alone, does not make sense. The following example is invalid and will log a warning when the engine is started:

```
// not a sensible pattern
(not a=A) -> B(id=a.id)
```

5.5.8. Followed-by

The followed by -> operator specifies that first the left hand expression must turn true and only then is the right hand expression evaluated for matching events.

Look for an A event and if encountered, look for a B event. As always, A and B can itself be nested event pattern expressions.

```
A -> B
```

This is a pattern that fires when 2 status events indicating an error occur one after the other.

```
StatusEvent(status='ERROR') -> StatusEvent(status='ERROR')
```

A pattern that takes all A events that are not followed by a B event within 5 minutes:

```
every A -> (timer:interval(5 min) and not B)
```

A pattern that takes all A events that are not preceded by B within 5 minutes:

```
every (timer:interval(5 min) and not B -> A)
```

Limiting Sub-Expression Count

The followed-by -> operator can optionally be provided with an expression that limits the number of sub-expression instances of the right-hand side pattern sub-expression.

The synopsis for the followed-by operator with limiting expression is:

```
lhs_expression -[limit_expression]> rhs_expression
```

Each time the lhs_expression pattern sub-expression turns true the pattern engine starts a new rhs_expression pattern sub-expression. The limit_expression returns an integer value that defines a maximum number of pattern sub-expression instances that can simultaneously be present for the same rhs_expression.

When the limit is reached the pattern engine issues a notification to any condition handlers registered with the engine as described in Section 12.11, “Condition Handling” and does not start a new pattern sub-expression instance for the right-hand side pattern sub-expression.

For example, consider the following pattern which returns for every A event the first B event that matches the `id` field value of the A event:

```
every a=A -> b=B(id = a.id)
```

In the above pattern, every time an A event arrives (lhs) the pattern engine starts a new pattern sub-expression (rhs) consisting of a filter for the first B event that has the same value for the `id` field as the A event.

In some cases your application may want to limit the number of right-hand side sub-expressions because of memory concerns or to reduce output. You may add a limit expression returning an integer value as part of the operator.

This example employs the followed-by operator with a limit expression to indicate that maximally 2 filters for B events (the right-hand side pattern sub-expression) may be active at the same time:

```
every a=A -[2]> b=B(id = a.id)
```

Note that the limit expression in the example above is not a limit per value of `id` field, but a limit counting all right-hand side pattern sub-expression instances that are managed by that followed-by sub-expression instance.

If your followed-by operator lists multiple sub-expressions with limits, each limit applies to the immediate right-hand side. For example, the pattern below limits the number of filters for B events to 2 and the number of filters for C events to 3:

```
every a=A -[2]> b=B(id = a.id) -[3]> c=C(id = a.id)
```

5.5.9. Pattern Guards

Guards are where-conditions that control the lifecycle of subexpressions. Custom guard functions can also be used. The section Chapter 15, *Extension and Plug-in* outlines guard plug-in development in greater detail.

The pattern guard where-condition has no relationship to the EPL `where` clause that filters sets of events.

Take as an example the following pattern expression:

```
MyEvent where timer.within(10 sec)
```

In this pattern the `timer:within` guard controls the subexpression that is looking for `MyEvent` events. The guard terminates the subexpression looking for `MyEvent` events after 10 seconds after start of the pattern. Thus the pattern alerts only once when the first `MyEvent` event arrives within 10 seconds after start of the pattern.

The `every` keyword requires additional discussion since it also controls subexpression lifecycle. Let's add the `every` keyword to the example pattern:

```
every MyEvent where timer.within(10 sec)
```

The difference to the pattern without `every` is that each `MyEvent` event that arrives now starts a new subexpression, including a new guard, looking for a further `MyEvent` event. The result is that, when a `MyEvent` arrives within 10 seconds after pattern start, the pattern execution will look for the next `MyEvent` event to arrive within

10 seconds after the previous one.

By placing parentheses around the `every` keyword and its subexpression, we can have the `every` under the control of the guard:

```
(every MyEvent) where timer.within(10 sec)
```

In the pattern above, the guard terminates the subexpression looking for all `MyEvent` events after 10 seconds after start of the pattern. This pattern alerts for all `MyEvent` events arriving within 10 seconds after pattern start, and then stops.

Guards do not change the truth value of the subexpression of which the guard controls the lifecycle, and therefore do not cause a restart of the subexpression when used with the `every` operator. For example, the next pattern stops returning matches after 10 seconds unless a match occurred within 10 seconds after pattern start:

```
every ( (A and B) where timer.within(10 sec) )
```

The `timer:within` Pattern Guard

The `timer:within` guard acts like a stopwatch. If the associated pattern expression does not turn true within the specified time period it is stopped and permanently false.

The synopsis for `timer:within` is as follows:

```
timer:within(time_period_expression)
```

The *time_period_expression* is a time period (see Section 4.2.1, “Specifying Time Periods”) or an expression providing a number of seconds as a parameter. The interval expression may contain references to properties of prior events in the same pattern as well as variables and substitution parameters.

This pattern fires if an `A` event arrives within 5 seconds after statement creation.

```
A where timer:within (5 seconds)
```

This pattern fires for all `A` events that arrive within 5 seconds. After 5 seconds, this pattern stops matching even if more `A` events arrive.

```
(every A) where timer:within (5 seconds)
```

This pattern matches for any one `A` or `B` event in the next 5 seconds.

```
( A or B ) where timer:within (5 sec)
```

This pattern matches for any 2 errors that happen 10 seconds within each other.

```
every (StatusEvent(status='ERROR') -> StatusEvent(status='ERROR') where timer:within (10 sec))
```

The following guards are equivalent:

```
timer:within(2 minutes 5 seconds)
timer:within(125 sec)
timer:within(125)
```

The `timer:withinmax` Pattern Guard

The `timer:withinmax` guard is similar to the `timer:within` guard and acts as a stopwatch that additionally has a counter that counts the number of matches. It ends the subexpression when either the stopwatch ends or the match counter maximum value is reached.

The synopsis for `timer:withinmax` is as follows:

```
timer:withinmax(time_period_expression, max_count_expression)
```

The *time_period_expression* is a time period (see Section 4.2.1, “Specifying Time Periods”) or an expression providing a number of seconds.

The *max_count_expression* provides the maximum number of matches before the guard ends the subexpression.

Each parameter expression may also contain references to properties of prior events in the same pattern as well as variables and substitution parameters.

This pattern fires for every A event that arrives within 5 seconds after statement creation but only up to the first two A events:

```
(every A) where timer:withinmax (5 seconds, 2)
```

If the result of the *max_count_expression* is 1, the guard ends the subexpression after the first match and indicates the first match.

This pattern fires for the first A event that arrives within 5 seconds after statement creation:

```
(every A) where timer:withinmax (5 seconds, 1)
```

If the result of the *max_count_expression* is zero, the guard ends the subexpression upon the first match and does not indicate any matches.

This example receives every A event followed by every B event (as each B event arrives) until the 5-second subexpression timer ends or X number of B events have arrived (assume X was declared as a variable):

```
every A -> (every B) where timer:withinmax (5 seconds, X)
```

The `while` Pattern Guard

The `while` guard is followed by an expression that the engine evaluates for every match reported by the guard pattern sub-expression. When the expression returns false the pattern sub-expression ends.

The synopsis for `while` is as follows:

```
while (guard_expression)
```

The *guard_expression* is any expression that returns a boolean true or false. The expression may contain references to properties of prior events in the same pattern as well as variables and substitution parameters.

Each time the subexpression indicates a match, the engine evaluates *guard_expression* and if true, passes the match and when false, ends the subexpression.

This pattern fires for every A event until an A event arrives that has a value of zero or less for its `size` property (assuming A events have an integer `size` property).

```
(every a=A) while (a.size > 0)
```

Note the parenthesis around the `every` subexpression. They ensure that, following precedence rules, the guard applies to the `every` operator as well.

Guard Time Interval Expressions

The `timer:within` and `timer:withinmax` guards may be parameterized by an expression that contains one or more references to properties of prior events in the same pattern.

As a simple example, this pattern matches every A event followed by a B event that arrives within `delta` seconds after the A event:

```
every a=A -> b=B where timer:within (a.delta seconds)
```

Herein A event is assumed to have a `delta` property that provides the number of seconds to wait for B events. Each arriving A event may have a different value for `delta` and the guard is therefore parameterized dynamically based on the prior A event received.

When multiple events accumulate, for example when using the match-until or repeat pattern elements, an index must be provided:

```
[2] a=A -> b=B where timer:within (a[0].delta + a[1].delta)
```

The above pattern matches after 2 A events arrive followed by a B event within a time interval after the A event that is defined by the sum of the `delta` properties of both A events.

Combining Guard Expressions

You can combine guard expression by using parenthesis around each subexpression.

The below pattern matches for each A event while A events of size greater then zero arrive and only within the first 20 seconds:

```
((every a=A) while (a.size > 0)) where timer:within(20)
```

5.6. Pattern Atoms

5.6.1. Filter Atoms

Filter atoms have been described in section Section 5.4, “Filter Expressions In Patterns”.

5.6.2. Time-based Observer Atoms

Observers observe time-based events for which the thread-of-control originates by the engine timer or external timer event. Custom observers can also be developed that observe timer events or other engine-external application events such as a file-exists check. The section Chapter 15, *Extension and Plug-in* outlines observer plug-in development in greater detail.

timer:interval

The `timer:interval` observer waits for the defined time before the truth value of the observer turns true. The observer takes a time period (see Section 4.2.1, “Specifying Time Periods”) as a parameter, or an expression that returns the number of seconds.

The observer may be parameterized by an expression that contains one or more references to properties of prior events in the same pattern, or may also reference variables, substitution parameters or any other expression returning a numeric value.

After an A event arrived wait 10 seconds then indicate that the pattern matches.

```
A -> timer:interval(10 seconds)
```

The pattern below fires every 20 seconds.

```
every timer:interval(20 sec)
```

The next example pattern fires for every A event that is not followed by a B event within 60 seconds after the A event arrived. The B event must have the same "id" property value as the A event.

```
every a=A -> (timer:interval(60 sec) and not B(id=a.id))
```

Consider the next example, which assumes that the A event has a property `waittime`:

```
every a=A -> (timer:interval(a.waittime + 2) and not B(id=a.id))
```

In the above pattern the logic waits for 2 seconds plus the number of seconds provided by the value of the `waittime` property of the A event.

timer:at

The `timer:at` observer is similar in function to the Unix “crontab” command. At a specified time the expression turns true. The `at` operator can also be made to pattern match at regular intervals by using an `every` operator in front of the `timer:at` operator.

The syntax is: `timer:at (minutes, hours, days of month, months, days of week [, seconds])`.

The value for seconds is optional. Each element allows wildcard `*` values. Ranges can be specified by means of lower bounds then a colon `:` then the upper bound. The division operator `*/x` can be used to specify that every x_{th} value is valid. Combinations of these operators can be used by placing these into square brackets(`[]`).

The `timer:at` observer may also be parameterized by an expression that contains one or more references to properties of prior events in the same pattern, or may also reference variables, substitution parameters or any other expression returning a numeric value. The frequency division operator `*/x` and parameters lists within brackets(`[]`) are an exception: they may only contain variables, substitution parameters or numeric values.

This expression pattern matches every 5 minutes past the hour.

```
every timer:at(5, *, *, *, *)
```

The below `timer:at` pattern matches every 15 minutes from 8am to 5:45pm (hours 8 to 17 at 0, 15, 30 and 45 minutes past the hour) on even numbered days of the month as well as on the first day of the month.

```
timer:at (*/15, 8:17, [*/2, 1], *, *)
```


The below table outlines the fields, valid values and keywords available for each field:

Table 5.5. Properties offered by sample statement aggregating price

Field Name	Mandatory?	Allowed Values	Additional Keywords
Minutes	yes	0 - 59	
Hours	yes	0 - 23	
Days Of Month	yes	1 - 31	last, weekday, lastweekday
Months	yes	1 - 12	
Days Of Week	yes	0 (Sunday) - 6 (Saturday)	last
Seconds	no	0 - 59	

The keyword `last` used in the days-of-month field means the last day of the month (current month). To specify the last day of another month, a value for the month field has to be provided. For example: `timer:at(*, *, last, 2, *)` is the last day of February.

The `last` keyword in the day-of-week field by itself simply means Saturday. If used in the day-of-week field after another value, it means "the last xxx day of the month" - for example "5 last" means "the last friday of the month". So the last Friday of the current month will be: `timer:at(*, *, *, *, 5 last)`. And the last Friday of June: `timer:at(*, *, *, 6, 5 last)`.

The keyword `weekday` is used to specify the weekday (Monday-Friday) nearest the given day. Variant could include month like in: `timer:at(*, *, 30 weekday, 9, *)` which is Friday September 28th (no jump over month).

The keyword `lastweekday` is a combination of two parameters, the `last` and the `weekday` keywords. A typical example could be: `timer:at(*, *, *, lastweekday, 9, *)` which will define Friday September 28th (example year is 2007).

timer:at and the every Operator

When using `timer:at` with the `every` operator the crontab-like timer computes the next time at which the timer should fire based on the specification and the current time. When using `every`, the current time is the time the timer fired or the statement start time if the timer has not fired once.

For example, this pattern fires every 1 minute starting at 1:00pm and ending at 1:59pm, every day:

```
every timer:at(*, 13, *, *, *)
```

Assume the above statement gets started at 1:05pm and 20 seconds. In such case the above pattern fires every 1 minute starting at 1:06pm and ending at 1:59pm for that day and 1:00pm to 1:59pm every following day.

To get the pattern to fire only once at 1pm every day, explicitly specify the minute to start. The pattern below fires every day at 1:00pm:

```
every timer:at(0, 13, *, *, *)
```

By specifying a second resolution the timer can be made to fire every second, for instance:

```
every timer:at(*, *, *, *, *, *)
```

Chapter 6. EPL Reference: Match Recognize

6.1. Overview

Using *match recognize* patterns are defined in the familiar syntax of regular expressions.

The match recognize syntax presents an alternative way to specify pattern detection as compared to the EPL pattern language described in the previous chapter. A comparison of match recognize and EPL patterns is below.

The match recognize syntax is a proposal for incorporation into the SQL standard. It is thus subject to change as the standard evolves and finalizes (it has not finalized yet). Please consult row-pattern-recogniton-11-public [<http://dist.codehaus.org/esper/row-pattern-recogniton-11-public.pdf>] for further information.

You may be familiar with regular expressions in the context of finding text of interest in a string, such as particular characters, words, or patterns of characters. Instead of matching characters, match recognize matches sequences of events of interest.

Esper can apply match-recognize patterns in real-time upon arrival of new events in a stream of events (also termed incrementally, streaming or continuous). Esper can also match patterns on-demand via the *iterator* pull-API, if specifying a named window or data window on a stream.

6.2. Comparison of Match Recognize and EPL Patterns

This section compares pattern detection via match recognize and via the EPL pattern language.

Table 6.1. Comparison Match Recognize to EPL Patterns

Category	EPL Patterns	Match Recognize
Purpose	Pattern detection in sequences of events.	Same.
Standards	Not standardized, similar to Rapide pattern language.	Proposal for incorporation into the SQL standard.
Real-time Processing	Yes.	Yes.
On-Demand query via Iterator	No.	Yes.
Language	Nestable expressions consisting of boolean AND, OR, NOT and time or arrival-based constructs such as <code>-></code> (followed-by), <code>timer:within</code> and <code>timer:interval</code> .	Regular expression consisting of variables each representing conditions on events.
Event Types	An EPL pattern may react to multiple different types of events.	The input is a single type of event (unless used with variant streams).
Data Window Interaction	Disconnected, i.e. an event leaving a data window does not change pattern state.	Connected, i.e. an event leaving a data window removes the event from match selection.

Category	EPL Patterns	Match Recognize
Semantic Evaluation	Truth-value based: A EPL pattern such as (A and B) can fire when a single event arrives that satisfies both A and B conditions.	Sequence-based: A regular expression (A B) requires at least two events to match.
Time Relationship Between Events	The <code>timer:within</code> , <code>timer:interval</code> and <code>NOT</code> operator can expressively search for absence of events or other more complex timing relationships.	Some support for detecting absence of events using the <code>interval</code> clause.
Extensibility	Custom pattern objects, user-defined functions.	User-defined functions, custom aggregation functions.
Memory Use	Likely between 500 bytes to 2k per open sequence, depends on pattern.	Likely between 100 bytes to 1k per open sequence, depends on pattern.

6.3. Syntax

The synopsis is as follows:

```
match_recognize (
  [ partition by partition_expression [, partition_expression] [,...] ]
  measures measure_expression as col_name [, measure_expression as col_name ] [,...]
  [ all matches ]
  [ after match skip (past last row | to next row | to current row) ]
  pattern ( variable_regular_expr [, variable_regular_expr] [,...] )
  [ interval time_period ]
  define variable as variable_condition [, variable as variable_condition] [,...]
)
```

The `match_recognize` keyword starts the match recognize definition and occurs right after the `from` clause in an EPL `select` statement. It is followed by parenthesis that surround the match recognize definition.

`Partition by` is optional and may be used to specify that events are to be partitioned by one or more event properties or expressions. If there is no `Partition by` then all rows of the table constitute a single partition. The regular expression applies to events in the same partition and not across partitions.

The `measures` clause defines columns that contain expressions over the pattern variables. The expressions can reference partition columns, singleton variables, aggregates as well as indexed properties on the group variables. Each `measure_expression` expression must be followed by the `as` keyword and a `col_name` column name.

The `all matches` keywords are optional and instructs the engine to find all possible matches. By default matches are ranked and the engine returns a single match following an algorithm to eliminate duplicate matches, as described below. When specifying `all matches`, matches may overlap and may start at the same row.

The `after match skip` keywords are optional and serve to determine the resumption point of pattern matching after a match has been found. By default the behavior is `after match skip past last row`. This means that after eliminating duplicate matches, the logic skips to resume pattern matching at the next event after the last event of the current match.

The `pattern` component is used to specify a regular expression. The regular expression is built from variable

names, and may use the operators such as `*`, `+`, `?`, `*?`, `+?`, `??` quantifiers and `|` alteration (concatenation is indicated by the absence of any operator sign between two successive items in a pattern).

With the optional `interval` keyword and time period you can control how long the engine should wait for further events to arrive that may be part of a matching event sequence, before indicating a match (or matches) (not applicable to on-demand pattern matching).

`Define` is a mandatory component, used to specify the boolean condition that defines a variable name that is declared in the pattern. A variable name does not require a definition and if there is no definition, the default is a predicate that is always true. Such a variable name can be used to match any row.

6.3.1. Syntax Example

For illustration, the examples in this chapter use the `TemperatureSensorEvent` event. Each event has 3 properties: the `id` property is a unique event id, the `device` is a sensor device number and the `temp` property is a temperature reading. An event described as `"id=E1, device=1, temp=100"` is a `TemperatureSensorEvent` event with id "E1" for device 1 with a reading of 100.

This example statement looks for two `TemperatureSensorEvent` events from the same device, directly following each other, that indicate a jump in temperature of 10 or more between the two events:

```
select * from TemperatureSensorEvent
match_recognize (
  partition by device
  measures A.id as a_id, B.id as b_id, A.temp as a_temp, B.temp as b_temp
  pattern (A B)
  define
    B as Math.abs(B.temp - A.temp) >= 10
)
```

The `partition by` ensures that the regular expression applies to sequences of events from the same device.

The `measures` clause provides a list of properties or expressions to be selected from matching events. Each property name must be prefixed by the variable name.

In the `pattern` component the statement declares two variables: `A` and `B`. As a matter of convention, variable names are uppercase characters.

The `define` clause specifies no condition for variable `A`. This means that `A` defaults to true and any event matches `A`. Therefore, the pattern can start at any event.

The pattern `A B` indicates to look for a pattern in which an event that fulfills the condition for variable `A` is immediately followed by an event that fulfills the condition for variable `B`. A pattern thus defines the sequence (or sequences) of conditions that must be met for the pattern to fire.

Below table is an example sequence of events and output of the pattern:

Table 6.2. Example

Arrival Time	Tuple	Output Event (if any)
1000	id=E1, device=1, temp=50	
2000	id=E2, device=1, temp=55	
3000	id=E3, device=1, temp=60	
4000	id=E4, device=1, temp=70	a_id = E3, b_id = E4, a_temp = 60, b_temp = 70

Arrival Time	Tuple	Output Event (if any)
5000	id=E5, device=1, temp=85	
6000	id=E6, device=1, temp=85	
7000	id=E7, device=2, temp=100	

At time 4000 when event with id `E4` (or event `E4` or just `E4` for short) arrives the pattern matches and produces an output event. Matching then skips past the last event of the current match (`E4`) and begins at event `E5` (the default skip clause is past last row). Therefore events `E4` and `E5` do not constitute a match.

At time 3000, events `E1` and `E3` do not constitute a match as `E3` does not immediately follow `E1`, since there is `E2` in between.

At time 7000, event `E7` does not constitute a match as it is from device 2 and thereby not in the same partition as prior events.

6.4. Pattern and Pattern Operators

The `pattern` specifies the pattern to be recognized in the ordered sequence of events in a partition using regular expression syntax. Each variable name in a pattern corresponds to a boolean condition, which is specified later using the `define` component of the syntax. Thus the `pattern` can be regarded as implicitly declaring one or more variable names; the definition of those variable names appears later in the syntax. If a variable is not defined the variable defaults to true.

It is permitted for a variable name to occur more than once in a pattern, for example `pattern (A B A)`.

6.4.1. Operator Precedence

The operators at the top of this table take precedence over operators lower on the table.

Table 6.3. Match Recognize Regular Expression Operator Precedence

Precedence	Operator	Description	Example
1	Grouping	()	<code>(A B)</code>
2	Single-character duplication	* + ?	<code>A* B+ C?</code>
3	Concatenation	(no operator)	<code>A B</code>
4	Alternation		<code>A B</code>

If you are not sure about the precedence, please consider placing parenthesis () around your groups. Parenthesis can also help make expressions easier to read and understand.

6.4.2. Concatenation

The concatenation is indicated by the absence of any operator sign between two successive items in a pattern.

In below pattern the two items A and B have no operator between them. The pattern matches for any event immediately followed by an event from the same device that indicates a jump in temperature over 10:

```
select * from TemperatureSensorEvent
match_recognize (
  partition by device
  measures A.id as a_id, B.id as b_id, A.temp as a_temp, B.temp as b_temp
  pattern (A B)
  define
    B as Math.abs(B.temp - A.temp) >= 10
)
```

Please see the Section 6.3.1, “Syntax Example” for a sample event sequence.

6.4.3. Alternation

The alternation operator is a vertical bar (|).

The alternation operator has the lowest precedence of all operators. It tells the engine to match either everything to the left of the vertical bar, or everything to the right of the vertical bar. If you want to limit the reach of the alternation, you will need to use round brackets for grouping.

This example pattern looks for a sequence of an event with a temperature over 50 followed immediately by either an event with a temperature less than 45 or an event that indicates the temperature jumped by 10 (all for the same device):

```
select * from TemperatureSensorEvent
match_recognize (
  partition by device
  measures A.id as a_id, B.id as b_id, C.id as c_id
  pattern (A (B | C))
  define
    A as A.temp >= 50,
    B as B.temp <= 45,
    C as Math.abs(B.temp - A.temp) >= 10)
```

Below table is an example sequence of events and output of the pattern:

Table 6.4. Example

Arrival Time	Tuple	Output Event (if any)
1000	id=E1, device=1, temp=50	
2000	id=E2, device=1, temp=45	a_id=E1, b_id=E2, c_id=null
3000	id=E3, device=1, temp=46	
4000	id=E4, device=1, temp=48	
5000	id=E5, device=1, temp=50	
6000	id=E6, device=1, temp=60	a_id = E5, b_id = null, c_id=E6

6.4.4. Quantifiers Overview

Quantifiers are postfix operators with the following choices:

Table 6.5. Quantifiers

Quantifier	Meaning
*	Zero or more matches (greedy).
+	One or more matches (greedy).
?	Zero or one match (greedy).
*?	Zero or more matches (reluctant).
+?	One or more matches (reluctant).
??	Zero or one match (reluctant).

6.4.5. Variables Can be Singleton or Group

A *singleton variable* is a variable in a pattern that does not have a quantifier or has a zero-or-one quantifier (? or ??) and occurs only once in the pattern (except with alteration). In the `measures` clause a singleton variable can be selected as:

```
variableName.propertyName
```

Variables with a zero-or-more or one-or-more quantifier, or variables that occur multiple places in a pattern (except when using alteration), may match multiple events and are *group variables*. In the `measures` clause a group variable must be selected either by providing an index or via any of the aggregation functions, such as `first`, `last`, `count` and `sum`:

```
variableName[index].propertyName
```

```
last(variableName.propertyName)
```

Please find examples of singleton and group variables and example `measures` clauses below.

Additional Aggregation Functions

For group variables all existing aggregation functions can be used and in addition the following aggregation functions may be used:

Table 6.6. Syntax and results of aggregate functions

Aggregate Function	Result
<code>first([all distinct] expression)</code>	Returns the first value.
<code>last([all distinct] expression)</code>	Returns the last value.

6.4.6. Eliminating Duplicate Matches

The execution of match recognize is continuous and real-time by default. This means that every arriving event, or batch of events if using batching, evaluates against the pattern and matches are immediately indicated. Elimination of duplicate matches occurs between all matches of the arriving events (or batch of events) at a given time.

As an alternative, and if your application does not require continuous pattern evaluation, you may use the `iterator` API to perform on-demand matching of the pattern. For the purpose of indicating to the engine to not generate continuous results, specify the `@Hint('iterate_only')` hint.

When using one-or-more, zero-or-more or zero-or-one quantifiers (`?`, `+`, `*`, `??`, `++`, `*?`), the output of the real-time continuous query can differ from the output of the on-demand `iterator` execution: The continuous query will output a match (or multiple matches) as soon as matches are detected at a given time upon arrival of events (not knowing what further events may arrive). The on-demand execution, since it knows all possible events in advance, can determine the longest match(es). Thus elimination of duplicate matches can lead to different results between real-time and on-demand use.

If the `all matches` keywords are specified, then all matches are returned as the result and no elimination of duplicate matches as below occurs.

Otherwise matches to a pattern in a partition are ordered by preferment. Preferment is given to matches based on the following priorities:

1. A match that begins at an earlier row is preferred over a match that begins at a later row.
2. Of two matches matching a greedy quantifier, the longer match is preferred.
3. Of two matches matching a reluctant quantifier, the shorter match is preferred.

After ranking matches by preferment, matches are chosen as follows:

1. The first match by preferment is taken.
2. The pool of matches is reduced as follows based on the `SKIP TO` clause: If `SKIP PAST LAST ROW` is specified, all matches that overlap the first match are discarded from the pool. If `SKIP TO NEXT ROW` is specified, then all matches that overlap the first row of the first match are discarded. If `SKIP TO CURRENT ROW` is specified, then no matches are discarded.
3. The first match by preferment of the ones remaining is taken.
4. Step 2 is repeated to remove more matches from the pool.
5. Steps 3 and 4 are repeated until there are no remaining matches in the pool.

6.4.7. Greedy Or Reluctant

Reluctant quantifiers are indicated by an additional question mark (`*?`, `++`, `??`). Reluctant quantifiers try to match as few rows as possible, whereas non-reluctant quantifiers are greedy and try to match as many rows as possible.

Greedy and reluctant come into play only for match selection among multiple possible matches. When specifying `all matches` there is no difference between greedy and reluctant quantifiers.

Consider the below example. The conditions may overlap: an event with a temperature reading of 105 and over matches both A and B conditions:

```
select * from TemperatureSensorEvent
match_recognize (
  partition by device
  measures A.id as a_id, B.id as b_id
  pattern (A?? B?)
  define
    A as A.temp >= 100
    B as B.temp >= 105)
```

A sample sequence of events and pattern matches:

Table 6.7. Example

Arrival Time	Tuple	Output Event (if any)
1000	id=E1, device=1, temp=99	
2000	id=E2, device=2, temp=106	a_id=null, b_id=E2
3000	id=E3, device=1, temp=100	a_id=E3, b_id=null

As the ? qualifier on condition B is greedy, event E2 matches the pattern and is indicated as a B event by the `measure` clause (and not as an A event therefore `a_id` is null).

6.4.8. Quantifier - One Or More (+ and +?)

The one-or-more quantifier (+) must be matched one or more times by events. The operator is greedy and the reluctant version is +?.

In the below example with `pattern (A+ B+)` the pattern consists of two variable names, A and B, each of which is quantified with +, indicating that they must be matched one or more times.

The pattern looks for one or more events in which the temperature is over 100 followed by one or more events indicating a higher temperature:

```
select * from TemperatureSensorEvent
match_recognize (
  partition by device
  measures first(A.id) as first_a, last(A.id) as last_a, B[0].id as b0_id, B[1].id as b1_id
  pattern (A+ B+)
  define
    A as A.temp >= 100,
    B as B.temp > A.temp)
```

An example sequence of events that matches the pattern above is:

Table 6.8. Example

Arrival Time	Tuple	Output Event (if any)
1000	id=E1, device=1, temp=99	
2000	id=E2, device=1, temp=100	

Arrival Time	Tuple	Output Event (if any)
3000	id=E3, device=1, temp=100	
4000	id=E4, device=1, temp=101	first_a = E2, last_a = E3, b0_id = E4, b1_id = null
5000	id=E5, device=1, temp=102	

6.4.9. Quantifier - Zero Or More (* and *?)

The zero-or-more quantifier (*) must be matched zero or more times by events. The operator is greedy and the reluctant version is *?.

The pattern looks for a sequence of events in which the temperature starts out below 50 and then stays between 50 and 60 and finally comes over 60:

```
select * from TemperatureSensorEvent
match_recognize (
  partition by device
  measures A.id as a_id, count(B.id) as count_b, C.id as c_id
  pattern (A B* C)
  define
    A as A.temp < 50,
    B as B.temp between 50 and 60,
    C as C.temp > 60)
```

An example sequence of events that matches the pattern above is:

Table 6.9. Example

Arrival Time	Tuple	Output Event (if any)
1000	id=E1, device=1, temp=55	
2000	id=E2, device=1, temp=52	
3000	id=E3, device=1, temp=49	
4000	id=E4, device=1, temp=51	
5000	id=E5, device=1, temp=55	
6000	id=E5, device=1, temp=61	a_id=E3, count_b=2, c_id=E6

6.4.10. Quantifier - Zero Or One (? and ??)

The zero-or-one quantifier (?) must be matched zero or one time by events. The operator is greedy and the reluctant version is ??.

The pattern looks for a sequence of events in which the temperature is below 50 and then dips to over 50 and then to under 50 before indicating a value over 55:

```
select * from TemperatureSensorEvent
match_recognize (
  partition by device
  measures A.id as a_id, B.id as b_id, C.id as c_id, D.id as d_id
  pattern (A B? C? D)
```

```
define
  A as A.temp < 50,
  B as B.temp > 50,
  C as C.temp < 50,
  D as D.temp > 55)
```

An example sequence of events that matches the pattern above is:

Table 6.10. Example

Arrival Time	Tuple	Output Event (if any)
1000	id=E1, device=1, temp=44	
2000	id=E2, device=1, temp=49	
3000	id=E3, device=1, temp=51	
4000	id=E4, device=1, temp=49	
5000	id=E5, device=1, temp=56	a_id=E2, b_id=E3, c_id=E4, d_id=E5
6000	id=E5, device=1, temp=61	

6.5. Define Clause

Within `define` are listed the boolean conditions that defines a variable name that is declared in the pattern.

A variable name does not require a definition and if there is no definition, the default is a predicate that is always true. Such a variable name can be used to match any row.

The definitions of variable names may reference the same or other variable names as prior examples have shown.

If a variable in your condition expression is a singleton variable, then only individual columns may be referenced. If the variable is not matched by an event, a `null` value is returned.

If a variable in your condition expression is a group variable, then only indexed columns may be referenced. If the variable is not matched by an event, a `null` value is returned.

Aggregation functions are not allowed within expressions of the `define` clause.

6.5.1. The `prev` Operator

The `prev` function may be used in a `define` expression to access columns of the previous row of a variable name. If there is no previous row, the `null` value is returned.

The `prev` function can accept an optional non-negative integer argument indicating the offset to the previous rows. That argument must be a constant. In this case, the engine returns the property from the N-th row preceding the current row, and if the row doesn't exist, it returns `null`.

This function can access variables currently defined, for example:

```
Y as Y.price < prev(Y.price, 2)
```

It is not legal to use `prev` with another variable then the one being defined:

```
// not allowed
Y as Y.price < prev(X.price, 2)
```

The `prev` function returns properties of events in the same partition. Also, it returns properties of events according to event order-of-arrival. When using data windows or deleting events from a named window, the remove stream does not remove events from the `prev` function.

The pattern looks for an event in which the temperature is greater or equal 100 and that, relative to that event, has an event preceding it by 2 events that also had a temperature greater or equal 100:

```
select * from TemperatureSensorEvent
match_recognize (
  partition by device
  measures A.id as a_id
  pattern (A)
  define
    A as A.temp > 100 and prev(A.temp, 2) > 100)
```

An example sequence of events that matches the pattern above is:

Table 6.11. Example

Arrival Time	Tuple	Output Event (if any)
1000	id=E1, device=1, temp=98	
2000	id=E2, device=1, temp=101	
3000	id=E3, device=1, temp=101	
4000	id=E4, device=1, temp=99	
5000	id=E5, device=1, temp=101	a_id=E5

6.6. Measure Clause

The `measures` clause defines exported columns that contain expressions over the pattern variables. The expressions can reference partition columns, singleton variables and any aggregation functions including `last` and `first` on the group variables.

Expressions in the `measures` clause must use the `as` keyword to assign a column name.

If a variable is a singleton variable then only individual columns may be referenced, not aggregates. If the variable is not matched by an event, a `null` value is returned.

If a variable is a group variable and used in an aggregate, then the aggregate is performed over all rows that have matched the variable. If a group variable is not in an aggregate function, its variable name must be post-fixed with an index. See Section 6.4.5, “Variables Can be Singleton or Group” for more information.

6.7. Datawindow-Bound

When using match recognize with a named window or stream bound by a data window, all events removed from the named window or data window also removed the match-in-progress that includes the event(s) removed.

The next example looks for four sensor events from the same device immediately following each other and indicating a rising temperature, but only events that arrived in the last 10 seconds are considered:

```
select * from TemperatureSensorEvent.win:time(10 sec)
match_recognize (
  partition by device
  measures A.id as a_id
  pattern (A B C D)
  define
    B as B.temp > A.temp,
    C as C.temp > B.temp,
    D as D.temp > C.temp)
```

An example sequence of events that matches the pattern above is:

Table 6.12. Example

Arrival Time	Tuple	Output Event (if any)
1000	id=E1, device=1, temp=80	
2000	id=E2, device=1, temp=81	
3000	id=E3, device=1, temp=82	
4000	id=E4, device=1, temp=81	
7000	id=E5, device=1, temp=82	
9000	id=E6, device=1, temp=83	
13000	id=E7, device=1, temp=84	a_id=E4, a_id=E5, a_id=E6, a_id=E7
15000	id=E8, device=1, temp=84	
20000	id=E9, device=1, temp=85	
21000	id=E10, device=1, temp=86	
26000	id=E11, device=1, temp=87	

6.8. Interval

With the optional `interval` keyword and time period you can control how long the engine should wait for further events to arrive that may be part of a matching event sequence, before indicating a match (or matches). This is not applicable to on-demand pattern matching.

The interval timer starts at the arrival of the first event matching a sequence for a partition. When the time interval passes and an event sequence matches, duplicate matches are eliminated and output occurs.

The next example looks for sensor events indicating a temperature of over 100 waiting for any number of additional events with a temperature of over 100 for 10 seconds before indicating a match:

```
select * from TemperatureSensorEvent
```

```

match_recognize (
partition by device
measures A.id as a_id, count(B.id) as count_b, first(B.id) as first_b, last(B.id) as last_b
pattern (A B*)
interval 5 seconds
define
  A as A.temp > 100,
  B as B.temp > 100)

```

An example sequence of events that matches the pattern above is:

Table 6.13. Example

Arrival Time	Tuple	Output Event (if any)
1000	id=E1, device=1, temp=98	
2000	id=E2, device=1, temp=101	
3000	id=E3, device=1, temp=102	
4000	id=E4, device=1, temp=104	
5000	id=E5, device=1, temp=104	
7000		a_id=E2, count_b=3, first_b=E3, last_b=E5

Notice that the engine waits 5 seconds (5000 milliseconds) after the arrival time of the first event E2 of the match at 2000, to indicate the match at 7000.

6.9. Limitations

Please note the following limitations:

1. Subqueries are not allowed in expressions within `match_recognize`.
2. Joins and outer joins are not allowed in the same statement as `match_recognize`.
3. `match_recognize` may not be used within `on-select` or `on-insert` statements.
4. When using `match_recognize` on unbound streams (no data window provided) the `iterator pull` API returns no rows.
5. A Statement Object Model API for `match_recognize` is not yet available.

Chapter 7. EPL Reference: Operators

Esper arithmetic and logical operator precedence follows standard arithmetic and logical operator precedence.

7.1. Arithmetic Operators

The below table outlines the arithmetic operators available.

Table 7.1. Syntax and results of arithmetic operators

Operator	Description
+, -	As unary operators they denote a positive or negative expression. As binary operators they add or subtract.
*, /	Multiplication and division are binary operators.
%	Modulo binary operator.

7.2. Logical And Comparison Operators

The below table outlines the logical and comparison operators available.

Table 7.2. Syntax and results of logical and comparison operators

Operator	Description
NOT	Returns true if the following condition is false, returns false if it is true.
OR	Returns true if either component condition is true, returns false if both are false.
AND	Returns true if both component conditions are true, returns false if either is false.
=, !=, <, > <=, >=,	Comparison.

7.3. Concatenation Operators

The below table outlines the concatenation operators available.

Table 7.3. Syntax and results of concatenation operators

Operator	Description
	Concatenates character strings

7.4. Binary Operators

The below table outlines the binary operators available.

Table 7.4. Syntax and results of binary operators

Operator	Description
&	Bitwise AND if both operands are numbers; conditional AND if both operands are boolean.
	Bitwise OR if both operands are numbers; conditional OR if both operands are boolean.
^	Bitwise exclusive OR (XOR).

7.5. Array Definition Operator

The { and } curly braces are array definition operators following the array initialization syntax. Arrays can be useful to pass to user-defined functions or to select array data in a select clause.

Array definitions consist of zero or more expressions within curly braces. Any type of expression is allowed within array definitions including constants, arithmetic expressions or event properties. This is the syntax of an array definition:

```
{ [expression [,expression...]] }
```

Consider the next statement that returns an event property named `actions`. The engine populates the `actions` property as an array of `System.String` values with a length of 2 elements. The first element of the array contains the `observation` property value and the second element the `command` property value of `RFIDEvent` events.

```
select {observation, command} as actions from RFIDEvent
```

The engine determines the array type based on the types returned by the expressions in the array definition. For example, if all expressions in the array definition return integer values then the type of the array is `System.Int32[]`. If the types returned by all expressions are compatible number types, such as integer and double values, the engine coerces the array element values and returns a suitable type, `System.Double[]` in this example. The type of the array returned is `Object[]` if the types of expressions cannot be coerced or return object values. Null values can also be used in an array definition.

Arrays can come in handy for use as parameters to user-defined functions:

```
select * from RFIDEvent where Filter.myFilter(zone, {1,2,3})
```

7.6. Dot Operator

You can use the dot operator to invoke a method on the result of an expression. The dot operator uses the dot (.) or period character.

The synopsis for the dot operator is as follows

```
(expression).method([parameter [...]] [.method(...)] [...])
```

The expression to evaluate by the dot operator is in parenthesis. After the dot character follows the method name and method parameters in parenthesis.

You may use the dot operator when your expression returns an object that you want to invoke a method on. The dot operator allows duck typing and convenient array and collection access methods.

This example statement invokes the `GetZones` method of the RFID event class by referring to the stream name assigned in the `from`-clause:

```
select (rfid).GetZones() from RFIDEvent as rfid
```

The `size()` method can be used to return the array length or collection size. Use the `Get` method to return the value at a given index for an array or collection.

The next statement selects array size and returns the last array element:

```
select (arrayproperty).size() as arraySize,  
       (arrayproperty).get((arrayproperty).size - 1) as lastInArray  
from ProductEvent
```

7.6.1. Duck Typing

Duck typing is when the engine checks at runtime for the existence of a method regardless of object class inheritance hierarchies. This can be useful, for example, when a dynamic property returns an object which may or may not provide a method to return the desired value.

Duck typing is disabled in the default configuration to consistently enforce strong typing. Please enable duck typing via engine expression settings as described in Section 13.4.18, “Engine Settings related to Expression Evaluation”.

The statement below selects a dynamic property by name `productDesc` and invokes the `getCounter()` method if that method exists on the property value, or returns the null value if the method does not exist for the dynamic property value or if the dynamic property value itself is null:

```
select (productDesc?).getCounter() as arraySize from ProductEvent
```

7.7. The 'in' Keyword

The `in` keyword determines if a given value matches any value in a list. The syntax of the keyword is:

```
test_expression [not] in (expression [,expression...] )
```

The *test_expression* is any valid expression. The keyword is followed by a list of expressions to test for a match. The optional `not` keyword specifies that the result of the predicate be negated.

The result of an `in` expression is of type `Boolean`. If the value of *test_expression* is equal to any expression from the comma-separated list, the result value is `true`. Otherwise, the result value is `false`.

The next example shows how the `in` keyword can be applied to select certain command types of RFID events:

```
select * from RFIDEvent where command in ('OBSERVATION', 'SIGNAL')
```

The statement is equivalent to:

```
select * from RFIDEvent where command = 'OBSERVATION' or command = 'SIGNAL'
```

Expression may also return an array, a `System.Collection.Generic.ICollection` or a `System.Collection.Generic.IDictionary`. Thus event properties that are lists, sets or maps may provide values to compare against *test_expression*.

All expressions must be of the same type or a compatible type to *test_expression*. The `in` keyword may coerce number values to compatible types. If *expression* returns an array, then the component type of the array must be compatible, unless the component type of the array is `Object`.

If *expression* returns an array of component type `Object`, the operation compares each element of the array, applying `equals` semantics.

If *expression* returns a `Collection`, the operation determines if the collection contains the value returned by *test_expression*, applying `contains` semantics.

If *expression* returns a `Map`, the operation determines if the map contains the key value returned by *test_expression*, applying `containsKey` semantics.

Constants, arrays, `Collection` and `Map` expressions or event properties can be used combined.

For example, and assuming a property named `'mySpecialCmdList'` exists that contains a list of command strings:

```
select * from RFIDEvent where command in ( 'OBSERVATION', 'SIGNAL', mySpecialCmdList)
```

When using prepared statements and substitution parameters with the `in` keyword, make sure to retain the parenthesis. Substitution values may also be arrays, `Collection` and `Map` values:

```
test_expression [not] in ( ? [,?...] )
```

Note that if there are no successes and at least one right-hand row yields null for the operator's result, the result of the any construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

7.8. The 'between' Keyword

The `between` keyword specifies a range to test. The syntax of the keyword is:

```
test_expression [not] between begin_expression and end_expression
```

The *test_expression* is any valid expression and is the expression to test for in the range defined by *begin_expression* and *end_expression*. The `not` keyword specifies that the result of the predicate be negated.

The result of a `between` expression is of type `Boolean`. If the value of *test_expression* is greater than or equal to the value of *begin_expression* and less than or equal to the value of *end_expression*, the result is `true`.

The next example shows how the `between` keyword can be used to select events with a price between 55 and 60 (inclusive).

```
select * from StockTickEvent where price between 55 and 60
```

The equivalent expression without `between` is:

```
select * from StockTickEvent where price >= 55 and price <= 60
```

And also equivalent to:

```
select * from StockTickEvent where price between 60 and 55
```

7.9. The 'like' Keyword

The `like` keyword provides standard SQL pattern matching. SQL pattern matching allows you to use `'_'` to match any single character and `'%'` to match an arbitrary number of characters (including zero characters). In Esper, SQL patterns are case-sensitive by default. The syntax of `like` is:

```
test_expression [not] like pattern_expression [escape string_literal]
```

The *test_expression* is any valid expression yielding a `String`-type or a numeric result. The optional `not` keyword specifies that the result of the predicate be negated. The `like` keyword is followed by any valid standard SQL *pattern_expression* yielding a `String`-typed result. The optional `escape` keyword signals the escape character to escape `'_'` and `'%'` values in the pattern.

The result of a `like` expression is of type `Boolean`. If the value of *test_expression* matches the *pattern_expression*, the result value is `true`. Otherwise, the result value is `false`.

An example for the `like` keyword is below.

```
select * from PersonLocationEvent where name like '%Jack%'
```

The escape character can be defined as follows. In this example the where-clause matches events where the suffix property is a single `'_'` character.

```
select * from PersonLocationEvent where suffix like '!_' escape '!'
```

7.10. The 'regexp' Keyword

The `regexp` keyword is a form of pattern matching based on regular expressions implemented through the `sys-`

`tem.Text.RegularExpression` provider. The syntax of `regexp` is:

```
test_expression [not] regexp pattern_expression
```

The *test_expression* is any valid expression yielding a `String`-type or a numeric result. The optional `not` keyword specifies that the result of the predicate be negated. The `regexp` keyword is followed by any valid regular expression *pattern_expression* yielding a `String`-typed result.

The result of a `regexp` expression is of type `Boolean`. If the value of *test_expression* matches the regular expression *pattern_expression*, the result value is `true`. Otherwise, the result value is `false`.

An example for the `regexp` keyword is below.

```
select * from PersonLocationEvent where name regexp '*Jack*'
```

7.11. The 'any' and 'some' Keywords

The `any` operator is true if the expression returns true for one or more of the values returned by a list of expressions including array, `Collection` and `Map` values.

The synopsis for the `any` keyword is as follows:

```
expression operator any (expression [,expression...] )
```

The left-hand expression is evaluated and compared to each expression result using the given operator, which must yield a `Boolean` result. The result of `any` is "true" if any true result is obtained. The result is "false" if no true result is found (including the special case where the expressions are collections that return no rows).

The *operator* can be any of the following values: `=`, `!=`, `<>`, `<`, `<=`, `>`, `>=`.

The `some` keyword is a synonym for `any`. The `in` construct is equivalent to `= any`.

Expression may also return an array, a `System.Collections.Generic.ICollection` or a `System.Collections.Generic.IDictionary`. Thus event properties that are lists, sets or maps may provide values to compare against.

All expressions must be of the same type or a compatible type. The `any` keyword coerces number values to compatible types. If *expression* returns an array, then the component type of the array must be compatible, unless the component type of the array is `Object`.

If *expression* returns an array, the operation compares each element of the array.

If *expression* returns a `Collection`, the operation determines if the collection contains the value returned by the left-hand expression, applying `contains` semantics. When using relationship operators `<`, `<=`, `>`, `>=` the operator applies to each element in the collection, and non-numeric elements are ignored.

If *expression* returns a `Map`, the operation determines if the map contains the key value returned by the left-hand expression, applying `containsKey` semantics. When using relationship operators `<`, `<=`, `>`, `>=` the operator applies to each key in the map, and non-numeric map keys are ignored.

Constants, arrays, `Collection` and `Map` expressions or event properties can be used combined.

The next statement demonstrates the use of the `any` operator:

```
select * from ProductOrder where category != any (categoryArray)
```

The above query selects ProductOrder event that have a category field and a category array, and returns only those events in which the category value is not in the array.

Note that if there are no successes and at least one right-hand row yields null for the operator's result, the result of the `any` construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

7.12. The 'all' Keyword

The `all` operator is true if the expression returns true for all of the values returned by a list of expressions including array, `Collection` and `Map` values.

The synopsis for the `all` keyword is as follows:

```
expression operator all (expression [,expression...] )
```

The left-hand expression is evaluated and compared to each expression result using the given operator, which must yield a Boolean result. The result of `all` is "true" if all rows yield true (including the special case where the expressions are collections that returns no rows). The result is "false" if any false result is found. The result is null if the comparison does not return false for any row, and it returns null for at least one row.

The *operator* can be any of the following values: `=`, `!=`, `<>`, `<`, `<=`, `>`, `>=`.

The `not in` construct is equivalent to `!= all`.

Expression may also return an array, a `System.Collection.Generic.ICollection` or a `System.Collection.Generic.IDictionary`. Thus event properties that are lists, sets or maps may provide values to compare against.

All expressions must be of the same type or a compatible type. The `all` keyword coerces number values to compatible types. If *expression* returns an array, then the component type of the array must be compatible, unless the component type of the array is `Object`.

If *expression* returns an array, the operation compares each element of the array.

If *expression* returns a `Collection`, the operation determines if the collection contains the value returned by the left-hand expression, applying `contains` semantics. When using relationship operators `<`, `<=`, `>`, `>=` the operator applies to each element in the collection, and non-numeric elements are ignored.

If *expression* returns a `Map`, the operation determines if the map contains the key value returned by the left-hand expression, applying `containsKey` semantics. When using relationship operators `<`, `<=`, `>`, `>=` the operator applies to each key in the map, and non-numeric map keys are ignored.

Constants, arrays, `Collection` and `Map` expressions or event properties can be used combined.

The next statement demonstrates the use of the `all` operator:

```
select * from ProductOrder where category = all (categoryArray)
```

The above query selects ProductOrder event that have a category field and a category array, and returns only those events in which the category value matches all values in the array.

Chapter 8. EPL Reference: Functions

8.1. Single-row Function Reference

Single-row functions return a single value for every single result row generated by your statement. These functions can appear anywhere where expressions are allowed.

Esper allows static library methods as single-row functions, and also features built-in single-row functions. In addition, Esper allows instance method invocations on named streams.

You may also register your own single-row function name with the engine so that your EPL statements become less cluttered. This is described in detail in Section 15.3, “Single-Row Functions”. Single-row functions that return an object can be chained.

Esper auto-imports the following library namespaces:

- System
- System.Collections
- System.Text

Thus static library methods can be used in all expressions as shown in below example:

```
select symbol, Math.Round(volume/1000)
from StockTickEvent.win:time(30 sec)
```

In general, arbitrary type names have to be fully qualified (e.g. System.Math) but Esper provides a mechanism for user-controlled imports of classes and packages as outlined in Section 13.4.5, “Class and package imports”.

The below table outlines the built-in single-row functions available.

Table 8.1. Syntax and results of single-row functions

Single-row Function	Result
<pre>case value when compare_value then result [when compare_value then result ...] [else result] end</pre>	Returns result where the first value equals compare_value.
<pre>case when condition then result [when condition then result ...] [else result] end</pre>	Returns the result for the first condition that is true.
<pre>cast(expression, type_name)</pre>	Casts the result of an expression to the given type.
<pre>coalesce(expression, expression [, expression ...])</pre>	Returns the first non-null value in the list, or null if there are no non-null values.

Single-row Function	Result
<code>current_timestamp[()]</code>	Returns the current engine time as a <code>long</code> millisecond value. Reserved keyword with optional parenthesis.
<code>exists(dynamic_property_name)</code>	Returns true if the dynamic property exists for the event, or false if the property does not exist.
<code>instanceof(expression, type_name [, type_name ...])</code>	Returns true if the expression returns an object whose type is one of the types listed.
<code>max(expression, expression [, expression ...])</code>	Returns the highest numeric value among the 2 or more comma-separated expressions.
<code>min(expression, expression [, expression ...])</code>	Returns the lowest numeric value among the 2 or more comma-separated expressions.
<code>prev(expression, event_property)</code>	Returns a property value or all properties of a previous event, relative to the event order within a data window, or according to an optional index parameter (N) the positional Nth-from-last value.
<code>prevtail(expression, event_property)</code>	Returns a property value or all properties of the first event in a data window relative to the event order within a data window, or according to an optional index parameter (N) the positional Nth-from-first value.
<code>prevwindow(event_property)</code>	Returns a single property value of all events or all properties of all events in a data window in the order that reflects the sort order of the data window.
<code>prevcount(event_property)</code>	Returns the count of events (number of data points) in a data window.
<code>prior(integer, event_property)</code>	Returns a property value of a prior event, relative to the natural order of arrival of events
<code>typeof(expression)</code>	If expression is a stream name, returns the event type name of the evaluated event, often used with variant streams. If expression is a property name or expression, returns the name of the expression result type.

8.1.1. The `case` Control Flow Function

The `case` control flow function has two versions. The first version takes a value and a list of compare values to compare against, and returns the result where the first value equals the compare value. The second version takes a list of conditions and returns the result for the first condition that is true.

The return type of a `case` expression is the compatible aggregated type of all return values.

The example below shows the first version of a `case` statement. It has a `String` return type and returns the value 'one'.

```
select case 1 when 1 then 'one' when 2 then 'two' else 'more' end from ...
```

The second version of the `case` function takes a list of conditions. The next example has a `Boolean` return type and returns the boolean value true.

```
select case when 1>0 then true else false end from ...
```

8.1.2. The `cast` Function

The `cast` function casts the return type of an expression to a designated type. The function accepts two parameters: The first parameter is the property name or expression that returns the value to be casted. The second parameter is the type to cast to.

Valid parameters for the second (type) parameter are:

- Any of the built-in types: `int`, `long`, `ulong`, `byte`, `sbyte`, `short`, `ushort`, `char`, `decimal`, `double`, `float`, `string`, where `string` is a short notation for `System.String`. The type name is not case-sensitive. For example:

```
cast(price, double)
```

- The fully-qualified type name of the class to cast to, for example:

```
cast(product, org.myproducer.Product)
```

The `cast` function is often used to provide a type for dynamic (unchecked) properties. Dynamic properties are properties whose type is not known at compile type. These properties are always of type `System.Object`.

The `cast` function as shown in the next statement casts the dynamic "price" property of an "item" in the `OrderEvent` to a double value.

```
select cast(item.price?, double) from OrderEvent
```

The `cast` function returns a `null` value if the expression result cannot be casted to the desired type, or if the expression result itself is `null`.

The `cast` function adheres to the following type conversion rules:

- For all numeric types, the `cast` function utilizes an internal cast conversion mechanism to convert numeric types, if required.

- For casts to `string` or `System.String`, the function calls `System.Convert.ToString` on the expression result.
- For casts to other objects including application objects, the `cast` function considers a type's superclasses as well as all directly or indirectly-implemented interfaces by superclasses .

8.1.3. The `Coalesce` Function

The result of the `coalesce` function is the first expression in a list of expressions that returns a non-null value. The return type is the compatible aggregated type of all return values.

This example returns a `String`-typed result of value 'foo':

```
select coalesce(null, 'foo') from ...
```

8.1.4. The `Current_Timestamp` Function

The `current_timestamp` function is a reserved keyword and requires no parameters. The result of the `current_timestamp` function is the long-type millisecond value of the current engine system time.

The function returns the current engine timestamp at the time of expression evaluation. When using external-timer events, the function provides the last value of the externally-supplied time at the time of expression evaluation.

This example selects the current engine time:

```
select current_timestamp from MyEvent
// equivalent to
select current_timestamp() from MyEvent
```

8.1.5. The `Exists` Function

The `exists` function returns a boolean value indicating whether the dynamic property, provided as a parameter to the function, exists on the event. The `exists` function accepts a single dynamic property name as its only parameter.

The `exists` function is for use with dynamic (unchecked) properties. Dynamic properties are properties whose type is not known at compile type. Dynamic properties return a null value if the dynamic property does not exist on an event, or if the dynamic property exists but the value of the dynamic property is null.

The `exists` function as shown next returns true if the "item" property contains an object that has a "serviceName" property. It returns false if the "item" property is null, or if the "item" property does not contain an object that has a property named "serviceName" :

```
select exists(item.serviceName?) from OrderEvent
```

8.1.6. The `Instance-Of` Function

The `instanceof` function returns a boolean value indicating whether the type of value returned by the expression is one of the given types. The first parameter to the `instanceof` function is an expression to evaluate. The second and subsequent parameters are type names.

The function determines the return type of the expression at runtime by evaluating the expression, and compares the type of object returned by the expression to the defined types. If the type of object returned by the expression matches any of the given types, the function returns `true`. If the expression returned `null` or a type that does not match any of the given types, the function returns `false`.

The `instanceof` function is often used in conjunction with dynamic (unchecked) properties. Dynamic properties are properties whose type is not known at compile type.

This example uses the `instanceof` function to select different properties based on the type:

```
select case when instanceof(item, com.mycompany.Service) then serviceName?
        when instanceof(item, com.mycompany.Product) then productName? end
from OrderEvent
```

The `instanceof` function returns `false` if the expression tested by `instanceof` returned `null`.

Valid parameters for the type parameter list are:

- Any of the built-in types: `int`, `long`, `byte`, `short`, `char`, `double`, `float`, `string`, where `string` is a short notation for `System.String`. The type name is not case-sensitive. For example, the next function tests if the dynamic "price" property is either of type `float` or type `double`:

```
instanceof(price?, double, float)
```

- The fully-qualified class name of the class to cast to, for example:

```
instanceof(product, org.myproducer.Product)
```

The function considers an event class's superclasses as well as all the directly or indirectly-implemented interfaces by superclasses.

8.1.7. The `min` and `max` Functions

The `min` and `max` function take two or more parameters that itself can be expressions. The `min` function returns the lowest numeric value among the 2 or more comma-separated expressions, while the `max` function returns the highest numeric value. The return type is the compatible aggregated type of all return values.

The next example shows the `max` function that has a `Double` return type and returns the value 1.1.

```
select max(1, 1.1, 2 * 0.5) from ...
```

The `min` function returns the lowest value. The statement below uses the function to determine the smaller of two timestamp values.

```
select symbol, min(ticks.timestamp, news.timestamp) as minT
    from StockTickEvent.win:time(30 sec) as ticks, NewsEvent.win:time(30 sec) as news
    where ticks.symbol = news.symbol
```

8.1.8. The `Previous` Function

The `prev` function returns the property value or all event properties of a previous event. For data windows that introduce a sort order other than the order of arrival, such as the sorted data window and the time order data window, the function returns the event at the specified position.

The `prev` function is not an aggregation function and therefore does not return results per group when used with `group by`. Please consider the `last`, `lastever` or `nth` aggregation functions instead as described in Section 8.2.2, “Data Window Aggregation Functions”. You must use an aggregation function instead of `prev` when querying a named window.

The first parameter to the `prev` function is an index parameter and denotes the *i*-th previous event, in the order established by the data window. If no index is provided, the default index is 1 and the function returns the previous event. The second parameter is a property name or stream name. If specifying a property name, the function returns the value for the previous event property value. If specifying a stream name, the function returns the previous event underlying object.

This example selects the value of the `price` property of the 2nd-previous event from the current Trade event:

```
select prev(2, price) from Trade.win:length(10)
```

By using the stream alias in the `previous` function, the next example selects the trade event itself that is immediately previous to the current Trade event

```
select prev(1, trade) from Trade.win:length(10) as trade
```

Since the `prev` function takes the order established by the data window into account, the function works well with sorted windows.

In the following example the statement selects the symbol of the 3 Trade events that had the largest, second-largest and third-largest volume.

```
select prev(0, symbol), prev(1, symbol), prev(2, symbol)
from Trade.ext:sort(3, volume desc)
```

The *i*-th previous event parameter can also be an expression returning an Integer-type value. The next statement joins the Trade data window with an `RankSelectionEvent` event that provides a `rank` property used to look up a certain position in the sorted Trade data window:

```
select prev(rank, symbol) from Trade.ext:sort(10, volume desc), RankSelectionEvent unidirectional
```

The `prev` function returns a `null` value if the data window does not currently hold the *i*-th previous event. The example below illustrates this using a time batch window. Here the `prev` function returns a `null` value for any events in which the previous event is not in the same batch of events. Note that the `prior` function as discussed below can be used if a `null` value is not the desired result.

```
select prev(1, symbol) from Trade.win:time_batch(1 min)
```

An alternative form of the `prev` function allows the index to not appear or appear after the property name if the index value is a constant and not an expression:

```
select prev(1, symbol) from Trade
// ... equivalent to ...
select prev(symbol) from Trade
// ... and ...
select prev(symbol, 1) from Trade
```

The combination of the `prev` function and `std:groupwin` view returns the property value for a previous event in the given data window group.

The following example returns for each event the current smallest price per symbol:

```
select symbol, prev(0, price) as topPricePerSymbol
from Trade.std:groupwin(symbol).ext:sort(1, price asc)
```

Restrictions

The following restrictions apply to the `prev` functions and its results:

- The function always returns a `null` value for remove stream (old data) events.
- The function requires a data window view, or a `std:groupwin` and data window view, without any additional sub-views. See Chapter 11, *EPL Reference: Views* for built-in data window views.

Comparison to the `prior` Function

The `prev` function is similar to the `prior` function. The key differences between the two functions are as follows:

- The `prev` function returns previous events in the order provided by the data window, while the `prior` function returns prior events in the order of arrival as posted by a stream's declared views.
- The `prev` function requires a data window view while the `prior` function does not have any view requirements.
- The `prev` function returns the previous event grouped by a criteria by combining the `std:groupwin` view and a data window. The `prior` function returns prior events posted by the last view regardless of data window grouping.
- The `prev` function returns a `null` value for remove stream events, i.e. for events leaving a data window. The `prior` function does not have this restriction.

8.1.9. The `prevtail` Function

The `prevtail` function returns the property value or all event properties of the positional-first event in a data window. For data windows that introduce a sort order other than the order of arrival, such as the sorted data window and the time order data window, the function returns the first event at the specified position.

The `prevtail` function is not an aggregation function and therefore does not return results per group when used with `group by`. Please consider the `first`, `firstever` or window aggregation functions instead as described in Section 8.2.2, “Data Window Aggregation Functions”. You must use an aggregation function instead of `prevtail` when querying a named window.

The first parameter is an index parameter and denotes the *i*-th from-first event in the order established by the data window. If no index is provided the default is zero and the function returns the first event in the data window. The second parameter is a property name or stream name. If specifying a property name, the function returns the value for the previous event property value. If specifying a stream name, the function returns the previous event underlying object.

This example selects the value of the `price` property of the first (oldest) event held in the length window:

```
select prevtail(price) from Trade.win:length(10)
```

By using the stream alias in the `prevtail` function, the next example selects the trade event itself that is the second event held in the length window:

```
select prevtail(1, trade) from Trade.win:length(10) as trade
```

Since the `prevtail` function takes the order established by the data window into account, the function works

well with sorted windows.

In the following example the statement selects the symbol of the 3 Trade events that had the smallest, second-smallest and third-smallest volume.

```
select prevtail(0, symbol), prevtail(1, symbol), prevtail(2, symbol)
from Trade.ext:sort(3, volume asc)
```

The *i*-th previous event parameter can also be an expression returning an Integer-type value. The next statement joins the Trade data window with an `RankSelectionEvent` event that provides a `rank` property used to look up a certain position in the sorted Trade data window:

```
select prevtail(rank, symbol) from Trade.ext:sort(10, volume asc), RankSelectionEvent unidirectional
```

The `prev` function returns a null value if the data window does not currently holds positional-first or the Nth-from-first event. For batch data windows the value returned is relative to the current batch.

The following example returns the first and second symbol value in the batch:

```
select prevtail(0, symbol), prevtail(1, symbol) from Trade.win:time_batch(1 min)
```

An alternative form of the `prevtail` function allows the index to not appear or appear after the property name if the index value is a constant and not an expression:

```
select prevtail(1, symbol) from Trade
// ... equivalent to ...
select prevtail(symbol) from Trade
// ... and ...
select prevtail(symbol, 1) from Trade
```

The combination of the `prevtail` function and `std:groupwin` view returns the property value for a positional first event in the given data window group.

Let's look at an example. This statement outputs the oldest price per symbol retaining the last 10 prices per symbol:

```
select symbol, prevtail(0, price) as oldestPrice
from Trade.std:groupwin(symbol).win:length(10)
```

Restrictions

The following restrictions apply to the `prev` functions and its results:

- The function always returns a null value for remove stream (old data) events.
- The function requires a data window view, or a `std:groupwin` and data window view, without any additional sub-views. See Chapter 11, *EPL Reference: Views* for built-in data window views.

8.1.10. The Previous-Window Function

The `prevwindow` function returns property values or all event properties for all events in a data window. For data windows that introduce a sort order other than the order of arrival, such as the sorted data window and the time order data window, the function returns the event data sorted in that order, otherwise it returns the events sorted by order of arrival with the newest arriving event first.

The `prevwindow` function is not an aggregation function and therefore does not return results per group when used with `group by`. Please consider the `window` aggregation function instead as described in Section 8.2.2,

“Data Window Aggregation Functions”. You must use an aggregation function instead of `prevwindow` when querying a named window.

The single parameter is a property name or stream name. If specifying a property name, the function returns the value of the event property for all events held by the data window. If specifying a stream name, the function returns the event underlying object for all events held by the data window.

This example selects the value of the `price` property of all events held in the length window:

```
select prevwindow(price) from Trade.win:length(10)
```

By using the stream alias in the `prevwindow` function, the next example selects all trade events held in the length window:

```
select prevwindow(trade) from Trade.win:length(10) as trade
```

When used with a data window that introduces a certain sort order, the `prevwindow` function returns events sorted according to that sort order.

The next statement outputs for every arriving event the current 10 underlying trade event objects that have the largest volume:

```
select prevwindow(trade) from Trade.ext:sort(10, volume desc) as trade
```

The `prevwindow` function returns a `null` value if the data window does not currently hold any events.

The combination of the `prevwindow` function and `std:groupwin` view returns the property value(s) for all events in the given data window group.

This example statement outputs all prices per symbol retaining the last 10 prices per symbol:

```
select symbol, prevwindow(price) from Trade.std:groupwin(symbol).win:length(10)
```

Restrictions

The following restrictions apply to the `prev` functions and its results:

- The function always returns a `null` value for remove stream (old data) events.
- The function requires a data window view, or a `std:groupwin` and data window view, without any additional sub-views. See Chapter 11, *EPL Reference: Views* for built-in data window views.

8.1.11. The `Previous-Count` Function

The `prevcount` function returns the number of events held in a data window.

The `prevcount` function is not an aggregation function and therefore does not return results per group when used with `group by`. Please consider the `count(*)` aggregation function instead as described in Section 8.2, “Aggregate Functions”. You must use an aggregation function instead of `prevcount` when querying a named window.

The single parameter is a property name or stream name of the data window to return the count for.

This example selects the number of data points for the `price` property held in the length window:

```
select prevcount(price) from Trade.win:length(10)
```

By using the stream alias in the `prevcount` function the next example selects the count of trade events held in the length window:

```
select prevcount(trade) from Trade.win:length(10) as trade
```

The combination of the `prevcount` function and `std:groupwin` view returns the count of events in the given data window group.

This example statement outputs the number of events retaining the last 10 events per symbol:

```
select symbol, prevcount(price) from Trade.std:groupwin(symbol).win:length(10)
```

Restrictions

The following restrictions apply to the `prev` functions and its results:

- The function always returns a `null` value for remove stream (old data) events.
- The function requires a data window view, or a `std:groupwin` and data window view, without any additional sub-views. See Chapter 11, *EPL Reference: Views* for built-in data window views.

8.1.12. The `Prior` Function

The `prior` function returns the property value of a prior event. The first parameter is an integer value that denotes the *i*-th prior event in the natural order of arrival. The second parameter is a property name for which the function returns the value for the prior event. The second parameter is a property name or stream name. If specifying a property name, the function returns the property value for the prior event. If specifying a stream name, the function returns the prior event underlying object.

This example selects the value of the `price` property of the 2nd-prior event to the current Trade event.

```
select prior(2, price) from Trade
```

By using the stream alias in the `prior` function, the next example selects the trade event itself that is immediately prior to the current Trade event

```
select prior(1, trade) from Trade as trade
```

The `prior` function can be used on any event stream or view and does not have any specific view requirements. The function operates on the order of arrival of events by the event stream or view that provides the events.

The next statement uses a time batch window to compute an average volume for 1 minute of Trade events, posting results every minute. The select-clause employs the `prior` function to select the current average and the average before the current average:

```
select average, prior(1, average)
  from TradeAverages.win:time_batch(1 min).stat:uni(volume)
```

8.1.13. The `Type-Of` Function

The `typeof` function, when parameterized by a stream name, returns the event type name of the evaluated event which can be useful with variant streams. When parameterized by an expression or property name, the function returns the type name of the expression result or `null` if the expression result is null.

In summary, the function determines the return type of the expression at runtime by evaluating the expression and returns the type name of the expression result.

The `typeof` function is often used in conjunction with variant streams. A variant stream is a predefined stream into which events of multiple disparate event types can be inserted. The `typeof` function, when passed a stream name alias, returns the name of the event type of each event in the stream.

The following example elaborates on the use of variant streams with `typeof`. The first statement declares a variant stream `SequencePatternStream`:

```
create variant schema SequencePatternStream as *
```

The next statement inserts all order events and is followed by a statement to insert all product events:

```
insert into SequencePatternStream select * from OrderEvent;
```

```
insert into SequencePatternStream select * from PriceEvent;
```

This example statement returns the event type name for each event in the variant stream:

```
select typeof(sps) from SequencePatternStream as sps
```

The next example statement detects a pattern by utilizing the `typeof` function to find pairs of order event immediately followed by product event:

```
select * from SequencePatternStream match_recognize(
  measures A as a, B as b
  pattern (A B)
  define A as typeof(A) = "OrderEvent",
         B as typeof(B) = "ProductEvent"
)
```

When passing a property name to the `typeof` function, the function evaluates whether the property type is event type (a fragment event type). If the property type is event type, the function returns the type name of the event in the property value or `null` if not provided. If the property type is not event type, the function returns the simple class name of the property value.

When passing an expression to the `typeof` function, the function evaluates the expression and returns the simple class name of the expression result value or `null` if the expression result value is `null`.

This example statement returns the simple class name of the value of the dynamic property `prop` of events in stream `MyStream`, or a `null` value if the property is not found for an event or the property value itself is `null`:

```
select typeof(prop?) from MyStream
```

When using subclasses or interface implementations as event classes or when using Map-event type inheritance, the function returns the event type name provided when the class or Map-type event was registered, or if the event type was not registered, the function returns the fully-qualified class name.

8.2. Aggregate Functions

Aggregation functions return a single value from a collection of input values. The `group by` keywords are often used in conjunction with aggregation functions to group the result-set by one or more columns.

The EPL language extends the standard SQL aggregation functions by useful aggregation functions that can track a data window or compute event rates. Your application may also add its own aggregation function as Section 15.5, “Aggregation Functions” describes.

Aggregation values are always computed incrementally: Insert and remove streams result in aggregation value changes. The exceptions are on-demand queries and joins when using the `unidirectional` keyword. Aggregation functions are optimized to retain the minimal information necessary to compute the aggregated result.

Aggregation functions can also be used with unbound streams when no data window is specified.

8.2.1. SQL-Standard Functions

The SQL-standard aggregation functions are shown in below table.

Table 8.2. Syntax and results of SQL-standard aggregation functions

Aggregate Function	Result
<code>avedev([all distinct] <i>expression</i>)</code>	Mean deviation of the (distinct) values in the expression, returning a value of <code>double</code> type.
<code>avg([all distinct] <i>expression</i>)</code>	Average of the (distinct) values in the expression, returning a value of <code>double</code> type.
<code>count([all distinct] <i>expression</i>)</code>	Number of the (distinct) non-null values in the expression, returning a value of <code>long</code> type.
<code>count(*)</code>	Number of events, returning a value of <code>long</code> type.
<code>max([all distinct] <i>expression</i>)</code>	Highest (distinct) value in the expression, returning a value of the same type as the expression itself returns.
<code>median([all distinct] <i>expression</i>)</code>	Median (distinct) value in the expression, returning a value of <code>double</code> type. Double Not-a-Number (NaN) values are ignored in the median computation.
<code>min([all distinct] <i>expression</i>)</code>	Lowest (distinct) value in the expression, returning a value of the same type as the expression itself returns.
<code>stddev([all distinct] <i>expression</i>)</code>	Standard deviation of the (distinct) values in the expression, returning a value of <code>double</code> type.
<code>sum([all distinct] <i>expression</i>)</code>	Totals the (distinct) values in the expression, returning a value of <code>long</code> , <code>double</code> , <code>float</code> or <code>integer</code> type depending on the expression.

If your application provides double-type values to an aggregation function, avoid using Not-a-Number (NaN) and infinity. Also when using double-type values, round-off errors (or rounding errors) may occur due to

double-type precision. Consider rounding your result value to the desired precision.

8.2.2. Data Window Aggregation Functions

The `first`, `last` and `window` aggregation functions return event properties of events present in a stream's data window. They are useful when information about current data window contents is required.

When comparing the `last` aggregation function to the `prev` function, the differences are as follows. The `prev` function is not an aggregation function and thereby not sensitive to the presence of `group by`. The `prev` function accesses data window contents directly and respects the sort order of the data window. The `last` aggregation function returns results based on arrival order and tracks data window contents in a separate shared data structure.

When comparing the `first` aggregation function to the `prevtail` function, the differences are as follows. The `prevtail` function is not an aggregation function and thereby not sensitive to the presence of `group by`. The `prevtail` function accesses data window contents directly and respects the sort order of the data window. The `first` aggregation function returns results based on arrival order and tracks data window contents in a separate shared data structure.

When comparing the `window` aggregation function to the `prevwindow` function, the differences are as follows. The `prevwindow` function is not an aggregation function and thereby not sensitive to the presence of `group by`. The `prevwindow` function accesses data window contents directly and respects the sort order of the data window. The `window` aggregation function returns results based on arrival order and tracks data window contents in a separate shared data structure.

When comparing the `count` aggregation function to the `prevcount` function, the differences are as follows. The `prevcount` function is not an aggregation function and thereby not sensitive to the presence of `group by`.

When comparing the `last` aggregation function to the `nth` aggregation function, the differences are as follows. The `nth` aggregation function does not consider out-of-order deletes (for example with on-delete and sorted windows) and does not revert to the prior expression value when the last event or nth-event was deleted from a data window. The `last` aggregation function tracks the data window and reflects out-of-order deletes.

From an implementation perspective, the `first`, `last` and `window` aggregation functions share a common data structure for each stream.

First Aggregation Function

The synopsis for the `first` aggregation function is:

```
first(*|stream.*|value_expression [, index_expression])
```

The `first` aggregation function returns properties of very first event in the data window. When used with `group by`, it returns properties of the first event in the data window for each group. When specifying an index expression, the function returns properties of the Nth-subsequent event to the first event, all according to order of arrival.

The first parameter to the function is required and defines the event properties or expression result to return. The second parameter is an optional `index_expression` that must return an integer value used as an index to evaluate the Nth-subsequent event to the first event.

You may specify the wildcard (*) character in which case the function returns the underlying event of the single selected stream. For joins and subqueries you must use the stream wildcard syntax below.

You may specify the stream name and wildcard (*) character in the *stream.** syntax. This returns the underlying event for the specified stream.

You may specify a *value_expression* to evaluate for the first event. The value expression may not select properties from multiple streams.

The *index_expression* is optional. If no index expression is provided, the function returns the first event. If present, the function evaluates the index expression to determine the value for N, and evaluates the Nth-subsequent event to the first event. A value of zero returns the first event and a value of 1 returns the event subsequent to the first event. You may not specify event properties in the index expression.

The function returns `null` if there are no events in the data window or when the index is larger than the number of events held in the data window. If used with `group by`, it returns `null` if there are no events in the data window for that group or when the index is larger than the number of events held in the data window for that group.

To explain, consider the statement below which selects the underlying event of the first sensor event held by the length window of 2 events.

```
select first(*) from SensorEvent.win:length(2)
```

Assume event E1, event E2 and event E3 are of type `SensorEvent`. When event E1 arrives the statement outputs the underlying event E1. When event E2 arrives the statement again outputs the underlying event E1. When event E3 arrives the statement outputs the underlying event E2, since event E1 has left the data window.

The stream wildcard syntax is useful for joins and subqueries. This example demonstrates a subquery that returns the first `SensorEvent` when a `DoorEvent` arrives:

```
select (select first(se.*) from SensorEvent.win:length(2) as se) from DoorEvent
```

The following example shows the use of an index expression. The output value for `f1` is the temperature property value of the first event in the data window, the value for `f2` is the temperature property value of the second event in the data window:

```
select first(temperature, 0) as f1, first(temperature, 1) as f2
from SensorEvent.win:time(10 sec)
```

Last Aggregation Function

The synopsis for the `last` aggregation function is:

```
last(*|stream.*|value_expression [, index_expression])
```

The `last` aggregation function returns properties of the very last event in the data window. When used with `group by`, it returns properties of the last event in the data window for each group. When specifying an index expression, the function returns properties of the Nth-prior event to the last event, all according to order of arrival.

Similar to the `first` aggregation function described above, you may specify the wildcard (*) character or stream name and wildcard (*) character or a *value_expression* to evaluate for the last event.

The *index_expression* is optional. If no index expression is provided, the function returns the last event. If present, the function evaluates the index expression to determine the value for N, and evaluates the Nth-prior event to the last event. A value of zero returns the last event and a value of 1 returns the event prior to the last

event. You may not specify event properties in the index expression.

The function returns `null` if there are no events in the data window or when the index is larger than the number of events held in the data window. If used with `group by`, it returns `null` if there are no events in the data window for that group or when the index is larger than the number of events held in the data window for that group.

The next statement selects the underlying event of the first and last sensor event held by the time window of 10 seconds:

```
select first(*), last(*) from SensorEvent.win:time(10 sec)
```

The statement shown next selects the last temperature (`f1`) and the prior-to-last temperature (`f1`) of sensor events in the last 10 seconds:

```
select last(temperature, 0) as f1, select last(temperature, 1) as f2
from SensorEvent.win:time(10 sec)
```

Window Aggregation Function

The synopsis for the window aggregation function is:

```
window(*|stream.*|value_expression)
```

The window aggregation function returns properties of all events in the data window. When used with `group by`, it returns properties of all events in the data window for each group.

Similar to the `first` aggregation function described above, you may specify the wildcard (*) character or stream name and wildcard (*) character or a *value_expression* to evaluate for all events.

The function returns `null` if there are no events in the data window. If used with `group by`, it returns `null` if there are no events in the data window for that group.

The next statement selects the underlying event of all events held by the time window of 10 seconds:

```
select window(*) from SensorEvent.win:time(10 sec)
```

The window aggregation function requires that your stream is bound by a data window. You may not use the window aggregation function on unbound streams with the exception of on-demand queries.

8.2.3. Additional Aggregation Functions

Esper provides the following additional aggregation functions beyond those in the SQL standard:

Table 8.3. Syntax and results of EPL aggregation functions

Aggregate Function	Result
<code>firstever(expression)</code>	<p>The <code>firstever</code> aggregation function returns the very first value ever. If used with <code>group by</code> it returns the first value ever for that group.</p> <p>If used with a data window, the result of the function does not change as data points leave a data window. Use the <code>first</code> or <code>prevtail</code> function to return values relative to a data window.</p>

Aggregate Function	Result
<code>lastever(<i>expression</i>)</code>	<p>Returns the last value or last value per group, if used with <code>group by</code>.</p> <p>This sample statement outputs the total price, the first price and the last price per symbol for the last 30 seconds of events and every 5 seconds:</p> <pre>select symbol, sum(price), lastever(price), firstever(price) from StockTickEvent.win:time(30 sec) group by symbol output every 5 sec</pre> <p>If used with a data window, the result of the function does not change as data points leave a data window (for example when all data points leave the data window). Use the <code>last</code> or <code>prev</code> function to return values relative to a data window.</p>
<code>leaving()</code>	<p>Returns true when any remove stream data has passed, for use in the <code>having</code> clause to output only when a data window has filled.</p> <p>The <code>leaving</code> aggregation function is useful when you want to trigger output after a data window has a remove stream data point. Use the <code>output after</code> syntax as an alternative to output after a time interval.</p> <p>This sample statement uses <code>leaving()</code> to output after the first data point leaves the data window, ignoring the first datapoint:</p> <pre>select symbol, sum(price) from StockTickEvent.win:time(30 sec) having leaving()</pre>
<code>nth(<i>expression</i>, <i>N_index</i>)</code>	<p>Returns the Nth oldest element; If N=0 returns the most recent value. If N=1 returns the value before the most recent value. If N is larger than the events held in the data window for this group, returns null.</p> <p>A maximum N historical values are stored, so it can be safely used to compare recent values in large views without incurring excessive overhead.</p> <p>As compared to the <code>prev</code> row function, this aggregation function works within the current <code>group by</code> group, see Section 3.7.2, “Output for Aggregation and Group-By”.</p> <p>This statement outputs every 2 seconds the groups that have new data and their last price and the previous-to-last price:</p> <pre>select symbol, nth(price, 1), last(price) from StockTickEvent group by symbol output last every 2 sec</pre>
<code>rate(<i>number_of_seconds</i>)</code>	<p>Returns an event arrival rate per second over the provided number of seconds, computed based on engine time.</p> <p>Returns null until events fill the number of seconds. Useful with <code>output</code></p>

Aggregate Function	Result
	<p>snapshot to output a current rate. Does not require a data window onto the stream(s).</p> <p>A sample statement to output, every 2 seconds, the arrival rate per second considering the last 10 seconds of events is shown here:</p> <pre data-bbox="608 394 1434 465">select rate(10) from StockTickEvent output snapshot every 2 sec</pre> <p>The aggregation function retains an engine timestamp value for each arriving event.</p>
<p><i>rate(timestamp_property[, accumulator])</i></p>	<p>Returns an event arrival rate over the data window including the last remove stream event. The <i>timestamp_property</i> is the name of a long-type property of the event that provides a timestamp value.</p> <p>The first parameter is a property name or expression providing milli-second timestamp values.</p> <p>The optional second parameter is a property or expression for computing an accumulation rate: If a value is provided as a second parameter then the accumulation rate for that quantity is returned (e.g. turnover in dollars per second).</p> <p>Requires a data window declared onto the stream. Returns null until events start leaving the window.</p> <p>This sample statement outputs event rate for each group (symbol) with fixed sample size of four events (and considering the last event that left). The <code>timestamp</code> event property must be part of the event for this to work.</p> <pre data-bbox="608 1301 1434 1397">select colour, rate(timestamp) as rate from StockTickEvent.std:groupwin(symbol).win:length(4) group by symbol</pre>

Built-in aggregation functions can be disabled via configuration (see Section 13.4.18.4, “Extended Built-in Aggregation Functions”). A custom aggregation function of the same name as a built-on function may be registered to override the built-in function.

8.3. User-Defined Functions

A user-defined function (UDF) is a single-row function that can be invoked anywhere as an expression itself or within an expression. The function must simply be a public static method that the classloader can resolve at statement creation time. The engine resolves the function reference at statement creation time and verifies parameter types.

You may register your own single-row function name for the user-defined function so that your EPL statements are less cluttered. This is described in detail in Section 15.3, “Single-Row Functions”.

User-defined functions can be also be invoked on instances of an event: Please see Section 4.4.5, “Using the Stream Name” to invoke event instance methods on a named stream.

The example below assumes a class `MyClass` that exposes a public static method `myFunction` accepting 2 parameters, and returning a numeric type such as `double`.

```
select 3 * com.mycompany.MyClass.MyFunction(price, volume) as myValue
from StockTick.win:time(30 sec)
```

User-defined functions also take array parameters as this example shows. The section on Section 7.5, “Array Definition Operator” outlines in more detail the types of arrays produced.

```
select * from RFIDEvent where com.mycompany.rfid.MyChecker.IsInZone(zone, {10, 20, 30})
```

Type names have to be fully qualified (e.g. `System.Math`) but Esper provides a mechanism for user-controlled imports of classes and packages as outlined in Section 13.4.5, “Class and package imports”.

User-defined functions can return any value including `null`, native objects or arrays. Therefore user-defined functions can serve to transform, convert or map events, or to extract information and assemble further events.

The following statement is a simple pattern that looks for events of type `E1` that are followed by events of type `E2`. It assigns the tags “e1” and “e2” that the function can use to assemble a final event for output:

```
select MyLib.MapEvents(e1, e2) from pattern [every e1=E1 -> e2=E2]
```

User-defined functions may also be chained: If a user-defined function returns an object then the object can itself be the target of the next function call and so on.

Assume that there is a `calculator` function in the `MyLib` class that returns a class which provides the `Search` method taking two parameters. The EPL that takes the result of the `Calculator` property and that calls the `Search` method on the result and returns its return value is shown below:

```
select MyLib.Calculator.Search(zonevariable, zone) from RFIDEvent]
```

A user-defined function should be implemented thread-safe.

Event Type Conversion via User-Defined Function

A function that converts from one event type to another event type is shown in the next example. The first statement declares a stream that consists of `MyEvent` events. The second statement employs a conversion function to convert `MyOtherEvent` events to events of type `MyEvent`:

```
insert into MyStream select * from MyEvent
insert into MyStream select MyLib.Convert(other) from MyOtherEvent as other
```

In the example above, assuming the event classes `MyEvent` and `MyOtherEvent` are types, the static method should have the following footprint:

```
public static MyEvent Convert(MyOtherEvent otherEvent)
```

User-Defined Function Result Cache

For user-defined functions that take no parameters or only constants as parameters the engine automatically caches the return result of the function, and invokes the function only once. This is beneficial to performance if your function indeed returns the same result for the same input parameters.

You may disable caching of return values of user-defined functions via configuration as described in Sec-

tion 13.4.18.3, “User-Defined Function or Static Method Cache”.

Parameter Matching

EPL follows CLR standards in terms of widening, performing widening automatically in cases where widening type conversion is allowed without loss of precision, for both boxed and primitive types.

When user-defined functions are overloaded, the function with the best match is selected based on how well the arguments to a function can match up with the parameters, giving preference to the function that requires the least number of widening conversions.

Boxing and unboxing of arrays is not supported in UDF. For example, an array of `int?` and an array of `int` are not compatible types.

When using `{}` array syntax in EPL, the resulting type is always a boxed type: `"{1, 2}"` is an array of `Integer` (and not `int` since it may contain null values), `"{1.0, 2d}"` is an array of `double` and `"{'A', 'B'}"` is an array of `string`, while `"{1, 'B', 2.0}"` is an array of `Object` (`Object[]`).

Chapter 9. EPL Reference: Enumeration Methods

9.1. Overview

EPL provides enumeration methods that work with lambda expressions to perform common tasks on subquery results, named windows, event properties or inputs that are or can be projected to a collection of events, scalar values or objects.

A lambda expression is an anonymous expression. Lambda expressions are useful for encapsulating user-defined expressions that are applied to each element in a collection. This section discusses built-in enumeration methods and their lambda expression parameters.

Lambda expressions use the lambda operator `=>`, which is read as "goes to". The left side of the lambda operator specifies the lambda expression input parameter(s) (if any) and the right side holds the expression. The lambda expression `x => x * x` is read "x goes to x times x.". Lambda expressions are also used for expression declaration as discussed in Section 4.2.7, "Expression Declaration".

When writing lambdas, you do not have to specify a type for the input parameter(s) or output result(s) because the engine can infer all types based on the input and the expression body. So if you are querying an `RFIDEvent`, for example, then the input variable is inferred to be an `RFIDEvent` event, which means you have access to its properties and methods.

The term *element* in respect to enumeration methods means a single event, scalar value or object in a collection that is the input to an enumeration method. The term *collection* means a sequence or group of elements.

The below table summarizes the built-in enumeration methods available:

Table 9.1. Enumeration Methods

Method	Result
<code>aggregate(seed, accumulator lambda)</code>	Aggregate elements by using seed as an initial accumulator value and applying an accumulator expression. Section 9.6.1, "Aggregate".
<code>alloy(predicate lambda)</code>	Return true when all elements satisfy a condition. Section 9.6.2, "Alloy".
<code>anyof(predicate lambda)</code>	Return true when any element satisfies a condition. Section 9.6.3, "AnyOf".
<code>average()</code>	Computes the average of values obtained from numeric elements. Section 9.6.4, "Average".
<code>average(projection lambda)</code>	Computes the average of values obtained from elements by invoking a projection expression on each element.

Method	Result
	Section 9.6.4, “Average”.
countof()	Returns the number of elements. Section 9.6.5, “CountOf”.
countof(predicate lambda)	Returns the number of elements that satisfy a condition. Section 9.6.5, “CountOf”.
except(source)	Produces the set difference of the two collections. Section 9.6.6, “Except”.
firstof()	Returns the first element. Section 9.6.7, “FirstOf”.
firstof(predicate lambda)	Returns the first element that satisfies a condition. Section 9.6.7, “FirstOf”.
groupby(key-selector lambda)	Groups the elements according to a specified key-selector expression. Section 9.6.8, “GroupBy”.
groupby(key-selector lambda, value-selector lambda)	Groups the elements according to a key-selector expression mapping each element to a value according to a value-selector. Section 9.6.8, “GroupBy”.
intersect(source)	Produces the set intersection of the two collections. Section 9.6.9, “Intersect”.
lastof()	Returns the last element. Section 9.6.10, “LastOf”.
lastof(predicate lambda)	Returns the last element that satisfies a condition. Section 9.6.10, “LastOf”.
leastFrequent()	Returns the least frequent value among a collection of values. Section 9.6.11, “LeastFrequent”.
leastFrequent(transform lambda)	Returns the least frequent value returned by the transform expression when applied to each element.

Method	Result
	Section 9.6.11, “LeastFrequent”.
max()	Returns the maximum value among a collection of elements. Section 9.6.12, “Max”.
max(value-selector lambda)	Returns the maximum value returned by the value-selector expression when applied to each element. Section 9.6.12, “Max”.
maxby(value-selector lambda)	Returns the element that provides the maximum value returned by the value-selector expression when applied to each element. Section 9.6.13, “MaxBy”.
min()	Returns the minimum value among a collection of elements. Section 9.6.12, “Max”.
min(value-selector lambda)	Returns the minimum value returned by the value-selector expression when applied to each element. Section 9.6.14, “Min”.
minby(value-selector lambda)	Returns the element that provides the minimum value returned by the value-selector expression when applied to each element.. Section 9.6.15, “MinBy”.
mostFrequent()	Returns the most frequent value among a collection of values. Section 9.6.16, “MostFrequent”.
mostFrequent(transform lambda)	Returns the most frequent value returned by the transform expression when applied to each element. Section 9.6.16, “MostFrequent”.
orderBy()	Sorts the elements in ascending order. Section 9.6.17, “OrderBy and OrderByDesc”.
orderBy(key-selector lambda)	Sorts the elements in ascending order according to a key. Section 9.6.17, “OrderBy and OrderByDesc”.
orderByDesc()	Sorts the elements in descending order.

Method	Result
	Section 9.6.17, “OrderBy and OrderByDesc”.
orderByDesc(key-selector lambda)	Sorts the elements in descending order according to a key. Section 9.6.17, “OrderBy and OrderByDesc”.
reverse	Reverses the order of elements. Section 9.6.18, “Reverse”.
selectFrom(transform lambda)	Transforms each element resulting in a collection of transformed elements. Section 9.6.19, “SelectFrom”.
sequenceEqual(second)	Determines whether two collections are equal by comparing each element (<code>equals</code> semantics apply). Section 9.6.20, “SequenceEqual”.
sumOf()	Computes the sum from a collection of numeric elements. Section 9.6.21, “SumOf”.
sumOf(projection lambda)	Computes the sum by invoking a projection expression on each element. Section 9.6.21, “SumOf”.
take(numElements)	Returns a specified number of contiguous elements from the start. Section 9.6.22, “Take”.
takeLast(numElements)	Returns a specified number of contiguous elements from the end. Section 9.6.23, “TakeLast”.
takeWhile(predicate lambda)	Returns elements from the start as long as a specified condition is true. Section 9.6.24, “TakeWhile”.
takeWhile((predicate, index) lambda)	Returns elements from the start as long as a specified condition is true, allowing each element's index to be used in the logic of the predicate expression. Section 9.6.24, “TakeWhile”.
takeWhileLast(predicate)	Returns elements from the end as long as a specified condition is true. Section 9.6.25, “TakeWhileLast”.

Method	Result
<code>takeWhileLast((predicate,index) lambda)</code>	Returns elements from the end as long as a specified condition is true, allowing each element's index to be used in the logic of the predicate expression. Section 9.6.25, “TakeWhileLast”.
<code>toMap(key-selector lambda, value-selector lambda)</code>	Returns a Map according to specified key selector and value-selector expressions. Section 9.6.26, “ToMap”.
<code>union(source)</code>	Forms a union of the input elements with source elements. Section 9.6.27, “Union”.
<code>where(predicate lambda)</code>	Filters elements based on a predicate. Section 9.6.28, “Where”.
<code>where((predicate,index) lambda)</code>	Filters elements based on a predicate, allowing each element's index to be used in the logic of the predicate expression. Section 9.6.28, “Where”.

9.2. Example Events

The examples in this section come out of the domain of location report (aka. RFID, asset tracking etc.) processing:

1. The `Item` event is a report of the location of a certain item. An item can be either a piece of luggage or a passenger.
2. The `LocationReport` event contains a list of `Item` items for which it reports location.
3. The `Zone` event describes areas that items may move through.

The examples use example single-row functions for computing the distance (`distance`) and for determining if a location falls within a rectangle (`inrect`) that are not provided by the EPL language. These example UDF functions are not enumeration methods and are used in EPL statements to provide a sensible example.

The `Item` event contains an `assetId` id, a (x,y) location, a `luggage` flag to indicate whether the item represents a luggage (true) or passenger (false), and the `assetIdPassenger` that holds the asset id of the associated passenger when the item is a piece of luggage.

The `Item` event is defined as follows (access methods not shown for brevity):

```
public class Item {
    String assetId;           // passenger or luggage asset id
    Location location;        // (x,y) location
    boolean luggage;          // true if this item is a luggage piece
    String assetIdPassenger;  // if the item is luggage, contains passenger associated
    ...
}
```

The `LocationReport` event contains a list of `Item` items for which it reports events.

The `LocationReport` event is defined as follows:

```
public class LocationReport {  
    List<Item> items;  
    ...  
}
```

The `Zone` event contains a zone name and (x1, y1, x2, y2) rectangle.

The `Zone` event is defined as follows:

```
public class Zone {  
    String name;  
    Rectangle rectangle;  
    ...  
}
```

The `Location` object is a nested object to `Item` and provides the current (x,y) location:

```
public class Location {  
    int x;  
    int y;  
    ...  
}
```

The `Rectangle` object is a nested object to `Zone` and provides a zone rectangle(x1,y1,x2,y2):

```
public class Rectangle {  
    int x1;  
    int y1;  
    int x2;  
    int y2;  
    ...  
}
```

9.3. How to Use

9.3.1. Syntax

The syntax for enumeration methods is the same syntax as for any chained invocation:

```
input_coll.enum_method_name( [method_parameter [, method_parameter [...]]]  
    .[ [enum_method_name(...) [...]] | property_name]
```

Following the `input_coll` input collection (options outlined below), is the `.` (dot) operator and the `enum_method_name` enumeration method name. It follows in parenthesis a comma-separated list of method parameter expressions. Additional enumeration methods can be chained thereafter. An event property name can follow for those enumeration methods returning an event-typed (non-scalar) element.

If the method parameter is a lambda expression with a single lambda-parameter, specify the lambda-parameter name followed by the `=>` lambda operator and followed by the expression. The synopsis for use with a single lambda-parameter is:

```
method_parameter: lamda_param => lamda_expression
```

If the method parameter is a lambda expression with two or more lambda-parameters, specify the lambda parameter names in parenthesis followed by the `=>` lambda operator followed by the expression. The synopsis for

use with multiple lambda-parameters is:

```
method_parameter: (lamda_param [,lamda_param [,...]]) => lamda_expression
```

Generally for lambda expressions, the engine applies the lambda expression to each element in the input collection. The expression yields a result that, depending on the particular enumeration method, is used for aggregation, as a filter or for output, for example.

9.3.2. Introductory Examples

Let's look at an EPL statement that employs the `where` enumeration method and a lambda expression. This example returns items that have a (x, y) location of (0, 0):

```
select items.where(i => i.location.x = 0 and i.location.y = 0) as zeroloc
from LocationReport
```

As enumeration methods can be chained, this selection is equivalent:

```
select items.where(i => i.location.x = 0).where(i => i.location.y = 0) as zeroloc
from LocationReport
```

According to above statement the engine outputs in field `zeroloc` a collection of `Item` objects matching the condition.

The `where` enumeration method has a second version that has two lambda-parameters. The second parameter is the name of the index property which represents the current index of the element within the collection.

This sample query returns a collection that consists of the first 3 items. This sample query does not use the `item` lambda parameter:

```
select items.where( (item, indexElement) => indexElement < 3) as firstThreeItems
from LocationReport
```

9.3.3. Input, Output and Limitations

It is not necessary to use classes for event representation. The example above applies the same to Map or XML underlying events.

For most enumeration methods the input can be any collection of events, scalar values or objects. For some enumeration methods limitations apply that are documented below. For example, the `sumOf` enumeration method requires a collection of numeric scalar values if used without parameters. If the input to `sumOf` is a collection of events the enumeration method requires a lambda expression as parameter that yields the numeric value to use to compute the sum.

Many examples of this section operate on the collection returned by the event property `items` in the `LocationReport` event class. There are many other inputs yielding collections as listed below. Most examples herein use an event property as a input simply because the example can thus be brief and does not need to refer to a sub-query or named window or other concept.

For enumeration methods that return a collection, for example `where` and `orderBy`, the engine outputs an implementation of the `Collection` interface that contains the selected value(s). The collection returned must be considered read-only. As Java does not allow resettable iterators, the `Collection` interface allows more flexibility to query size and navigate among collection elements. We recommend against down-casting a collection re-

turned by the engine to a more specific subclass of the `Collection` interface.

For enumeration methods that return an element, for example `firstOf`, `lastOf`, `minBy` and `maxBy` the engine outputs the scalar value or the underlying event if operating on events. You may add an event property name after the enumeration method to return a property value.

Enumeration methods generally retain the order of elements provided by the collection.

The following restrictions apply to enumeration methods:

1. Enumeration methods returning a collection return a read-only implementation of the `Collection` interface. You may not use any of the write-methods such as `add` or `remove` on a result collection.

9.4. Inputs

The input of data for built-in enumeration methods is a collection of scalar values, events or other objects. Input can originate from any of the following:

1. A subquery.
2. A named window.
3. A property of an event that is itself a collection of events or classes, i.e. an indexed property.
4. Any of the data window aggregation functions (`window`, `first`, `last`).
5. The special `prevwindow`, `prev` and `prevtail` single-row functions.
6. A plug-in single-row function or user-defined function.
7. A declared expression.
8. Another enumeration method that returns a collection.
9. An array returned by the `{ }` array operator.

9.4.1. Subquery Results

Subqueries can return the rows of another stream's data window or rows from a named window. By providing a where-clause the rows returned by a subquery can be correlated to data provided by stream(s) in the from-clause. See Section 4.11, “Subqueries”.

A subquery that selects `(*)` wildcard provides a collection of events as input. A subquery that selects a single value expression provides a collection of scalar values as input. Subqueries that selects multiple value expressions are not allowed as input to enumeration methods.

The following example uses a subquery to retrieve all zones for each location report item where the location falls within the rectangle of the zone. Please see a description of example events and functions above.

```
select assetId,
  (select * from Zone.std:unique(name)).where(z => inrect(z.rectangle, location)) as zones
from Item
```

You may place the subquery in an expression declaration to reuse the subquery in multiple places of the same EPL statement.

This sample EPL declares the same query as above in an expression declaration:

```
expression myquery {itm =>
  (select * from Zone.std:unique(name)).where(z => inrect(z.rectangle, itm.location))
}
select assetId, myquery(item) as subq,
  myquery(item).where(z => z.zone = 'Z01') as assetItem
```

```
from Item as item
```

The above query also demonstrates how an enumeration method, in the example the `where`-method, can be run across the results returned by a subquery in an expression declaration.

Place a single column in the subquery select-clause to provide a collection of scalar values as input.

The next example selects all names of zones and orders the names returning an order collection of string names every 30 seconds:

```
select (select name from Zone.std:unique(name)).orderBy() as orderedZones
from pattern[every timer:interval(30)]
```

Note that the engine can cache intermediate results thereby is not forced to re-evaluate the subquery for each occurrence in the `select`-clause.

9.4.2. Named Window

Named windows are globally-visible data windows. See Section 4.15, “Creating and Using Named Windows”.

You may specify the named window name as input for an enumeration method and can optionally provide a correlation where-clause. The syntax is equivalent to a sub-query against a named window but much shorter.

Synopsis:

```
named-window-name[(correlation-expression)].enum-method-name(...)
```

When selecting all events in a named window you do not need the *correlation-expression*. To select a subset of data in the named window, specify a *correlation-expression*. From the perspective of best runtime performance, a correlation expression is preferred to reduce the number of rows returned.

The following example first declares a named window to hold the last zone event per zone name:

```
create window ZoneWindow.std:unique(name) as Zone
```

Then we create a statement to insert zone events that arrive to the named window:

```
insert into ZoneWindow select * from Zone
```

Finally this statement queries the named window to retrieve all zones for each location report item where the location falls within the rectangle of the zone:

```
select ZoneWindow.where(z => inrect(z.rectangle, location)) as zones from Item
```

If you have a filter or correlation expression, append the expression to the named window name and place in parenthesis.

This slightly modified query is the example above except that it adds a filter expression such that only zones with name Z1, Z2 or Z3 are considered:

```
select ZoneWindow(name in ('Z1', 'Z2', 'Z3')).where(z => inrect(z.rectangle, location)) as zones
from Item
```

You may prefix property names provided by the named window with the name to disambiguate property

names.

This sample query prefixed the `name` property and returns the count of matching zones:

```
select ZoneWindow(ZoneWindow.name in ('Z1', 'Z2', 'Z3')).countof() as zoneCount
from Item
```

The engine internally interprets the shortcut syntax and creates a subquery from it. Thus all indexing and query planning for subqueries against named windows apply here as well.

9.4.3. Event Property

Indexed event properties are event properties that are a collection, array or iterable of scalar values or objects.

The `LocationReport` event from the example contains a list of `Item` events. Any indexed property (list, array, collection, iterable) is eligible for use as input to an enumeration method. If the indexed property contains non-scalar objects the objects are treated as events and can be used as input to enumeration methods as a collection of events.

The next sample query returns items that are less then 20 units away from the center, taking the `items` event property provided by each `LocationReport` event as input:

```
select items.where(p => distance(0, 0, p.location.x, p.location.y) < 20) as centeritems
from LocationReport
```

The following example consists of two statements: The first statement declares an a new event type and the second statement invokes the `sequenceEqual` method to compare sequences contained in two properties of the same event:

```
create schema MyEvent (seqone String[], seqtwo String[])
```

```
select seqone.sequenceEqual(seqtwo) from MyEvent
```

9.4.4. Data Window Aggregation

Data window aggregation functions return events present in a stream's current data window. They are aggregation functions and as such sensitive to the presence of `group by`. See Section 8.2.2, “Data Window Aggregation Functions”.

When using data window aggregation functions as input to a built-in enumeration method you can specify the `*` wildcard to provide a collection of events as input. Or you can specify a property name in the data window aggregation function to provide a collection of scala values as input. Use any of the `window`, `first` or `last` functions.

In this example query the `window(*)` aggregation function returns the last 10 seconds of item location reports for the same asset id as the incoming event. Among that last 10 seconds of events for the same asset id, the enumeration method returns those item location reports where the distance to center is less then 20, for each arriving `Item` event.

Sample query:

```
select window(*).where(p => distance(0, 0, p.location.x, p.location.y) < 20) as centeritems
from Item(type='P').win:time(10) group by assetId
```

The next sample query instead selects the asset id property of all events and returns an ordered collection:

```
select window(assetId).orderBy() as orderedAssetIds
from Item.win:time(10) group by assetId
```

9.4.5. `prev`, `prevwindow` and `prevtail` Single-Row Functions as Input

The `prev`, `prevwindow` and `prevtail` single-row functions allow access into a stream's data window however are not aggregation functions and as such not sensitive to the presence of `group by`. See Section 8.1.10, “The Previous-Window Function”.

When using any of the `prev` single-row functions as input to a built-in enumeration method you can specify the stream name as a parameter to the function or an event property. The input to the enumeration method is a collection of events if you specify the stream name, or a collection of scalar value if you specify an event property.

In this example query the `prevwindow(stream)` single-row function returns the last 10 seconds of item location reports, among which the enumeration method filters those item location reports where the distance to center is less then 20, for each Item event that arrived in the last 10 seconds considering passenger-type Item events only (see filter type = 'P').

Sample query:

```
select prevwindow(items)
  .where(p => distance(0, 0, p.location.x, p.location.y) < 20) as centeritems
from Item(type='P').win:time(10) as items
```

This sample query demonstrates the use of the `prevwindow` function to return a collection of scalar values (collection of asset id) as input to `orderBy`:

```
select prevwindow(assetId).orderBy() as orderedAssetIds
from Item.win:time(10) as items
```

9.4.6. Single-Row Function or UDF

Your single-row or user-defined function can return either an array or any collection that implements either the `Collection` or `Iterable` interface. For arrays, the array component type and for collections, the collection or iterable generic type should be the class providing event properties.

As an example, assume a `ZoneFactory` class exists and a static method `getZones()` returns a list of zones to filter items, for example:

```
public class ZoneFactory {
    public static Iterable<Zone> getZones() {
        List<Zone> zones = new ArrayList<Zone>();
        zones.add(new Zone("Z1", new Rectangle(0, 0, 20, 20)));
        return zones;
    }
}
```

Import the class through runtime or static configuration, or add the method above as a plug-in single-row function.

The following query returns for each Item event all zones that the item belongs to:

```
select ZoneFactory.getZones().where(z => inrect(z.rectangle, item.location)) as zones
from Item as item
```

If the class and method were registered as a plug-in single-row function, you can leave the class name off, for example:

```
select getZones().where(z => inrect(z.rectangle, item.location)) as zones
from Item as item
```

Your single-row or user-defined function can also return an array, collection or iterable or scalar values.

For example, the static method `getZoneNames()` returns a list of zone names:

```
public static String[] getZoneNames() {
    return new String[] { "Z1", "Z2" };
}
```

The following query returns zone names every 30 seconds and excludes zone Z1:

```
select getZoneNames().where(z => z != "Z1")
from pattern[every timer:interval(30)]
```

9.4.7. Declared Expression

A declared expression may return input data for an enumeration method.

The below query declares an expression that returns all passenger location reports among the items in the location report event in a column named `passengerCollection`. The query uses the result returned by the declared expression a second time to filter through the list returning the passenger location report where the asset id is a given value in a column named `passengerP01`.

Sample query:

```
expression passengers {
    lr => lr.items.where(l => l.type='P')
}
select passengers(lr) as passengerCollection,
    passengers(lr).where(x => assetId = 'P01') as passengerP01
from LocationReport lr
```

The engine applies caching techniques to avoid re-evaluating the declared expression multiple times.

9.5. Example

Following the RFID asset tracking example as introduced earlier, this section introduces two use cases solved by enumeration methods.

The first use case requires us to find any luggage that is more than 20 units away from the passenger that the luggage belongs to. The declared expression `lostLuggage` solves this question.

The second question to answer is: For each of such lost luggage what single other passenger is nearest to that luggage. The declared expression `nearestOwner` which uses `lostLuggage` answers this question.

Below is the complete EPL statement (one statement not multiple):

```
// expression to return a collection of lost luggage
expression lostLuggage {
    lr => lr.items.where(l => l.type='L' and
        lr.items.some(p => p.type='P' and p.assetId=l.assetIdPassenger
```

```

        and LRUtil.distance(l.location.x, l.location.y, p.location.x, p.location.y) > 20))
    }

    // expression to return all passengers
    expression passengers {
        lr => lr.items.where(l => l.type='P')
    }

    // expression to find the nearest owner
    expression nearestOwner {
        lr => lostLuggage(lr).toMap(key => key.assetId,
            value => passengers(lr).minBy(
                p => LRUtil.distance(value.location.x, value.location.y, p.location.x, p.location.y))
            )
    }

    select lostLuggage(lr) as val1, nearestOwner(lr) as val2 from LocationReport lr

```

9.6. Reference

9.6.1. Aggregate

The `aggregate` enumeration method takes an expression providing the initialization value (seed) and an accumulator lambda expression. The return value is the final accumulator value.

Via the `aggregate` method you may perform a calculation over elements. The method initializes the aggregated value by evaluating the expression provided in the first parameter. The method then calls the lambda expression of the second parameter once for each element in the input. The lambda expression receives the last aggregated value and the element from the input. The result of the expression replaces the previous aggregated value and returns the final result after completing all elements.

An expression example with scalar values:

```
{1, 2, 3}.aggregate(0, (result, value) => result + value) // Returns 6
```

The example below aggregates price of each `OrderEvent` in the last 10 seconds computing a total price:

```

// Initialization value is zero.
// Aggregate by adding up the price.
select window(*).aggregate(0, (result, order) => result + order.price) as totalPrice
from OrderEvent.win:time(10)

```

In the query above, the initialization value is zero, `result` is used for the last aggregated value and `order` denotes the element that the expression adds the value of the price property.

This example aggregation builds a comma-separated list of all asset ids of all items:

```

select items.aggregate('',
    (result, item) => result || (case when result='' then '' else ',' end) || item.assetId) as assets
from LocationReport

```

In above query, the empty string `''` represents the initialization value. The name `result` is used for the last aggregated value and the name `item` is used to denote the element.

The type value returned by the initialization expression must match to the type of value returned by the accumulator lambda expression.

If the input is null the method returns null. If the input is empty the method returns the initialization value.

9.6.2. AllOf

The `allOf` enumeration method determines whether all elements satisfy the predicate condition.

The method takes a single parameter: The predicate lambda expression that must yield a Boolean result. The enumeration method applies the lambda expression to each element and if the expression returns true for all elements, the method returns true.

An expression example with scalar values:

```
{1, 2, 3}.allOf(v => v > 0)    // Returns true as all values are > 0
{1, 2, 3}.allOf(v => v > 1)    // Returns false
```

The EPL statement below returns true when all items are within 1000 unit distance of center:

```
select items.allOf(i => distance(i.location.x, i.location.y, 0, 0) < 1000) as centered
from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns true.

9.6.3. AnyOf

The `anyOf` enumeration method determines whether any element satisfies the predicate condition.

The only parameter is the predicate lambda expression that must yield a Boolean result. The enumeration method applies the lambda expression to each element and if the expression returns true for all elements, the method returns true.

An expression example with scalar values:

```
{1, 2, 3}.anyOf(v => v > 0)    // Returns true
{1, 2, 3}.anyOf(v => v > 1)    // Returns true
{1, 2, 3}.anyOf(v => v > 3)    // Returns false
```

The EPL statement below return true when any of the items are within 10 unit distance of center:

```
select items.anyOf(i => distance(i.location.x, i.location.y, 0, 0) < 10) as centered
from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns false.

9.6.4. Average

The `average` enumeration method computes the average of scalar values. If passing a projection lambda expression the method computes the average obtained by invoking the projection lambda expression on each element.

The method takes a projection lambda expression yielding a numeric value as a parameter. It applies the lambda expression to each element and computes the average of the result, returning a Double value. A BigDecimal is returned for expressions returning BigInteger or BigDecimal.

An expression example with scalar values:

```
{1, 2, 3}.average()    // Returns 2
```

The EPL statement as shown next computes the average distance from center among all items in the location report event:

```
select items.average(i => distance(i.location.x, i.location.y, 0, 0)) as avgdistance
from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns double zero or BigDecimal zero.

9.6.5. CountOf

The `countof` enumeration method returns the number of elements, or the number of elements that satisfy a condition.

The enumeration method has two versions: The first version takes no parameters and computes the number of elements. The second version takes a predicate lambda expression that must yield Boolean true or false, and computes the number of elements that satisfy the condition.

An expression example with scalar values:

```
{1, 2, 3}.countOf()    // Returns 3
{1, 2, 3}.countOf(v => v < 2)    // Returns 1
```

The next sample statement counts the number of items:

```
select items.countOf() as cnt from LocationReport
```

This example statement counts the number of items that have a distance to center that is less then 20 units:

```
select items.countOf(i => distance(i.location.x, i.location.y, 0, 0) < 20) as cntcenter
from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns integer zero.

9.6.6. Except

The `except` enumeration method forms a set difference of the input elements with the elements that the parameter expression yields.

The enumeration method takes a single parameter that must itself return a collection of events, objects or scalar values. The method returns the elements of the first collection that do not appear in the second collection.

An expression example with scalar values:

```
{1, 2, 3}.except({1})    // Returns {2, 3}
```

The following statement compares the items of the last location report against all items in the previous 10 location reports, and reports for each combination only those items in the current item report that are not also in the location report compared to:

```
select za.items.except(zb.items) as itemsCompared
from LocationReport as za unidirectional, LocationReport.win:length(10) as zb
```


If the input is null the method returns null. For scalar values and objects equals-semantics apply.

9.6.7. FirstOf

The `firstOf` enumeration method returns the first element or the first element that satisfies a condition.

The method has two versions: The first version takes no parameters and returns the first element. The second version takes a predicate lambda expression yielding true or false. It applies the lambda expression to each element and returns the first element for which the expression returns true. The return type is the element itself and not a collection. You may append a property name to return the property value for the first element.

An expression example with scalar values:

```
{1, 2, 3}.firstOf()    // Returns 1
{1, 2, 3}.firstOf(v => v / 2 > 1)  // Returns 3
```

In the following EPL sample the query returns the first item that has a distance to center that is less than 20 units:

```
select items.firstof(i => distance(i.location.x, i.location.y, 0, 0) < 20) as firstcenter
from LocationReport
```

The next sample EPL returns the first item's asset id:

```
select items.firstof().assetId as firstAssetId from LocationReport
```

If the input is null, empty or if none of the elements match the condition the method returns null.

9.6.8. GroupBy

The `groupby` enumeration method groups the elements according to a specified key-selector lambda expression. There are two versions of the `groupby` method.

The first version of the method takes a key-selector lambda expression and returns a Map of key with each value a list of objects, one for each distinct key that was encountered. The result is a `Map<Object, Collection<Object>>` wherein object is the event underlying object.

The second version of the method takes a key-selector lambda expression and value-selector lambda expression and returns a Map of key with each value a list of values, one for each distinct key that was encountered. The result is a `Map<Object, Collection<Object>>` wherein object is the result of applying the value-selector expression.

The next query filters out all luggage items using a `where` method and then groups by the luggage's passenger asset id. It returns a map of passenger asset id and the collection of luggage items for each passenger:

```
select items.where(type='L').groupby(i => assetIdPassenger) as luggagePerPerson
from LocationReport
```

The query shown below generates a map of item asset id and distance to center:

```
select items.groupby(
    k => assetId, v => distance(v.location.x, v.location.y, 0, 0)) as distancePerItem
from LocationReport
```

If the input is null the method returns null. Null values as key and value are allowed.

9.6.9. Intersect

The `intersect` enumeration method forms a set intersection of the input elements with the elements that the parameter expression yields.

The enumeration method takes a single parameter that must itself return a collection of events, objects or scalar values. The method returns the elements of the first collection that also appear in the second collection.

An expression example with scalar values:

```
{1, 2, 3}.intersect({2, 3})    // Returns {2, 3}
```

The following statement compares the items of the last location report against all items in the previous 10 location reports, and reports for each combination all items in the current item report that also occur in the other location report:

```
select za.items.intersect(zb.items) as itemsCompared
from LocationReport as za unidirectional, LocationReport.win:length(10) as zb
```

If the input is null the method returns null. For scalar values and objects equals-semantics apply.

9.6.10. LastOf

The `lastOf` enumeration method returns the last element or the last element that satisfies a condition.

The method has two versions: The first version takes no parameters and returns the last element. The second version takes a predicate lambda expression yielding true or false. It applies the lambda expression to each element and returns the last element for which the expression returns true. The return type is the element itself and not a collection. You may append a property name to return the property value for the last element.

An expression example with scalar values:

```
{1, 2, 3}.lastOf()           // Returns 3
{1, 2, 3}.lastOf(v => v < 3)  // Returns 2
```

In the following EPL sample the query returns the last item that has a distance to center that is less than 20 units:

```
select items.lastof(i => distance(i.location.x, i.location.y, 0, 0) < 20) as lastcenter
from LocationReport
```

The next sample EPL returns the last item's asset id:

```
select items.lastof().assetId as lastAssetId from LocationReport
```

If the input is null, empty or if none of the elements match the condition the method returns null.

9.6.11. LeastFrequent

The `leastFrequent` enumeration method returns the least frequent value among a collection of values, or the least frequent value after applying a transform expression to each element.

The method has two versions: The first version takes no parameters and returns the least frequent value. The second version takes a transform lambda expression yielding the value to count occurrences for. The method applies the lambda expression to each element and returns the expression result value with the least number of occurrences. The return type is the type of value in the collection or the type of value returned by the transform lambda expression if one was provided.

An expression example with scalar values:

```
{1, 2, 3, 2, 1}.leastFrequent()    // Returns 3
```

The example EPL below returns the least frequent item type, counting the distinct item types among all items for the current LocationReport event:

```
select items.leastFrequent(i => type) as leastFreqType from LocationReport
```

If the input is null or empty the method returns null. The transform expression may also yield null. A null value can be returned as the most frequent value if the most frequent value is null. If multiple values have the same number of occurrences the method returns the first value with the least number of occurrences considering the ordering of the collection.

9.6.12. Max

The `max` enumeration method returns the maximum value among a collection of values.

If no value-selector lambda expression is provided, the method finds the maximum.

If a value-selector lambda expression is provided, the enumeration method invokes a value-selector lambda expression on each element and returns the maximum value. The type of value returned follows the return type of the lambda expression that was provided as parameter.

An expression example with scalar values:

```
{1, 2, 3, 2, 1}.max()    // Returns 3
```

The next query returns the maximum distance of any item from center:

```
select items.max(i => distance(i.location.x, i.location.y, 0, 0)) as maxcenter  
from LocationReport
```

The value-selector lambda expression must return a comparable type: Any primitive or boxed type or `Comparable` type is permitted.

If the input is null, empty or if none of the elements when transformed return a non-null value the method returns null.

9.6.13. MaxBy

The `maxBy` enumeration method returns the element that provides the maximum value returned by the value-selector lambda expression when applied to each element.

The enumeration method returns the element itself. You may append an event property name to return a property value of the element.

The next query returns the first item with the maximum distance to center:

```
select items.maxBy(i => distance(i.location.x, i.location.y, 0, 0)) as maxItemCenter
from LocationReport
```

The next sample returns the type of the item with the largest asset id (string comparison) among all items:

```
select items.maxBy(i => assetId).type as minAssetId from LocationReport
```

The transform expression must return a comparable type: Any primitive or boxed type or `Comparable` type is permitted.

If the input is null, empty or if none of the elements when transformed return a non-null value the method returns null.

9.6.14. Min

The `min` enumeration method returns the minimum value among a collection of values.

If no value-selector lambda expression is provided, the method finds the minimum.

If a value-selector lambda expression is provided, the enumeration method invokes a value-selector lambda expression on each element and returns the minimum value. The type of value returned follows the return type of the lambda expression that was provided as parameter.

An expression example with scalar values:

```
{1, 2, 3, 2, 1}.min()    // Returns 1
```

The next query returns the minimum distance of any item to center:

```
select items.min(i => distance(i.location.x, i.location.y, 0, 0)) as mincenter
from LocationReport
```

The transform expression must return a comparable type: Any primitive or boxed type or `Comparable` type is permitted.

If the input is null, empty or if none of the elements when transformed return a non-null value the method returns null.

9.6.15. MinBy

The `minBy` enumeration method returns the element that provides the minimum value returned by the value-selector lambda expression when applied to each element.

The enumeration method returns the element itself. You may append an event property name to return a property value of the element.

The next query returns the first item with the minimum distance to center:

```
select items.minBy(i => distance(i.location.x, i.location.y, 0, 0)) as minItemCenter
from LocationReport
```

The next sample returns the type of the item with the smallest asset id (string comparison) among all items:

```
select items.minBy(i => assetId).type as minAssetId from LocationReport
```

The transform expression must return a comparable type: Any primitive or boxed or `Comparable` type is permitted.

If the input is null, empty or if none of the elements when transformed return a non-null value the method returns null.

9.6.16. MostFrequent

The `mostFrequent` enumeration method returns the most frequent value among a collection of values, or the most frequent value after applying a transform expression to each element.

The method has two versions: The first version takes no parameters and returns the most frequent value. The second version takes a transform lambda expression yielding the value to count occurrences for. The method applies the lambda expression to each element and returns the expression result value with the most number of occurrences. The return type is the type of value in the collection or the type of value returned by the transform lambda expression if one was provided.

An expression example with scalar values:

```
{1, 2, 3, 2, 1, 2}.mostFrequent() // Returns 2
```

The example EPL below returns the least frequent item type, counting the distinct item types among all items for the current `LocationReport` event:

```
select items.leastFrequent(i => type) as leastFreqType from LocationReport
```

If the input is null or empty the method returns null. The transform expression may also yield null. A null value can be returned as the most frequent value if the most frequent value is null. If multiple values have the same number of occurrences the method returns the first value with the most number of occurrences considering the ordering of the collection.

9.6.17. OrderBy and OrderByDesc

The `orderBy` enumeration method sorts elements in ascending order according to a key. The `orderByDesc` enumeration method sorts elements in descending order according to a key.

The enumeration method takes a single key-selector lambda expression as parameter and orders elements according to the key yielded by the expression. For same-value keys, it maintains the existing order.

An expression example with scalar values:

```
{2, 3, 2, 1}.orderBy() // Returns {1, 2, 2, 3}
```

This example orders all items from a location report according to their distance from center:

```
select items.orderBy(i => distance(i.location.x, i.location.y, 0, 0)) as itemsNearFirst,
       items.orderByDesc(i => distance(i.location.x, i.location.y, 0, 0)) as itemsFarFirst
from LocationReport
```

The key-selector lambda expression must return a comparable type: Any primitive or boxed or `Comparable` type is permitted.

If the input is null the method returns null. If the input is empty the method returns an empty collection.

9.6.18. Reverse

The `reverse` enumeration method simply reverses the order of elements returning a collection.

An expression example with scalar values:

```
{2, 3, 2, 1}.reverse()    // Returns {1, 2, 3, 2}
```

The following EPL reverses the items:

```
select items.reverse() as reversedItems from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns an empty collection.

9.6.19. SelectFrom

The `selectFrom` enumeration method transforms each element resulting in a collection of transformed elements.

The enumeration method applies a transformation lambda expression to each element and returns the result of each transformation as a collection. Use the `new` operator to yield multiple values for each element, see ???.

The next EPL query returns a collection of asset ids:

```
select items.selectFrom(i => assetId) as itemAssetIds from LocationReport
```

This sample EPL query evaluates each item and returns the asset id as well as the distance from center for each item:

```
select items.selectFrom(i =>
  new {
    assetId,
    distanceCenter = distance(i.location.x, i.location.y, 0, 0)
  } ) as itemInfo from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns an empty collection.

9.6.20. SequenceEqual

The `sequenceEqual` enumeration method determines whether two collections are equal by comparing each element.

The method enumerates the two source collections in parallel and compares corresponding elements by using the `equals` method to compare. The method takes a single parameter expression that must return a collection containing elements of the same type as the input. The method returns true if the two source sequences are of equal length and their corresponding elements are equal.

An expression example with scalar values:

```
{1, 2, 3}.sequenceEqual({1})    // Returns false
{1, 2, 3}.sequenceEqual({1, 2, 3}) // Returns true
```

The following example compares the asset id of all items to the asset ids returned by a method `ItemUtil.redListed()` which is assumed to return a list of asset id of string type:

```
select items.selectFrom(i => assetId).sequenceEquals(ItemUtil.redListed()) from LocationReport
```

If the input is null the method returns null.

9.6.21. SumOf

The `sumOf` enumeration method computes the sum. If a projection lambda expression is provided, the method invokes the projection lambda expression on each element and computes the sum on each returned value.

The projection lambda expression should yield a numeric value, `BigDecimal` or `BigInteger` value. Depending on the type returned by the projection lambda expression the method returns either `Integer`, `Long`, `Double`, `BigDecimal` or `BigInteger`.

An expression example with scalar values:

```
{1, 2, 3}.sumOf()    // Returns 6
```

The following example computes the sum of the distance of each item to center:

```
select items.sum(i => distance(i.location.x, i.location.y, 0, 0) as totalAllDistances  
from LocationReport
```

If the input is null or empty the method returns null.

9.6.22. Take

The `take` enumeration method returns a specified number of contiguous elements from the start.

The enumeration method takes a single size (non-lambda) expression that returns an `Integer` value.

An expression example with scalar values:

```
{1, 2, 3}.take(2)    // Returns {1, 2}
```

The following example returns the first 5 items:

```
select items.take(5) as first5Items from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns an empty collection.

9.6.23. TakeLast

The `takeLast` enumeration method returns a specified number of contiguous elements from the end.

The enumeration method takes a single size (non-lambda) expression that returns an `Integer` value.

An expression example with scalar values:

```
{1, 2, 3}.takeLast(2)    // Returns {2, 3}
```

The following example returns the last 5 items:

```
select items.takeLast(5) as last5Items from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns an empty collection.

9.6.24. TakeWhile

The `takeWhile` enumeration method returns elements from the start as long as a specified condition is true.

The enumeration method has two versions. The first version takes a predicate lambda expression and the second version takes a predicate lambda expression and index for use within the predicate expression. Both versions return elements from the start as long as the specified condition is true.

An expression example with scalar values:

```
{1, 2, 3}.takeWhile(v => v < 3)    // Returns {1, 2}
{1, 2, 3}.takeWhile((v,ind) => ind > 2)    // Returns {1, 2}
{1, 2, -1, 4, 5, 6}.takeWhile((v,ind) => ind < 5 and v > 0)    // Returns {1, 2} (Take while index<5 and v>0)
```

This example selects all items from a location report in the order provided until the first item that has a distance to center greater than 20 units:

```
select items.takeWhile(i => distance(i.location.x, i.location.y, 0, 0) < 20)
from LocationReport
```

The second version of the `where` represents the index of the input element starting at zero for the first element.

The next example is similar to the query above but also limits the result to the first 10 items:

```
select items.takeWhile((i, ind) => distance(i.location.x, i.location.y, 0, 0) < 20) and ind < 10)
from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns an empty collection.

9.6.25. TakeWhileLast

The `takeWhileLast` enumeration method returns elements from the end as long as a specified condition is true.

The enumeration method has two versions. The first version takes a predicate lambda expression and the second version takes a predicate lambda expression and index for use within the predicate expression. Both versions return elements from the end as long as the specified condition is true.

An expression example with scalar values:

```
{1, 2, 3}.takeWhileLast(v => v < 3)    // Returns {} (empty collection)
{1, 2, 3}.takeWhileLast(v => v > 1)    // Returns {2, 3}
{1, 2, 3}.takeWhileLast((v,ind) => ind > 2)    // Returns {2, 3}
{1, 2, -1, 4, 5, 6}.takeWhileLast((v,ind) => ind < 5 and v > 0)    // Returns {4, 5, 6} (Take while index<5 and v>0)
```

This example selects all items from a location report, starting from the last element and proceeding backwards, until the first item that has a distance to center greater than 20 units:

```
select items.takeWhileLast(i => distance(i.location.x, i.location.y, 0, 0) < 20)
from LocationReport
```


The second version provides the index of the input element starting at zero for the last element (reverse index).

The next example is similar to the query above but also limits the result to the last 10 items:

```
select items.takeWhile((i, ind) => distance(i.location.x, i.location.y, 0, 0) < 20) and ind < 10)
from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns an empty collection.

9.6.26. ToMap

The `toMap` enumeration method returns a Map according to specified key-selector lambda expression and value-selector lambda expression.

The enumeration method takes a key-selector expression and a value-selector expression. For each element the method applies the key-selector expression to determine the map key and the value-selector expression to determine the map value. If the key already exists in the map the value is overwritten.

The next example EPL outputs a map of item asset id and distance to center for each item:

```
select items.toMap(k => k.assetId, v => distance(v.location.x, v.location.y, 0, 0)) as assetDistance
from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns an empty map.

9.6.27. Union

The `union` enumeration method forms a union of the input elements with the elements that the parameter expression yields.

The enumeration method takes a single parameter that must itself return a collection of events (input), objects or scalar values. It appends the collection to the input elements and returns the appended collection of elements.

An expression example with scalar values:

```
{1, 2, 3}.union({4, 5})    // Returns {1, 2, 3, 4, 5}
```

This example selects a union of all items that have an asset id of L001 or that are of type passenger:

```
select items.where(i => i.assetId = 'L001')
      .union(items.where(i => i.type = 'P')) as itemsUnion
from LocationReport
```

If the input is null the method returns null.

9.6.28. Where

The `where` enumeration method filters elements based on a predicate.

The enumeration method has two versions. The first version takes a predicate lambda expression and the second version takes a predicate lambda expression and index for use within the predicate expression. Both version returns all elements for which the predicate expression is true.

An expression example with scalar values:

```
{1, 2, 3}.where(v => v != 2)    // Returns {1, 3}
```

This example selects all items from a location report that are passenger-type:

```
select items.where(p => p.type = 'P') from LocationReport
```

The second version of the `where` represents the index of the input element starting at zero for the first element.

The example below selects all items from a location report that are passenger-type but ignores the first 3 elements:

```
select items.where((p, ind) => p.type = 'P' and ind > 2) from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns an empty collection.

Chapter 10. EPL Reference: Date-Time Methods

10.1. Overview

EPL date-time methods work on date-time values to perform common tasks such as comparing times and time periods, adding or subtracting time periods, setting or rounding calendar fields and querying fields.

Date-time methods operate on:

1. Any expression or event property that returns either a long-type millisecond value, a `java.util.Calendar` or `java.util.Date` including subclasses. Consider the built-in single-row function `current_timestamp` for use with date-time methods.
2. Any event for which the event type declares a start timestamp property name and optionally also an end timestamp property name. Date-time methods operate on events by means of the *stream-alias.method-name* syntax.

The below table summarizes the built-in date-time methods available:

Table 10.1. Date-Time Methods

Method	Result
<code>after(event or timestamp)</code>	Returns true if an event happens after another event, or a timestamp is after another timestamp. Section 10.4.1, “After”.
<code>before(event or timestamp)</code>	Returns true if an event happens before another event, or a timestamp is before another timestamp. Section 10.4.2, “Before”.
<code>between(timestamp, timestamp, boolean, boolean)</code>	Returns true if a timestamp is between two timestamps. Section 10.3.1, “Between”.
<code>coincides(event or timestamp)</code>	Returns true if an event and another event happen at the same time, or two timestamps are the same value. Section 10.4.3, “Coincides”.
<code>during(event or timestamp)</code>	Returns true if an event happens during the occurrence of another event, or when a timestamps falls within the occurrence of an event. Section 10.4.4, “During”.
<code>finishes(event or timestamp)</code>	Returns true if an event starts after another event starts and the event ends at the same time as the other event.

Method	Result
	Section 10.4.5, “Finishes”.
finishedBy(event or timestamp)	<p>Returns true if an event starts before another event starts and ends at the same time as the other event.</p> <p>Section 10.4.6, “Finished By”.</p>
format()	<p>Formats the date-time returning a string.</p> <p>Section 10.3.2, “Format”.</p>
get(field)	<p>Returns the value of the given date-time value field.</p> <p>Section 10.3.3, “Get (By Field)”.</p>
getMillisOfSecond() getSecondOfMinute() getMinuteOfHour() getHourOfDay() getDayOfWeek() getDayOfMonth() getDayOfYear() getWeekyear() getMonthOfYear() getYear() getEra()	<p>Returns the value of the given date-time value field.</p> <p>Section 10.3.4, “Get (By Name) ”.</p>
includes(event or timestamp)	<p>Returns true if the parameter event happens during the occurrence of the event, or when a timestamps falls within the occurrence of an event.</p> <p>Section 10.4.7, “Includes”.</p>
meets(event or timestamp)	<p>Returns true if the event's end time is the same as another event's start time.</p> <p>Section 10.4.8, “Meets”.</p>
metBy(event or timestamp)	<p>Returns true if the event's start time is the same as another event's end time.</p> <p>Section 10.4.9, “Met By”.</p>

Method	Result
<code>minus(duration-millis)</code>	<p>Returns a date-time with the specified duration in long-type milli-seconds taken away.</p> <p>Section 10.3.5, “Minus”.</p>
<code>minus(time-period)</code>	<p>Returns a date-time with the specified duration in time-period syntax taken away.</p> <p>Section 10.3.5, “Minus”.</p>
<code>overlaps(event or timestamp)</code>	<p>Returns true if the event starts before another event starts and finishes after the other event starts, but before the other event finishes (events have an overlapping period of time).</p> <p>Section 10.4.10, “Overlaps”.</p>
<code>overlappedBy(event or timestamp)</code>	<p>Returns true if the parameter event starts before the input event starts and the parameter event finishes after the input event starts, but before the input event finishes (events have an overlapping period of time).</p> <p>Section 10.4.11, “Overlapped By”.</p>
<code>plus(duration-millis)</code>	<p>Returns a date-time with the specified duration in long-type milli-seconds added.</p> <p>Section 10.3.6, “Plus”.</p>
<code>plus(time-period)</code>	<p>Returns a date-time with the specified duration in time-period syntax added.</p> <p>Section 10.3.6, “Plus”.</p>
<code>roundCeiling(field)</code>	<p>Returns a date-time rounded to the highest whole unit of the date-time field.</p> <p>Section 10.3.7, “RoundCeiling”.</p>
<code>roundFloor(field)</code>	<p>Returns a date-time rounded to the lowest whole unit of the date-time field.</p> <p>Section 10.3.8, “RoundFloor”.</p>
<code>roundHalf(field)</code>	<p>Returns a date-time rounded to the nearest whole unit of the date-time field.</p> <p>Section 10.3.9, “RoundHalf”.</p>
<code>set(field, value)</code>	<p>Returns a date-time with the specified field set to the value returned by a value expression.</p>

Method	Result
	Section 10.3.10, “Set (By Field)”.
<code>starts(event or timestamp)</code>	<p>Returns true if an event and another event start at the same time and the event's end happens before the other event's end.</p> <p>Section 10.4.12, “Starts”.</p>
<code>startedBy(event or timestamp)</code>	<p>Returns true if an event and another event start at the same time and the other event's end happens before the input event's end.</p> <p>Section 10.4.13, “Started By”.</p>
<code>withDate(year,month,day)</code>	<p>Returns a date-time with the specified date, retaining the time fields.</p> <p>Section 10.3.11, “WithDate”.</p>
<code>withMax(field)</code>	<p>Returns a date-time with the field set to the maximum value for the field.</p> <p>Section 10.3.12, “WithMax”.</p>
<code>withMin(field)</code>	<p>Returns a date-time with the field set to the minimum value for the field.</p> <p>Section 10.3.13, “WithMin”.</p>
<code>withTime(hour,minute,sec,msec)</code>	<p>Returns a date-time with the specified time, retaining the date fields.</p> <p>Section 10.3.14, “WithTime”.</p>
<code>toCalendar()</code>	<p>Returns the <code>Calendar</code> object for this date-time value.</p> <p>Section 10.3.15, “ToCalendar”.</p>
<code>toDate()</code>	<p>Returns the <code>Date</code> object for this date-time value.</p> <p>Section 10.3.16, “ToDate”.</p>
<code>toMillisec()</code>	<p>Returns the long-type milliseconds value for this date-time value.</p> <p>Section 10.3.17, “ToMillisec”.</p>

10.2. How to Use

10.2.1. Syntax

The syntax for date-time methods is the same syntax as for any chained invocation:

```
input_val.datetime_method_name( [method_parameter [, method_parameter [...]]])
.[ datetime_method_name(...) [...]]
```

Following the *input_val* input value is the . (dot) operator and the *datetime_method_name* date-time method name. It follows in parenthesis a comma-separated list of method parameter expressions. Additional date-time methods can be chained thereafter.

The input value can be any expression or event property that returns a value of type long or `java.util.Calendar` or `java.util.Date`. If the input value is null, the expression result is also null.

The input value can also be an event. In this case the event type of the event must have the start timestamp property name defined and optionally also the end timestamp property name.

The following example EPL statement employs the `withTime` date-time method. This example returns the current engine time with the time-part set to 1 am:

```
select current_timestamp.withTime(1, 0, 0, 0) as timelam from MyEvent
```

As date-time methods can be chained, this EPL is equivalent:

```
select current_timestamp.set('hour', 1).set('min', 0).set('sec', 0).set('msec', 0) as timelam
from MyEvent
```

The statement above outputs in field `timelam` a long-type millisecond-value reflecting 1am on the same date as engine time. Since the input value is provided by the built-in `current_timestamp` function which returns current engine date as a long-type millisecond value the output is also a long-type millisecond value.

You may apply a date-time method to an event property.

Assume that the `RFIDEvent` event type has a Date-type property by name `timeTaken`. The following query rounds each time-taken value down to the nearest minute and outputs a Date-type value in column `timeTakenRounded`:

```
select timeTaken.roundFloor('min') as timeTakenRounded from RFIDEvent
```

You may apply a date-time method to events. This example assumes that the `RFIDEvent` and `WifiEvent` event types both have a timestamp property defined. The EPL compares the timestamps of the `RFIDEvent` and the `WifiEvent`:

```
select rfid.after(wifi) as isAfter
from RFIDEvent.std:lastevent() rfid, WifiEvent.std:lastevent() wifi
```

For comparing date-time values and considering event duration (event start and end timestamps) we recommend any of the interval algebra methods. You may also compare millisecond values using the `between` or `in` ranges and inverted ranges or relational operators (`>`, `<`, `>=`, `<=`).

From a performance perspective, the date-time method evaluation ensures that for each unique chain of date-time methods only a single calendar objects is copied or created when necessary.

10.3. Calendar and Formatting Reference

10.3.1. Between

The `between` date-time method compares the input date-time value to the two date-time values passed in and returns true if the input value falls between the two parameter values.

The synopsis is:

```
input_val.between(range_start, range_end [, include_start, include_end])
```

The method takes either 2 or 4 parameters. The first two parameters *range_start* and *range_end* are expressions or properties that yield either a long-typed, Date-typed or Calendar-typed range start and end value.

The next two parameters *include_start* and *include_end* are optional. If not specified, the range start value and range end value are included in the range i.e. specify a closed range where both endpoints are included. If specified, the expressions must return a boolean-value indicating whether to include the range start value and range end value in the range.

The example below outputs true when the time-taken property value of the RFID event falls between the time-start property value and the time-end property value (closed range includes endpoints):

```
select timeTaken.between(timeStart, timeEnd) from RFIDEvent
```

The example below performs the same test as above but does not include endpoints (open range includes neither endpoint):

```
select timeTaken.between(timeStart, timeEnd, false, false) from RFIDEvent
```

If the range end value is less then the range start value, the algorithm reverses the range start and end value.

If the input date-time value or any of the parameter values evaluate to null the method returns a null result value.

10.3.2. Format

The `format` date-time method formats the date-time returning a string.

The method takes no parameters. It returns the date-time value formatted using the default locale format obtained from `new SimpleDateFormat()`.

The example below outputs the time-taken property value of the RFID event:

```
select timeTaken.format() as timeTakenStr from RFIDEvent
```

10.3.3. Get (By Field)

The `get` date-time method returns the value of the given date-time value field.

The method takes a single string-constant field name as parameter. Please see Section 4.2.1, “Specifying Time Periods” for a list of recognized keywords (not case-sensitive).

The method returns the numeric value of the field within the date-time value. The value returned adheres to Calendar-class semantics: For example, the value for `month` starts at zero and has a maximum of 11.

The example below outputs the month value of the time-taken property value of the RFID event:

```
select timeTaken.get('month') as timeTakenMonth from RFIDEvent
```


10.3.4. Get (By Name)

The following list of getter-methods are available: `getMillisOfSecond()`, `getSecondOfMinute()`, `getMinuteOfHour()`, `getHourOfDay()`, `getDayOfWeek()`, `getDayOfMonth()`, `getDayOfYear()`, `getWeekYear()`, `getMonthOfYear()`, `getYear()` and `getEra()`.

All get-methods take no parameter and return the numeric value of the field within the date-time value. The value returned adheres to `Calendar`-class semantics: For example, the value for `month` starts at zero and has a maximum of 11.

The example below outputs the month value of the time-taken property value of the RFID event:

```
select timeTaken.getMonthOfYear() as timeTakenMonth from RFIDEvent
```

10.3.5. Minus

The `minus` date-time method returns a date-time with the specified duration taken away.

The method has two versions: The first version takes the duration as a long-type millisecond value. The second version takes the duration as a time-period expression, see Section 4.2.1, “Specifying Time Periods”.

The example below demonstrates the time-period parameter to subtract two minutes from the time-taken property value of the RFID event:

```
select timeTaken.minus(2 minutes) as timeTakenMinus2Min from RFIDEvent
```

The next example is equivalent but passes a millisecond-value instead:

```
select timeTaken.minus(2*60*1000) as timeTakenMinus2Min from RFIDEvent
```

10.3.6. Plus

The `plus` date-time method returns a date-time with the specified duration added.

The method has two versions: The first version takes the duration as a long-type millisecond value. The second version takes the duration as a time-period expression, see Section 4.2.1, “Specifying Time Periods”.

The next example adds two minutes to the time-taken property value of the RFID event:

```
select timeTaken.plus(2 minutes) as timeTakenPlus2Min from RFIDEvent
```

The next example is equivalent but passes a millisecond-value instead:

```
select timeTaken.plus(2*60*1000) as timeTakenPlus2Min from RFIDEvent
```

10.3.7. RoundCeiling

The `roundCeiling` date-time method rounds to the highest whole unit of the date-time field.

The method takes a single string-constant field name as parameter. Please see Section 4.2.1, “Specifying Time Periods” for a list of recognized keywords (not case-sensitive).

The next example rounds-to-ceiling the minutes of the time-taken property value of the RFID event:

```
select timeTaken.roundCeiling('min') as timeTakenRounded from RFIDEvent
```

If the input time is 2002-05-30 09:01:23.050, for example, the output is 2002-05-30 09:02:00.000 (example timestamps are in format yyyy-MM-dd HH:mm:ss.SSS).

10.3.8. RoundFloor

The `roundFloor` date-time method rounds to the lowest whole unit of the date-time field.

The method takes a single string-constant field name as parameter. Please see Section 4.2.1, “Specifying Time Periods” for a list of recognized keywords (not case-sensitive).

The next example rounds-to-floor the minutes of the time-taken property value of the RFID event:

```
select timeTaken.roundFloor('min') as timeTakenRounded from RFIDEvent
```

If the input time is 2002-05-30 09:01:23.050, for example, the output is 2002-05-30 09:01:00.000 (example timestamps are in format yyyy-MM-dd HH:mm:ss.SSS).

10.3.9. RoundHalf

The `roundFloor` date-time method rounds to the nearest whole unit of the date-time field.

The method takes a single string-constant field name as parameter. Please see Section 4.2.1, “Specifying Time Periods” for a list of recognized keywords (not case-sensitive).

The next example rounds the minutes of the time-taken property value of the RFID event:

```
select timeTaken.roundHalf('min') as timeTakenRounded from RFIDEvent
```

The following table provides a few examples of the rounding (example timestamps are in format yyyy-MM-dd HH:mm:ss.SSS):

Table 10.2. RoundHalf Examples

Input	Output
2002-05-30 09:01:23.050	2002-05-30 09:01:00.000
2002-05-30 09:01:29.999	2002-05-30 09:01:00.000
2002-05-30 09:01:30.000	2002-05-30 09:02:00.000

10.3.10. Set (By Field)

The `set` date-time method returns a date-time with the specified field set to the value returned by an expression.

The method takes a string-constant field name and an expression returning an integer-value as parameters. Please see Section 4.2.1, “Specifying Time Periods” for a list of recognized keywords (not case-sensitive).

The method returns the new date-time value with the field set to the provided value. Note that value adheres to Calendar-class semantics: For example, the value for `month` starts at zero and has a maximum of 11.

The example below outputs the time-taken with the value for `month` set to April:

```
select timeTaken.set('month', 3) as timeTakenMonth from RFIDEvent
```

10.3.11. WithDate

The `withDate` date-time method returns a date-time with the specified date, retaining the time fields.

The method takes three expressions as parameters: An expression for year, month and day.

The method returns the new date-time value with the date fields set to the provided values. For expressions returning null the method ignores the field for which null is returned. Note the Calendar-class semantics: For example, the value for `month` starts at zero and has a maximum of 11.

The example below outputs the time-taken with the date set to May 30, 2002:

```
select timeTaken.withDate(2002, 4, 30) as timeTakenDated from RFIDEvent
```

10.3.12. WithMax

The `withMax` date-time method returns a date-time with the field set to the maximum value for the field.

The method takes a string-constant field name as parameter. Please see Section 4.2.1, “Specifying Time Periods” for a list of recognized keywords (not case-sensitive).

The method returns the new date-time value with the specific date field set to the maximum value.

The example below outputs the time-taken property value with the second-part as 59 seconds:

```
select timeTaken.withMax('sec') as timeTakenMaxSec from RFIDEvent
```

10.3.13. WithMin

The `withMin` date-time method returns a date-time with the field set to the minimum value for the field.

The method takes a string-constant field name as parameter. Please see Section 4.2.1, “Specifying Time Periods” for a list of recognized keywords (not case-sensitive).

The method returns the new date-time value with the specific date field set to the minimum value.

The example below outputs the time-taken property value with the second-part as 0 seconds:

```
select timeTaken.withMin('sec') as timeTakenMaxSec from RFIDEvent
```

10.3.14. WithTime

The `withTime` date-time method returns a date-time with the specified time, retaining the date fields.

The method takes four expressions as parameters: An expression for hour, minute, second and millisecond.

The method returns the new date-time value with the time fields set to the provided values. For expressions returning null the method ignores the field for which null is returned.

The example below outputs the time-taken with the time set to 9am:

```
select timeTaken.withTime(9, 0, 0, 0) as timeTakenDated from RFIDEvent
```

10.3.15. ToCalendar

The `toCalendar` date-time method returns the `Calendar` object for this date-time value.

The method takes no parameters.

The example below outputs the time-taken as a `Calendar` object:

```
select timeTaken.toCalendar() as timeTakenCal from RFIDEvent
```

10.3.16. ToDate

The `toDate` date-time method returns the `Date` object for this date-time value.

The method takes no parameters.

The example below outputs the time-taken as a `Date` object:

```
select timeTaken.toDate() as timeTakenDate from RFIDEvent
```

10.3.17. ToMillisec

The `toMillisec` date-time method returns the long-typed millisecond value for this date-time value.

The method takes no parameters.

The example below outputs the time-taken as a long-typed millisecond value:

```
select timeTaken.toMillisec() as timeTakenLong from RFIDEvent
```

10.4. Interval Algebra Reference

Interval algebra methods compare start and end timestamps of events or timestamps in general.

When the expression input is only a timestamp value, such as a long-type value or a `Date` or `Calendar` object, the start and end timestamp represented by that value are the same timestamp value.

When expression input is an event stream alias, the engine determine the event type for the stream. If the event type declares a start timestamp property name, the engine uses that start timestamp property to determine the start timestamp for the event. If the event type also declares an end timestamp property name, the engine uses that end timestamp property to determine the end timestamp for the event (i.e. an event with duration). If an end timestamp property name is not declared, the start and end timestamp for each event is the same value and the event is considered to have zero duration (i.e. a point-in-time event).

Interval algebra methods all return `Boolean`-type value. When the input value start timestamp is null, or the end timestamp (if declared for the event type) is null or any of the start timestamp and end timestamp (if declared for the event type) values of the first parameter is null, the result value is null.

Examples

The examples in this section simply use `A` and `B` as event type names. The alias `a` is used to represent `A`-type events and respectively the alias `b` represents `B`-type events.

The `create-schema` for types `A` and `B` is shown next. The two types are declared the same. The example declares the property providing start timestamp values as `startts` and the property providing end timestamp values as `endts`:

```
create schema A as (startts long, endts long) starttimestamp 'startts' endtimestamp 'endts'
```

```
create schema B as (startts long, endts long) starttimestamp 'startts' endtimestamp 'endts'
```

The sample EPL below joins the last `A` and the last `B` event. It detects `A-B` event combinations for which, when comparing timestamps, the last `A` event that occurs before the last `B` event. The example employs the `before` method:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.before(b)
```

For simplicity, the examples in this section refer to `A` and the alias `a` as the input event. The examples refer to `B` and the alias `b` as the parameter event.

Interval Algebra Parameters

The first parameter of each interval algebra methods is the event or timestamp to compare to.

All remaining parameters to interval algebra methods are intervals and can be any of the following:

1. A constant, an event property or more generally any expression returning a numeric value that is the number of seconds. For example, in the expression `a.before(b, 2)` the parameter `2` is interpreted to mean 2 seconds. The expression `a.before(b, myIntervalProperty)` is interpreted to mean `myIntervalProperty` seconds.
2. A time period expression as described in Section 10.4.7, “Includes”. For example: `a.before(b, 1 hour 2 minutes)`.

When an interval parameter is provided and is null, the method result value is null.

Performance

The engine analyzes interval algebra methods as well as the `between` date-time method in the `where`-clause and builds a query plan for execution of joins and subqueries. The query plan can include hash and btree index lookups using the start and end timestamps as computed by expressions or provided by events as applicable. Consider turning on query plan logging to obtain information on the query plan used.

The query planning is generally most effective when no additional thresholds or ranges are provided to interval algebra methods, as the query planner may not consider an interval algebra method that it cannot plan.

The query planner may also not optimally plan the query execution if events or expressions return different types of date representation. Query planning works best if all date representations use the same long, Date or

Calendar types.

Limitations

Date-time method that change date or time fields, such as `withTime`, `withDate`, `set` or `round` methods set the end timestamp to the start timestamp.

For example, in the following expression the parameter to the `after` method has a zero duration, and not the end timestamp that the event `B` `ends` property provides.

```
a.after(b.withTime(9, 0, 0, 0))
```

10.4.1. After

The `after` date-time method returns true if an event happens after another event, or a timestamp is after another timestamp.

The method compares the input value's start timestamp (`a.startTimestamp`) to the first parameter's end timestamp (`b.endTimestamp`) to determine whether A happens after B.

If used with one parameter, for example in `a.after(b)`, the method returns true if A starts after B ends.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.after(b)
// Above matches when:
//   a.startTimestamp - b.endTimestamp > 0
```

If providing two parameters, for example in `a.after(b, 5 sec)`, the method returns true if A starts at least 5 seconds after B ends.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.after(b, 5 sec)
// Above matches when:
//   a.startTimestamp - b.endTimestamp >= 5 seconds
```

If providing three parameters, for example in `a.after(b, 5 sec, 10 sec)`, the method returns true if A starts at least 5 seconds but no more then 10 seconds after B ends.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.after(b, 5 sec, 10 sec)
// Above matches when:
//   5 seconds <= a.startTimestamp - b.endTimestamp <= 10 seconds
```

Negative values for the range are allowed. For example in `a.after(b, -5 sec, -10 sec)`, the method returns true if A starts at least 5 seconds but no more then 10 seconds before B ends.

If the range low endpoint is greater than the range high endpoint, the engine automatically reverses them. Thus `a.after(b, 10 sec, 5 sec)` is the same semantics as `a.after(b, 5 sec, 10 sec)`.

10.4.2. Before

The `before` date-time method returns true if an event happens before another event, or a timestamp is before

another timestamp.

The method compares the input value's end timestamp (`a.endTimestamp`) and the first parameter's start timestamp (`b.startTimestamp`) to determine whether A happens before B.

If used with one parameter, for example in `a.before(b)`, the method returns true if A ends before B starts.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.before(b)
// Above matches when:
//   b.startTimestamp - a.endTimestamp > 0
```

If providing two parameters, for example in `a.before(b, 5 sec)`, the method returns true if A ends at least 5 seconds before B starts.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.before(b, 5 sec)
// Above matches when:
//   b.startTimestamp - a.endTimestamp >= 5 seconds
```

If providing three parameters, for example in `a.before(b, 5 sec, 10 sec)`, the method returns true if A ends at least 5 seconds but no more than 10 seconds before B starts.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.before(b, 5 sec, 10 sec)
// Above matches when:
//   5 seconds <= b.startTimestamp - a.endTimestamp <= 10 seconds
```

Negative values for the range are allowed. For example in `a.before(b, -5 sec, -10 sec)`, the method returns true if A starts at least 5 seconds but no more than 10 seconds after B starts.

If the range low endpoint is greater than the range high endpoint, the engine automatically reverses them. Thus `a.before(b, 10 sec, 5 sec)` is the same semantics as `a.before(b, 5 sec, 10 sec)`.

10.4.3. Coincides

The `coincides` date-time method returns true if an event and another event happen at the same time, or two timestamps are the same value.

The method compares the input value's start and end timestamp with the first parameter's start and end timestamp and determines if they equal.

If used with one parameter, for example in `a.coincides(b)`, the method returns true if the start timestamp of A and B are the same and the end timestamps of A and B are also the same.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.coincides(b)
// Above matches when:
//   a.startTimestamp = b.startTimestamp and a.endTimestamp = b.endTimestamp
```

If providing two parameters, for example in `a.coincides(b, 5 sec)`, the method returns true if the difference between the start timestamps of A and B is equal to or less than 5 seconds and the difference between the end timestamps of A and B is also equal to or less than 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.coincides(b, 5 sec)
// Above matches when:
//   abs(a.startTimestamp - b.startTimestamp) <= 5 sec and
//   abs(a.endTimestamp - b.endTimestamp) <= 5 sec
```

If providing three parameters, for example in `a.coincides(b, 5 sec, 10 sec)`, the method returns true if the difference between the start timestamps of A and B is equal to or less than 5 seconds and the difference between the end timestamps of A and B is equal to or less than 10 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.coincides(b, 5 sec, 10 sec)
// Above matches when:
//   abs(a.startTimestamp - b.startTimestamp) <= 5 seconds and
//   abs(a.endTimestamp - b.endTimestamp) <= 10 seconds
```

A negative value for interval parameters is not allowed. If your interval parameter is itself an expression that returns a negative value the engine logs a warning message and returns null.

10.4.4. During

The `during` date-time method returns true if an event happens during the occurrence of another event, or when a timestamp falls within the occurrence of an event..

The method determines whether the input value's start and end timestamp are during the first parameter's start and end timestamp. The symmetrical opposite is Section 10.4.7, “Includes”.

If used with one parameter, for example in `a.during(b)`, the method returns true if the start timestamp of A is after the start timestamp of B and the end timestamp of A is before the end timestamp of B.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.during(b)
// Above matches when:
//   b.startTimestamp < a.startTimestamp <= a.endTimestamp < b.endTimestamp
```

If providing two parameters, for example in `a.during(b, 5 sec)`, the method returns true if the difference between the start timestamps of A and B is equal to or less than 5 seconds and the difference between the end timestamps of A and B is also equal to or less than 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.during(b, 5 sec)
// Above matches when:
//   0 < a.startTimestamp - b.startTimestamp <= 5 sec and
//   0 < a.endTimestamp - b.endTimestamp <= 5 sec
```

If providing three parameters, for example in `a.during(b, 5 sec, 10 sec)`, the method returns true if the difference between the start timestamps of A and B and the difference between the end timestamps of A and B is between 5 and 10 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.during(b, 5 sec, 10 sec)
// Above matches when:
```



```
// 5 seconds <= a.startTimestamp - b.startTimestamp <= 10 seconds and
// 5 seconds <= a.endTimestamp - b.endTimestamp <= 10 seconds
```

If providing five parameters, for example in `a.during(b, 5 sec, 10 sec, 20 sec, 30 sec)`, the method returns true if the difference between the start timestamps of A and B is between 5 seconds and 10 seconds and the difference between the end timestamps of A and B is between 20 seconds and 30 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b
  where a.during(b, 5 sec, 10 sec, 20 sec, 30 sec)
// Above matches when:
// 5 seconds <= a.startTimestamp - b.startTimestamp <= 10 seconds and
// 20 seconds < a.endTimestamp - b.endTimestamp <= 30 seconds
```

10.4.5. Finishes

The `finishes` date-time method returns true if an event starts after another event starts and the event ends at the same time as the other event.

The method determines whether the input value's start timestamp is after the first parameter's start timestamp and the end timestamp of the input value and the first parameter are the same. The symmetrical opposite is Section 10.4.6, “Finished By”.

If used with one parameter, for example in `a.finishes(b)`, the method returns true if the start timestamp of A is after the start timestamp of B and the end timestamp of A and B are the same.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.finishes(b)
// Above matches when:
// b.startTimestamp < a.startTimestamp and a.endTimestamp = b.endTimestamp
```

If providing two parameters, for example in `a.finishes(b, 5 sec)`, the method returns true if the start timestamp of A is after the start timestamp of B and the difference between the end timestamps of A and B is equal to or less than 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.finishes(b, 5 sec)
// Above matches when:
// b.startTimestamp < a.startTimestamp and
// abs(a.endTimestamp - b.endTimestamp ) <= 5 seconds
```

A negative value for interval parameters is not allowed. If your interval parameter is itself an expression that returns a negative value the engine logs a warning message and returns null.

10.4.6. Finished By

The `finishedBy` date-time method returns true if an event starts before another event starts and the event ends at the same time as the other event.

The method determines whether the input value's start timestamp happens before the first parameter's start timestamp and the end timestamp of the input value and the first parameter are the same. The symmetrical opposite is Section 10.4.5, “Finishes”.

If used with one parameter, for example in `a.finishedBy(b)`, the method returns true if the start timestamp of A is before the start timestamp of B and the end timestamp of A and B are the same.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.finishedBy(b)
// Above matches when:
//   a.startTimestamp < b.startTimestamp and a.endTimestamp = b.endTimestamp
```

If providing two parameters, for example in `a.finishedBy(b, 5 sec)`, the method returns true if the start timestamp of A is before the start timestamp of B and the difference between the end timestamps of A and B is equal to or less than 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.finishedBy(b, 5 sec)
// Above matches when:
//   a.startTimestamp < b.startTimestamp and
//   abs(a.endTimestamp - b.endTimestamp) <= 5 seconds
```

10.4.7. Includes

The `includes` date-time method returns true if the parameter event happens during the occurrence of the input event, or when a timestamps falls within the occurrence of an event.

The method determines whether the first parameter's start and end timestamp are during the input value's start and end timestamp. The symmetrical opposite is Section 10.4.4, “During”.

If used with one parameter, for example in `a.includes(b)`, the method returns true if the start timestamp of B is after the start timestamp of A and the end timestamp of B is before the end timestamp of A.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.includes(b)
// Above matches when:
//   a.startTimestamp < b.startTimestamp <= b.endTimestamp < a.endTimestamp
```

If providing two parameters, for example in `a.includes(b, 5 sec)`, the method returns true if the difference between the start timestamps of A and B is equal to or less than 5 seconds and the difference between the end timestamps of A and B is also equal to or less than 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.includes(b, 5 sec)
// Above matches when:
//   0 < b.startTimestamp - a.startTimestamp <= 5 sec and
//   0 < a.endTimestamp - b.endTimestamp <= 5 sec
```

If providing three parameters, for example in `a.includes(b, 5 sec, 10 sec)`, the method returns true if the difference between the start timestamps of A and B and the difference between the end timestamps of A and B is between 5 and 10 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.includes(b, 5 sec, 10 sec)
// Above matches when:
//   5 seconds <= a.startTimestamp - b.startTimestamp <= 10 seconds and
```

```
// 5 seconds <= a.endTimestamp - b.endTimestamp <= 10 seconds
```

If providing five parameters, for example in `a.includes(b, 5 sec, 10 sec, 20 sec, 30 sec)`, the method returns true if the difference between the start timestamps of A and B is between 5 seconds and 10 seconds and the difference between the end timestamps of A and B is between 20 seconds and 30 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b
  where a.includes(b, 5 sec, 10 sec, 20 sec, 30 sec)
// Above matches when:
// 5 seconds <= a.startTimestamp - b.startTimestamp <= 10 seconds and
// 20 seconds <= a.endTimestamp - b.endTimestamp <= 30 seconds
```

10.4.8. Meets

The `meets` date-time method returns true if the event's end time is the same as another event's start time.

The method compares the input value's end timestamp and the first parameter's start timestamp and determines whether they equal.

If used with one parameter, for example in `a.meets(b)`, the method returns true if the end timestamp of A is the same as the start timestamp of B.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.meets(b)
// Above matches when:
// a.endTimestamp = b.startTimestamp
```

If providing two parameters, for example in `a.meets(b, 5 sec)`, the method returns true if the difference between the end timestamp of A and the start timestamp of B is equal to or less than 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.meets(b, 5 sec)
// Above matches when:
// abs(b.startTimestamp - a.endTimestamp) <= 5 seconds
```

A negative value for the interval parameter is not allowed. If your interval parameter is itself an expression that returns a negative value the engine logs a warning message and returns null.

10.4.9. Met By

The `metBy` date-time method returns true if the event's start time is the same as another event's end time.

The method compares the input value's start timestamp and the first parameter's end timestamp and determines whether they equal.

If used with one parameter, for example in `a.metBy(b)`, the method returns true if the start timestamp of A is the same as the end timestamp of B.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.metBy(b)
// Above matches when:
```

```
// a.startTimestamp = b.endTimestamp
```

If providing two parameters, for example in `a.metBy(b, 5 sec)`, the method returns true if the difference between the end timestamps of B and the start timestamp of A is equal to or less than 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.metBy(b, 5 sec)
// Above matches when:
// abs(a.startTimestamp - b.endTimestamp) <= 5 seconds
```

A negative value for the interval parameter is not allowed. If your interval parameter is itself an expression that returns a negative value the engine logs a warning message and returns null.

10.4.10. Overlaps

The `overlaps` date-time method returns true if the event starts before another event starts and finishes after the other event starts, but before the other event finishes (events have an overlapping period of time).

The method determines whether the input value's start and end timestamp indicate an overlap with the first parameter's start and end timestamp, such that A starts before B starts and A ends after B started but before B ends.

If used with one parameter, for example in `a.overlaps(b)`, the method returns true if the start timestamp of A is before the start timestamp of B and the end timestamp of A is before the end timestamp of B.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.overlaps(b)
// Above matches when:
// a.startTimestamp < b.startTimestamp < a.endTimestamp < b.endTimestamp
```

If providing two parameters, for example in `a.overlaps(b, 5 sec)`, the method returns true if, in addition, the difference between the end timestamp of A and the start timestamp of B is equal to or less than 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.overlaps(b, 5 sec)
// Above matches when:
// a.startTimestamp < b.startTimestamp < a.endTimestamp < b.endTimestamp and
// 0 <= a.endTimestamp - b.startTimestamp <= 5 seconds
```

If providing three parameters, for example in `a.overlaps(b, 5 sec, 10 sec)`, the method returns true if, in addition, the difference between the end timestamp of A and the start timestamp of B is between 5 and 10 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.overlaps(b, 5 sec, 10 sec)
// Above matches when:
// a.startTimestamp < b.startTimestamp < a.endTimestamp < b.endTimestamp and
// 5 seconds <= a.endTimestamp - b.startTimestamp <= 10 seconds
```

10.4.11. Overlapped By

The `overlappedBy` date-time method returns true if the parameter event starts before the input event starts and the parameter event finishes after the input event starts, but before the input event finishes (events have an overlapping period of time).

The method determines whether the input value's start and end timestamp indicate an overlap with the first parameter's start and end timestamp, such that B starts before A starts and B ends after A started but before A ends.

If used with one parameter, for example in `a.overlappedBy(b)`, the method returns true if the start timestamp of B is before the start timestamp of A and the end timestamp of B is before the end timestamp of A.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.overlappedBy(b)
// Above matches when:
//   b.startTimestamp < a.startTimestamp < b.endTimestamp < a.endTimestamp
```

If providing two parameters, for example in `a.overlappedBy(b, 5 sec)`, the method returns true if, in addition, the difference between the end timestamp of B and the start timestamp of A is equal to or less than 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.overlappedBy(b, 5 sec)
// Above matches when:
//   b.startTimestamp < a.startTimestamp < b.endTimestamp < a.endTimestamp and
//   0 <= b.endTimestamp - a.startTimestamp <= 5 seconds
```

If providing three parameters, for example in `a.overlappedBy(b, 5 sec, 10 sec)`, the method returns true if, in addition, the difference between the end timestamp of B and the start timestamp of A is between 5 and 10 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.overlappedBy(b, 5 sec, 10 sec)
// Above matches when:
//   b.startTimestamp < a.startTimestamp < b.endTimestamp < a.endTimestamp and
//   5 seconds <= b.endTimestamp - a.startTimestamp <= 10 seconds
```

10.4.12. Starts

The `starts` date-time method returns true if an event and another event start at the same time and the event's end happens before the other event's end.

The method determines whether the start timestamps of the input value and the first parameter are the same and the end timestamp of the input value is before the end timestamp of the first parameter.

If used with one parameter, for example in `a.starts(b)`, the method returns true if the start timestamp of A and B are the same and the end timestamp of A is before the end timestamp of B.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.starts(b)
// Above matches when:
//   a.startTimestamp = b.startTimestamp and a.endTimestamp < b.endTimestamp
```

If providing two parameters, for example in `a.starts(b, 5 sec)`, the method returns true if the difference between the start timestamps of A and B is between is equal to or less then 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.starts(b, 5 sec)
// Above matches when:
//   abs(a.startTimestamp - b.startTimestamp) <= 5 seconds and
//   a.endTimestamp < b.endTimestamp
```

A negative value for the interval parameter is not allowed. If your interval parameter is itself an expression that returns a negative value the engine logs a warning message and returns null.

10.4.13. Started By

The `startedBy` date-time method returns true if an event and another event start at the same time and the other event's end happens before the input event's end.

The method determines whether the start timestamp of the input value and the first parameter are the same and the end timestamp of the first parameter is before the end timestamp of the input value.

If used with one parameter, for example in `a.startedBy(b)`, the method returns true if the start timestamp of A and B are the same and the end timestamp of B is before the end timestamp of A.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.startedBy(b)
// Above matches when:
//   a.startTimestamp = b.startTimestamp and b.endTimestamp < a.endTimestamp
```

If providing two parameters, for example in `a.startedBy(b, 5 sec)`, the method returns true if the difference between the start timestamps of A and B is between is equal to or less then 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.startedBy(b, 5 sec)
// Above matches when:
//   abs(a.startTimestamp - b.startTimestamp) <= 5 seconds and
//   b.endTimestamp < a.endTimestamp
```

A negative value for the interval parameter is not allowed. If your interval parameter is itself an expression that returns a negative value the engine logs a warning message and returns null.

Chapter 11. EPL Reference: Views

This chapter outlines the views that are built into Esper. All views can be arbitrarily combined as many of the examples below show. The section on Chapter 3, *Processing Model* provides additional information on the relationship of views, filtering and aggregation. Please also see Section 4.4.3, “Specifying Views” for the use of views in the `from` clause with streams, patterns and named windows.

Esper organizes built-in views in namespaces and names. Views that provide sliding or tumbling data windows are in the `win` namespace. Other most commonly used views are in the `std` namespace. The `ext` namespace are views that order events. The `stat` namespace is used for views that derive statistical data.

Esper distinguishes between data window views and derived-value views. Data windows, or data window views, are views that retain incoming events until an expiry policy indicates to release events. Derived-value views derive a new value from event streams and post the result as events of a new type.

Two or more data window views can be combined. This allows a sets of events retained by one data window to be placed into a union or an intersection with the set of events retained by one or more other data windows. Please see Section 4.4.4, “Multiple Data Window Views” for more detail.

The keep-all data window counts as a data window but has no expiry policy: it retains all events received. The grouped-window declaration allocates a new data window per grouping criteria and thereby counts as a data window, but cannot appear alone.

The next table summarizes data window views:

Table 11.1. Built-in Data Window Views

View	Syntax	Description
Length window	<code>win:length(<i>size</i>)</code>	Sliding length window extending the specified number of elements into the past.
Length batch window	<code>win:length_batch(<i>size</i>)</code>	Tumbling window that batches events and releases them when a given minimum number of events has been collected.
Time window	<code>win:time(<i>time period</i>)</code>	Sliding time window extending the specified time interval into the past.
Externally-timed window	<code>win:ext_timed(<i>timestamp expression</i>, <i>time period</i>)</code>	Sliding time window, based on the millisecond time value supplied by an expression.
Time batch window	<code>win:time_batch(<i>time period</i>[<i>,optional reference point</i>] [<i>,flow control</i>])</code>	Tumbling window that batches events and releases them every specified time interval, with flow control options.
Time-Length combination batch window	<code>win:time_length_batch(<i>time period</i>, <i>size</i> [<i>,flow control</i>])</code>	Tumbling multi-policy time and length batch window with flow control options.
Time-Accumulating window	<code>win:time_accum(<i>time period</i>)</code>	Sliding time window accumulates events until no more events arrive

View	Syntax	Description
		within a given time interval.
Keep-All window	win:keepall()	The keep-all data window view simply retains all events.
Sorted window	ext:sort(<i>size</i> , <i>sort criteria</i>)	Sorts by values returned by sort criteria expressions and keeps only the top events up to the given size.
Time-Order View	ext:time_order(<i>timestamp expression</i> , <i>time period</i>)	Orders events that arrive out-of-order, using an expression providing timestamps to be ordered.
Unique	std:unique(<i>unique criteria(s)</i>)	Retains only the most recent among events having the same value for the criteria expression(s). Acts as a length window of size 1 for each distinct expression value.
Grouped Data Window	std:groupwin(<i>grouping criteria(s)</i>)	Groups events into sub-views by the value of the specified expression(s), generally used to provide a separate data window per group.
Last Event	std:lastevent()	Retains the last event, acts as a length window of size 1.
First Event	std:firstevent()	Retains the very first arriving event, disregarding all subsequent events.
First Unique	std:firstunique(<i>unique criteria(s)</i>)	Retains only the very first among events having the same value for the criteria expression(s), disregarding all subsequent events for same value(s).
First Length	win:firstlength(<i>size</i>)	Retains the first <i>size</i> events, disregarding all subsequent events.
First Time	win:firsttime(<i>time period</i>)	Retains the events arriving until the time interval has passed, disregarding all subsequent events.

The table below summarizes views that derive information from received events and present the derived information as an insert and remove stream of events that are typed specifically to carry the result of the computations:

Table 11.2. Built-in Derived-Value Views

View	Syntax	Description
Size	std:size([<i>expression</i> , ...])	Derives a count of the number of events in a data window, or in an insert stream if used without a data window, and optionally provides additional event properties as listed in para-

View	Syntax	Description
		meters.
Univariate statistics	<code>stat:uni(<i>value expression</i> [,<i>expression</i>, ...])</code>	Calculates univariate statistics on the values returned by the expression.
Regression	<code>stat:linest(<i>value expression</i>, <i>value expression</i> [,<i>expression</i>, ...])</code>	Calculates regression on the values returned by two expressions.
Correlation	<code>stat:correl(<i>value expression</i>, <i>value expression</i> [,<i>expression</i>, ...])</code>	Calculates the correlation value on the values returned by two expressions.
Weighted average	<code>stat:weighted_avg(<i>value expression</i>, <i>value expression</i> [,<i>expression</i>, ...])</code>	Calculates weighted average given a weight expression and an expression to compute the average for.

A Note on View Parameters

The syntax for view specifications starts with the namespace name and the name and is followed by optional view parameter expressions in parenthesis:

```
namespace:name(view_parameters)
```

This example specifies a time window of 5 seconds:

```
select * from StockTickEvent.win:time(5 sec)
```

All expressions are allowed as parameters to views, including expressions that contain variables or substitution parameters for prepared statements.

For example, assuming a variable by name `VAR_WINDOW_SIZE` is defined:

```
select * from StockTickEvent.win:time(VAR_WINDOW_SIZE)
```

Expression parameters for views are evaluated at the time the view is first created. They are not continuously re-evaluated by built-in views. For applications that provide a custom plug-in view, such custom views may re-evaluate parameter expressions.

If a view takes no parameters, use empty parenthesis `()`.

11.1. Window views

All the views explained below are data window views, as are `std:unique`, `std:firstunique`, `std:lastevent` and `std:firstevent`.

11.1.1. Length window (`win:length`)

This view is a moving (sliding) length window extending the specified number of elements into the past. The view takes a single expression as a parameter providing a numeric size value that defines the window size:

```
win:length(size_expression)
```

The below example sums the price for the last 5 stock ticks for symbol GE.

```
select sum(price) from StockTickEvent(symbol='GE').win:length(5)
```

The next example keeps a length window of 10 events of stock trade events, with a separate window for each symbol. The sum of price is calculated only for the last 10 events for each symbol and aggregates per symbol:

```
select sum(price) from StockTickEvent.std:groupwin(symbol).win:length(10) group by symbol
```

11.1.2. Length batch window (`win:length_batch`)

This window view buffers events (tumbling window) and releases them when a given minimum number of events has been collected. Provide an expression defining the number of events to batch as a parameter:

```
win:length_batch(size_expression)
```

The next statement buffers events until a minimum of 10 events have collected. Listeners to updates posted by this view receive updated information only when 10 or more events have collected.

```
select * from StockTickEvent.win:length_batch(10)
```

11.1.3. Time window (`win:time`)

This view is a moving (sliding) time window extending the specified time interval into the past based on the system time. Provide a time period (see Section 4.2.1, “Specifying Time Periods”) or an expression defining the number of seconds as a parameter:

```
win:time(time period)
```

```
win:time(seconds_interval_expression)
```

For the GE stock tick events in the last 1 second, calculate a sum of price.

```
select sum(price) from StockTickEvent(symbol='GE').win:time(1 sec)
```

The following time windows are equivalent specifications:

```
win:time(2 minutes 5 seconds)
win:time(125 sec)
win:time(125)
win:time(MYINTERVAL) // MYINTERVAL defined as a variable
```

11.1.4. Externally-timed window (`win:ext_timed`)

Similar to the time window, this view is a moving (sliding) time window extending the specified time interval into the past, but based on the millisecond time value supplied by a timestamp expression. The view takes two parameters: the expression to return long-typed timestamp values, and a time period or expression that provides a number of seconds:

```
win:ext_timed(timestamp_expression, time_period)
```

```
win:ext_timed(timestamp_expression, seconds_interval_expression)
```

The key difference comparing the externally-timed window to the regular time window is that the window slides not based on the engine time, but strictly based on the result of the timestamp expression when evaluated against the events entering the window.

The algorithm underlying the view compares the timestamp value returned by the expression when the oldest event arrived with the timestamp value returned by the expression for the newest arriving event on event arrival. If the time interval between the timestamp values is larger than the timer period parameter, then the algorithm removes all oldest events tail-first until the difference between the oldest and newest event is within the time interval. The window therefore slides only when events arrive and only considers each event's timestamp property (or other expression value returned) and not engine time.

This view holds stock tick events of the last 10 seconds based on the timestamp property in `StockTickEvent`.

```
select * from StockTickEvent.win:ext_timed(timestamp, 10 seconds)
```

The externally-timed data window expects strict ordering of the timestamp values returned by the timestamp expression. The view is not useful for ordering events in time order, please use the time-order view instead.

11.1.5. Time batch window (`win:time_batch`)

This window view buffers events (tumbling window) and releases them every specified time interval in one update. The view takes a time period or an expression providing a number of seconds as a parameter, plus optional parameters described next.

```
win:time_batch(time_period [,optional_reference_point] [,flow_control])
```

```
win:time_batch(seconds_interval_expression [,optional_reference_point] [,flow_control])
```

The time batch window takes a second, optional parameter that serves as a reference point to batch flush times. If not specified, the arrival of the first event into the batch window sets the reference point. Therefore if the reference point is not specified and the first event arrives at time t_1 , then the batch flushes at time t_1 plus *time_period* and every *time_period* thereafter.

The below example batches events into a 5 second window releasing new batches every 5 seconds. Listeners to updates posted by this view receive updated information only every 5 seconds.

```
select * from StockTickEvent.win:time_batch(5 sec)
```

By default, if there are no events arriving in the current interval (insert stream), and no events remain from the prior batch (remove stream), then the view does not post results to listeners. The view allows overriding this default behavior via flow control keywords.

The synopsis with flow control parameters is:

```
win:time_batch(time_period or seconds_interval_expr [,optional_reference_point]
[, "flow-control-keyword [, keyword...]" ] )
```

The `FORCE_UPDATE` flow control keyword instructs the view to post an empty result set to listeners if there is no data to post for an interval. When using this keyword the `irstream` keyword should be used in the `select` clause to ensure the remove stream is also output.

The `START_EAGER` flow control keyword instructs the view to post empty result sets even before the first event arrives, starting a time interval at statement creation time. As when using `FORCE_UPDATE`, the view

also posts an empty result set to listeners if there is no data to post for an interval, however it starts doing so at time of statement creation rather than at the time of arrival of the first event.

Taking the two flow control keywords in one sample statement, this example presents a view that waits for 10 seconds. It posts empty result sets after one interval after the statement is created, and keeps posting an empty result set as no events arrive during intervals:

```
select * from MyEvent.win:time_batch(10 sec, "FORCE_UPDATE, START_EAGER")
```

The optional reference point is provided as a long-value of milliseconds relative to January 1, 1970 and time 00:00:00.

The following example statement sets the reference point to 5 seconds and the batch size to 1 hour, so that each batch output is 5 seconds after each hour:

```
select * from OrderSummaryEvent.win:time_batch(1 hour, 5000L)
```

11.1.6. Time-Length combination batch window (`win:time_length_batch`)

This data window view is a combination of time and length batch (tumbling) windows. Similar to the time and length batch windows, this view batches events and releases the batched events when either one of the following conditions occurs, whichever occurs first: the data window has collected a given number of events, or a given time interval has passed.

The view parameters take 2 forms. The first form accepts a time period or an expression providing a number of seconds, and an expression for the number of events:

```
win:time_length_batch(time_period, number_of_events_expression)
```

```
win:time_length_batch(seconds_interval_expression, number_of_events_expression)
```

The next example shows a time-length combination batch window that batches up to 100 events or all events arriving within a 1-second time interval, whichever condition occurs first:

```
select * from MyEvent.win:time_length_batch(1 sec, 100)
```

In this example, if 100 events arrive into the window before a 1-second time interval passes, the view posts the batch of 100 events. If less than 100 events arrive within a 1-second interval, the view posts all events that arrived within the 1-second interval at the end of the interval.

By default, if there are no events arriving in the current interval (insert stream), and no events remain from the prior batch (remove stream), then the view does not post results to listeners. This view allows overriding this default behavior via flow control keywords.

The synopsis of the view with flow control parameters is:

```
win:time_length_batch(time_period or seconds_interval_expression, number_of_events_expression,  
    "flow control keyword [, keyword...]")
```

The `FORCE_UPDATE` flow control keyword instructs the view to post an empty result set to listeners if there is no data to post for an interval. The view begins posting no later than after one time interval passed after the first event arrives. When using this keyword the `irstream` keyword should be used in the `select` clause to ensure the remove stream is also output.

The `START_EAGER` flow control keyword instructs the view to post empty result sets even before the first event arrives, starting a time interval at statement creation time. As when using `FORCE_UPDATE`, the view also posts an empty result set to listeners if there is no data to post for an interval, however it starts doing so at time of statement creation rather than at the time of arrival of the first event.

Taking the two flow control keywords in one sample statement, this example presents a view that waits for 10 seconds or reacts when the 5th event arrives, whichever comes first. It posts empty result sets after one interval after the statement is created, and keeps posting an empty result set as no events arrive during intervals:

```
select * from MyEvent.win:time_length_batch(10 sec, 5, "FORCE_UPDATE, START_EAGER")
```

11.1.7. Time-Accumulating window (`win:time_accum`)

This data window view is a specialized moving (sliding) time window that differs from the regular time window in that it accumulates events until no more events arrive within a given time interval, and only then releases the accumulated events as a remove stream.

The view accepts a single parameter: the time period or seconds-expression specifying the length of the time interval during which no events must arrive until the view releases accumulated events. The synopsis is as follows:

```
win:time_accum(time_period)
```

```
win:time_accum(seconds_interval_expression)
```

The next example shows a time-accumulating window that accumulates events, and then releases events if within the time interval no more events arrive:

```
select * from MyEvent.win:time_accum(10 sec)
```

This example accumulates events, until when for a period of 10 seconds no more `MyEvent` events arrive, at which time it posts all accumulated `MyEvent` events.

Your application may only be interested in the batches of events as events leave the data window. This can be done simply by selecting the remove stream of this data window, populated by the engine as accumulated events leave the data window all-at-once when no events arrive during the time interval following the time the last event arrived:

```
select rstream * from MyEvent.win:time_accum(10 sec)
```

If there are no events arriving, then the view does not post results to listeners.

11.1.8. Keep-All window (`win:keepall`)

This keep-all data window view simply retains all events. The view does not remove events from the data window, unless used with a named window and the `on delete` clause.

The view accepts no parameters. The synopsis is as follows:

```
win:keepall()
```

The next example shows a keep-all window that accumulates all events received into the window:

```
select * from MyEvent.win:keepall()
```

Note that since the view does not release events, care must be taken to prevent retained events from using all available memory.

11.1.9. First Length (**win:firstlength**)

The `firstlength` view retains the very first *size_expression* events.

The synopsis is:

```
win:firstlength(size_expression)
```

If used within a named window and an `on-delete` clause deletes events, the view accepts further arriving events until the number of retained events reaches the size of *size_expression*.

The below example creates a view that retains only the first 10 events:

```
select * from MyEvent.win:firstlength(10)
```

11.1.10. First Time (**win:firsttime**)

The `firsttime` view retains all events arriving within a given time interval after statement start.

The synopsis is:

```
win:firsttime(time_period)
```

```
win:firsttime(seconds_interval_expression)
```

The below example creates a view that retains only those events arriving within 1 minute and 10 seconds of statement start:

```
select * from MyEvent.win:firsttime(1 minute 10 seconds)
```

11.2. Standard view set

11.2.1. Unique (**std:unique**)

The `unique` view is a view that includes only the most recent among events having the same value(s) for the result of the specified expression or list of expressions.

The synopsis is:

```
std:unique(unique_expression [, unique_expression ...])
```

The view acts as a length window of size 1 for each distinct value returned by an expression, or combination of values returned by multiple expressions. It thus posts as old events the prior event of the same value(s), if any.

An expression may return a `null` value. The engine treats a `null` value as any other value. An expression can

also return a custom application object, whereby the application class should implement the `GetHashCode` and `Equals` methods.

The below example creates a view that retains only the last event per symbol.

```
select * from StockTickEvent.std:unique(symbol)
```

The next example creates a view that retains the last event per symbol and feed.

```
select * from StockTickEvent.std:unique(symbol, feed)
```

11.2.2. Grouped Data Window (`std:groupwin`)

This view groups events into sub-views by the value returned by the specified expression or the combination of values returned by a list of expressions. The view takes a single expression to supply the group criteria values, or a list of expressions as parameters, as the synopsis shows:

```
std:groupwin(grouping_expression [, grouping_expression ...])
```

The *grouping_expression* expression(s) return one or more group keys, by which the view creates sub-views for each distinct group key. Note that the expression should not return an unlimited number of values: the grouping expression should not return a time value or otherwise unlimited key.

An expression may return a `null` value. The engine treats a `null` value as any other value. An expression can also return a custom application object, whereby the application class should implement the `GetHashCode` and `Equals` methods.

This example computes the total price for the last 5 events considering the last 5 events per each symbol, aggregating the price across all symbols (since no `group by` clause is specified the aggregation is across all symbols):

```
select symbol, sum(price) from StockTickEvent.std:groupwin(symbol).win:length(5)
```

The `@Hint("reclaim_group_aged=age_in_seconds")` hint instructs the engine to discard grouped data window state that has not been updated for *age_in_seconds* seconds. The optional `@Hint("reclaim_group_freq=sweep_frequency_in_seconds")` can be specified in addition to control the frequency at which the engine sweeps data window state. If the hint is not specified, the frequency defaults to the same value as *age_in_seconds*. Use the hints when your group criteria returns a changing or unlimited number of values. By default and without hints the view does not reclaim or remove data windows for group criteria values.

The updated sample statement with both hints:

```
// Remove data window views for symbols not updated for 10 seconds or more and sweep every 30 seconds
@Hint('reclaim_group_aged=10,reclaim_group_freq=30')
select symbol, sum(price) from StockTickEvent.std:groupwin(symbol).win:length(5)
```

To compute the total price for the last 5 events considering the last 5 events per each symbol and outputting a price per symbol, add the `group by` clause:

```
select symbol, sum(price) from StockTickEvent.std:groupwin(symbol).win:length(5) group by symbol
```

The `std:groupwin` grouped-window view can also take multiple expressions that provide values to group by. This example computes the total price for each symbol and feed for the last 10 events per symbol and feed

combination:

```
select sum(price) from StockTickEvent.std:groupwin(symbol, feed).win:length(10)
```

The order in which the `std:groupwin` grouped-window view appears within sub-views of a stream controls the data the engine derives from events for each group. The next 2 statements demonstrate this using a length window.

Without the `std:groupwin` declaration query the same query returns the total price per symbol for only the last 10 events across all symbols. Here the engine allocates only one length window for all events:

```
select sum(price) from StockTickEvent.win:length(10)
```

We have learned that by placing the `std:groupwin` grouped-window view before other views, these other views become part of the grouped set of views. The engine dynamically allocates a new view instance for each sub-view, every time it encounters a new group key such as a new value for `symbol`. Therefore, in `std:groupwin(symbol).win:length(10)` the engine allocates a new length window for each distinct symbol. However in `win:length(10)` alone the engine maintains a single length window.

The `std:groupwin` can be used with multiple data window views to achieve a grouped intersection or union policy.

The next query retains the last 4 events per symbol and only those events that are also not older than 10 seconds:

```
select * from StockTickEvent.std:groupwin(symbol).win:length(4).win:time(10)
```

Last, we consider a grouped data window for two group criteria. Here, the query results are total price per symbol and feed for the last 100 events per symbol and feed.

```
select sum(price) from StockTickEvent.std:groupwin(symbol, feed).win:length(100)
```

For advanced users: There is an optional view that can control how the `std:groupwin` grouped-window view gets evaluated and that view is the `std:merge` view. The merge view can only occur after a `std:groupwin` grouped-window view in a view chain and controls at what point in the view chain the merge of the data stream occurs from view-instance-per-criteria to single view.

Compare the following statements:

```
select * from Market.std:groupwin(ticker).win:length(1000000)
    .stat:weighted_avg(price, volume).std:merge(ticker)
// ... and ...
select * from Market.std:groupwin(ticker).win:length(1000000).std:merge(ticker)
    .stat:weighted_avg(price, volume)
```

If your statement does not specify the optional `std:merge` view, the semantics are the same as the first statement.

The first statement, in which the merge-view is added to the end (same as no merge view), computes weighted average per ticker, considering, per-ticker, the last 1M Market events for each ticker. The second statement, in which the merge view is added to the middle, computes weighted average considering, per-ticker, the last 1M Market events, computing the weighted average for all such events using a single view rather than multiple view instances with one view per ticker.

11.2.3. Size (`std:size`)

This view posts the number of events received from a stream or view plus any additional event properties or expression values listed as parameters. The synopsis is:

```
std:size([expression, ...])
```

The view posts a single long-typed property named `size`. The view posts the prior size as old data, and the current size as new data to update listeners of the view. Via the `GetEnumerator` method of the statement the size value can also be polled (read).

As optional parameters the view takes a list of expressions that the view evaluates against the last arriving event and provides along the `size` field.

An alternative to receiving a data window event count is the `prevcount` function. Compared to the `std:size` view the `prevcount` function requires a data window while the `std:size` view does not. The related `count(...)` aggregation function provides a count per group when used with `group by`.

When combined with a data window view, the size view reports the current number of events in the data window in the insert stream and the prior number of events in the data window as the remove stream. This example reports the number of tick events within the last 1 minute:

```
select size from StockTickEvent.win:time(1 min).std:size()
```

To select additional event properties you may add each event property to output as a parameter to the view.

The next example selects the symbol and feed event properties in addition to the `size` property:

```
select size, symbol, feed from StockTickEvent.win:time(1 min).std:size(symbol, feed)
```

The size view is also useful in conjunction with a `std:groupwin` grouped-window view to count the number of events per group. The EPL below returns the number of events per symbol.

```
select size from StockTickEvent.std:groupwin(symbol).std:size()
```

When used without a data window, the view simply counts the number of events:

```
select size from StockTickEvent.std:size()
```

All views can be used with pattern statements as well. The next EPL snippet shows a pattern where we look for tick events followed by trade events for the same symbol. The size view counts the number of occurrences of the pattern.

```
select size from pattern[every s=StockTickEvent -> TradeEvent(symbol=s.symbol)].std:size()
```

11.2.4. Last Event (`std:lastevent`)

This view exposes the last element of its parent view:

```
std:lastevent()
```

The view acts as a length window of size 1. It thus posts as old events the prior event in the stream, if any.

This example statement retains the last stock tick event for the symbol GE.

```
select * from StockTickEvent(symbol='GE').std:lastevent()
```

If you want to output the last event within a sliding window, please see Section 8.1.8, “The Previous Function”. That function accepts a relative (count) or absolute index and returns event properties or an event in the context of the specified data window.

11.2.5. First Event (`std:firstevent`)

This view retains only the first arriving event:

```
std:firstevent()
```

All events arriving after the first event are discarded.

If used within a named window and an `on-delete` clause deletes the first event, the view resets and will retain the next arriving event.

An example of a statement that retains the first `ReferenceData` event arriving is:

```
select * from ReferenceData.std:firstevent()
```

If you want to output the first event within a sliding window, please see Section 8.1.8, “The Previous Function”. That function accepts a relative (count) or absolute index and returns event properties or an event in the context of the specified data window.

11.2.6. First Unique (`std:firstunique`)

The `firstunique` view retains only the very first among events having the same value for the specified expression or list of expressions.

The synopsis is:

```
std:firstunique(unique_expression [, unique_expression ...])
```

If used within a named window and an `on-delete` clause deletes events, the view resets and will retain the next arriving event for the expression result value(s) of the deleted events.

The below example creates a view that retains only the first event per category:

```
select * from ReferenceData.std:firstunique(category)
```

11.3. Statistics views

The statistics views can be used combined with data window views or alone. Very similar to aggregation functions, these views aggregate or derive information from an event stream. As compared to aggregation functions, statistics views can post multiple derived fields including properties from the last event that was received. The derived fields and event properties are available for querying in the `where`-clause and are often compared to prior values using the `prior` function.

Statistics views accept one or more primary value expressions and any number of optional additional expressions that return values based on the last event received.

11.3.1. Univariate statistics (`stat:uni`)

This view calculates univariate statistics on a numeric expression. The view takes a single value expression as a parameter plus any number of optional additional expressions to return properties of the last event. The value expression must return a numeric value:

```
stat:uni(value_expression [,expression, ...])
```

After the value expression you may optionally list additional expressions or event properties to evaluate for the stream and return their value based on the last arriving event.

Table 11.3. Univariate statistics derived properties

Property Name	Description
datapoints	Number of values, equivalent to <code>count(*)</code> for the stream
total	Sum of values
average	Average of values
variance	Variance
stddev	Sample standard deviation (square root of variance)
stddevpa	Population standard deviation

The below example selects the standard deviation on price for stock tick events for the last 10 events.

```
select stddev from StockTickEvent.win:length(10).stat:uni(price)
```

To add properties from the event stream you may simply add all additional properties as parameters to the view.

This example selects all of the derived values, based on the price property, plus the values of the symbol and feed event properties:

```
select * from StockTickEvent.win:length(10).stat:uni(price, symbol, feed)
```

11.3.2. Regression (`stat:linest`)

This view calculates regression and related intermediate results on the values returned by two expressions. The view takes two value expressions as parameters plus any number of optional additional expressions to return properties of the last event. The value expressions must return a numeric value:

```
stat:linest(value_expression, value_expression [,expression, ...])
```

After the two value expressions you may optionally list additional expressions or event properties to evaluate for the stream and return their value based on the last arriving event.

Table 11.4. Regression derived properties

Property Name	Description
slope	Slope.

Property Name	Description
YIntercept	Y intercept.
XAverage	X average.
XStandardDeviationPop	X standard deviation population.
XStandardDeviationSample	X standard deviation sample.
XSum	X sum.
XVariance	X variance.
YAverage	X average.
YStandardDeviationPop	Y standard deviation population.
YStandardDeviationSample	Y standard deviation sample.
YSum	Y sum.
YVariance	Y variance.
dataPoints	Number of data points.
n	Number of data points.
sumX	Sum of X (same as X Sum).
sumXSq	Sum of X squared.
sumXY	Sum of X times Y.
sumY	Sum of Y (same as Y Sum).
sumYSq	Sum of Y squared.

The next example calculates regression and returns the slope and y-intercept on price and offer for all events in the last 10 seconds.

```
select slope, YIntercept from StockTickEvent.win:time(10 seconds).stat:linest(price, offer)
```

To add properties from the event stream you may simply add all additional properties as parameters to the view.

This example selects all of the derived values, based on the price and offer properties, plus the values of the symbol and feed event properties:

```
select * from StockTickEvent.win:time(10 seconds).stat:linest(price, offer, symbol, feed)
```

11.3.3. Correlation (stat:correl)

This view calculates the correlation value on the value returned by two expressions. The view takes two value expressions as parameters plus any number of optional additional expressions to return properties of the last event. The value expressions must be return a numeric value:

```
stat:correl(value_expression, value_expression [,expression, ...])
```

After the two value expressions you may optionally list additional expressions or event properties to evaluate for the stream and return their value based on the last arriving event.

Table 11.5. Correlation derived properties

Property Name	Description
correlation	Correlation between two event properties

The next example calculates correlation on price and offer over all stock tick events for GE:

```
select correlation from StockTickEvent(symbol='GE').stat:correl(price, offer)
```

To add properties from the event stream you may simply add all additional properties as parameters to the view.

This example selects all of the derived values, based on the price and offer property, plus the values of the feed event property:

```
select * from StockTickEvent(symbol='GE').stat:correl(price, offer, feed)
```

11.3.4. Weighted average (`stat:weighted_avg`)

This view returns the weighted average given an expression returning values to compute the average for and an expression returning weight. The view takes two value expressions as parameters plus any number of optional additional expressions to return properties of the last event. The value expressions must return numeric values:

```
stat:weighted_avg(value_expression_field, value_expression_weight [,expression, ...])
```

After the value expression you may optionally list additional expressions or event properties to evaluate for the stream and return their value based on the last arriving event.

Table 11.6. Weighted average derived properties

Property Name	Description
average	Weighted average

A statement that derives the volume-weighted average price for the last 3 seconds for a given symbol is shown below:

```
select average
from StockTickEvent(symbol='GE').win:time(3 seconds).stat:weighted_avg(price, volume)
```

To add properties from the event stream you may simply add all additional properties as parameters to the view.

This example selects all of the derived values, based on the price and volume properties, plus the values of the symbol and feed event properties:

```
select *
from StockTickEvent.win:time(3 seconds).stat:weighted_avg(price, volume, symbol, feed)
```

Aggregation functions could instead be used to compute the weighted average as well. The next example also posts weighted average per symbol considering the last 3 seconds of stock tick data:

```
select symbol, sum(price*volume)/sum(volume)
from StockTickEvent.win:time(3 seconds) group by symbol
```

The following example computes weighted average keeping a separate data window per symbol considering the last 5 events of each symbol:

```
select symbol, average
from StockTickEvent.std:groupwin(symbol).win:length(5).stat:weighted_avg(price, volume)
```

11.4. Extension View Set

The views in this set are data windows that order events according to a criteria.

11.4.1. Sorted Window View (`ext:sort`)

This view sorts by values returned by the specified expression or list of expressions and keeps only the top (or bottom) events up to the given size.

The syntax is as follows:

```
ext:sort(size_expression,
        sort_criteria_expression [asc/desc][, sort_criteria_expression [asc/desc]...])
```

An expression may be followed by the optional `asc` or `desc` keywords to indicate that the values returned by that expression are sorted in ascending or descending sort order.

The view below retains only those events that have the highest 10 prices and reports a total price:

```
select sum(price) from StockTickEvent.ext:sort(10, price desc)
```

The following example sorts events first by price in descending order, and then by symbol name in ascending (alphabetical) order, keeping only the 10 events with the highest price (with ties resolved by alphabetical order of symbol).

```
select * from StockTickEvent.ext:sort(10, price desc, symbol asc)
```

11.4.2. Time-Order View (`ext:time_order`)

This view orders events that arrive out-of-order, using timestamp-values provided by an expression, and by comparing that timestamp value to engine system time.

The syntax for this view is as follows.

```
ext:time_order(timestamp_expression, time_period)
```

```
ext:time_order(timestamp_expression, seconds_interval_expression)
```

The first parameter to the view is the expression that supplies timestamp values. The timestamp is expected to be a long-typed millisecond value that denotes an event's time of consideration by the view (or other expres-

sion). This is typically the time of arrival. The second parameter is a number-of-seconds expression or the time period specifying the time interval that an arriving event should maximally be held, in order to consider older events arriving at a later time.

Since the view compares timestamp values to engine time, the view requires that the timestamp values and current engine time are both following the same clock. Therefore, to the extent that the clocks that originated both timestamps differ, the view may produce inaccurate results.

As an example, the next statement uses the `arrival_time` property of `MyTimestampedEvent` events to order and release events by arrival time:

```
insert rstream into ArrivalTimeOrderedStream
select rstream * from MyTimestampedEvent.ext:time_order(arrival_time, 10 sec)
```

In the example above, the `arrival_time` property holds a long-typed timestamp value in milliseconds. On arrival of an event, the engine compares the timestamp value of each event to the tail-time of the window. The tail-time of the window is, in this example, 10 seconds before engine time (continuously sliding). If the timestamp value indicates that the event is older then the tail-time of the time window, the event is released immediately in the remove stream. If the timestamp value indicates that the event is newer then the tail-time of the window, the view retains the event until engine time moves such that the event timestamp is older then tail-time.

The examples thus holds each arriving event in memory anywhere from zero seconds to 10 seconds, to allow for older events (considering arrival time timestamp) to arrive. In other words, the view holds an event with an arrival time equal to engine time for 10 seconds. The view holds an event with an arrival time that is 2 seconds older then engine time for 8 seconds. The view holds an event with an arrival time that is 10 or more seconds older then engine time for zero seconds, and releases such (old) events immediately into the remove stream.

The insert stream of this sliding window consists of all arriving events. The remove stream of the view is ordered by timestamp value: The event that has the oldest timestamp value is released first, followed by the next newer events. Note the statement above uses the `rstream` keyword in both the `insert into` clause and the `select` clause to select ordered events only. It uses the `insert into` clause to makes such ordered stream available for subsequent statements to use.

It is up to your application to populate the timestamp property into your events or use a sensible expression that returns timestamp values for consideration by the view. The view also works well if you use externally-provided time via timer events.

Chapter 12. API Reference

12.1. API Overview

Esper has the following primary interfaces:

- The event and event type interfaces are described in Section 12.5, “Event and Event Type”.
- The administrative interface to create and manage EPL and pattern statements, and set runtime configurations, is described in Section 12.3, “The Administrative Interface”.
- The runtime interface to send events into the engine, set and get variable values and execute on-demand queries, is described in Section 12.4, “The Runtime Interface”.

For EPL introductory information please see Section 4.1, “EPL Introduction” and patterns are described at Section 5.1, “Event Pattern Overview”.

The automated API documentation is also a great source for API information.

12.2. The Service Provider Interface

The `EPServiceProvider` interface represents an engine instance. Each instance of an Esper engine is completely independent of other engine instances and has its own administrative and runtime interface.

An instance of the Esper engine is obtained via static methods on the `EPServiceProviderManager` class. The `GetDefaultProvider` method and the `GetProvider(String providerURI)` methods return an instance of the Esper engine. The latter can be used to obtain multiple instances of the engine for different provider URI values. The `EPServiceProviderManager` determines if the provider URI matches all prior provider URI values and returns the same engine instance for the same provider URI value. If the provider URI has not been seen before, it creates a new engine instance.

The code snippet below gets the default instance Esper engine. Subsequent calls to get the default engine instance return the same instance.

```
EPServiceProvider epService = EPServiceProviderManager.GetDefaultProvider();
```

This code snippet gets an Esper engine for the provider URI `RFIDProcessor1`. Subsequent calls to get an engine with the same provider URI return the same instance.

```
EPServiceProvider epService = EPServiceProviderManager.GetProvider("RFIDProcessor1");
```

Since the `GetProvider` methods return the same cached engine instance for each URI, there is no need to statically cache an engine instance in your application.

An existing Esper engine instance can be reset via the `Initialize` method on the `EPServiceProvider` instance. This operation stops and removes all statements and resets the engine to the configuration provided when the engine instance for that URI was obtained. If no configuration is provided, an empty (default) configuration applies.

After `Initialize` your application must obtain new administrative and runtime services. Any administrative

and runtime services obtained before the initialize are invalid and have undefined behavior.

The next code snippet outlines a typical sequence of use:

```
// Configure the engine, this is optional
Configuration config = new Configuration();
config.Configure("configuration.xml"); // load a configuration from file
config.XXX = ...; // make additional configuration settings

// Obtain an engine instance
EPServiceProvider epService = EPServiceProviderManager.GetDefaultProvider(config);

// Optionally, use initialize if the same engine instance has been used before to start clean
epService.Initialize();

// Optionally, make runtime configuration changes
epService.EPAdministrator.GetConfiguration().Add...(...);

// Destroy the engine instance when no longer needed, frees up resources
epService.Dispose();
```

An existing Esper engine instance can be destroyed via the `Dispose` method on the `EPServiceProvider` instance. This stops and removes all statements as well as frees all resources held by the instance. After a `Dispose` the engine can no longer be used.

The `EPServiceProvider` exposes a number of events that are used to inform observers when an engine instance is about to be destroyed, after an engine instance has been initialized, when a new statement gets created and when a statement gets started, stopped or destroyed. Event handlers are registered using the events in the table below.

Table 12.1. Events For Service State Changes

Event on <code>EPServiceProvider</code>	Description	
<code>ServiceDestroyRequested</code>	Occurs after the service has been "destroyed" and all resources have been cleaned up.	
<code>ServiceInitialized</code>	Occurs after the service provider has been initialized.	
<code>StatementCreate</code>	Occurs after a statement has been created.	
<code>StatementStateChange</code>	Occurs when a statement started, stopped or destroyed.	

As engine instances are completely independent, your application may not send `EventBean` instances obtained from one engine instance into a second engine instance since the event type space between two engine instances is not shared.

12.3. The Administrative Interface

12.3.1. Creating Statements

Create event pattern expression and EPL statements via the administrative interface `EPAdministrator`.

This code snippet gets an Esper engine then creates an event pattern and an EPL statement.

```
EPServiceProvider epService = EPServiceProviderManager.GetDefaultProvider();
EPAdministrator admin = epService.EPAdministrator;

EPStatement 10secRecurTrigger = admin.CreatePattern(
    "every timer:at(*, *, *, *, *, */10)");

EPStatement countStmt = admin.CreateEPL(
    "select count(*) from MarketDataBean.win:time(60 sec)");
```

Note that event pattern expressions can also occur within EPL statements. This is outlined in more detail in Section 4.4.2, “Pattern-based Event Streams”.

The `Create` methods on `EPAdministrator` are overloaded and allow an optional statement name to be passed to the engine. A statement name can be useful for retrieving a statement by name from the engine at a later time. The engine assigns a statement name if no statement name is supplied on statement creation.

The `createPattern` and `CreateEPL` methods return `EPStatement` instances. Statements are automatically started and active when created. A statement can also be stopped and started again via the `Stop` and `Start` methods shown in the code snippet below.

```
countStmt.Stop();
countStmt.Start();
```

The `Create` methods on `EPAdministrator` also accept a user object. The user object is associated with a statement at time of statement creation and is a single, unnamed field that is stored with every statement. Applications may put arbitrary objects in this field. Use the `UserObject` property on `EPStatement` to obtain the user object of a statement.

Your application may create new statements or stop and destroy existing statements using any thread and also within event handlers or subscriber code. If using native object events, your application may not create or manage statements in the event object itself while the same event is currently being processed by a statement.

12.3.2. Receiving Statement Results

Esper provides three choices for your application to receive statement results. Your application can use all mechanisms alone or in any combination for each statement. The choices are:

Table 12.2. Choices For Receiving Statement Results

Name	Method, Property or Event on <code>EPStatement</code>	Description
Event Handlers	Events	<p>Your application provides implementations of the <code>UpdateEventHandler</code> to the statement. Event handlers receive <code>UpdateEventArgs</code> instances containing statement results.</p> <p>The engine continuously indicates results to all event handlers as soon they occur, and following output rate limiting clauses if specified.</p>

Name	Method, Property or Event on EPStatement	Description
Subscriber Object	Subscriber	<p>Your application provides a vanilla object that exposes methods to receive statement results.</p> <p>The engine continuously indicates results to the single subscriber as soon they occur, and following output rate limiting clauses if specified.</p> <p>This is the fastest method to receive statement results, as the engine delivers strongly-typed results directly to your application objects without the need for building an <code>EventBean</code> result set as in the Event Handler choice.</p> <p>There can be at most 1 Subscriber Object registered per statement. If you require more than one event handler, use the event handler instead (or in addition). The Subscriber Object is bound to the statement with a strongly typed support which ensure direct delivery of new events without type conversion. This optimization is made possible because there can only be 0 or 1 Subscriber Object per statement.</p>
Pull API	<code>GetSafeEnumerator</code> and <code>GetEnumerator</code>	<p>Your application asks the statement for results and receives a set of events via <code>System.Collections.IEnumerable<EventBean></code>.</p> <p>This is useful if your application does not need continuous indication of new results in real-time.</p>

Your application may attach one or more event handlers, zero or one single subscriber and in addition use the Pull API on the same statement. There are no limitations to the use of enumerator, subscriber or event handler alone or in combination to receive statement results.

The best delivery performance can generally be achieved by attaching a subscriber and by not attaching event handlers. The engine is aware of the event handlers and subscriber attached to a statement. The engine uses this information internally to reduce statement overhead. For example, if your statement does not have event handlers or a subscriber attached, the engine does not need to continuously generate results for delivery.

If your application attaches both a subscriber and one or more event handlers then the subscriber receives the result first before any of the event handlers.

If your application attaches more then one event handler then the event handlers receive results first in the order they were added to the statement, and To change the order of delivery among event handlers your application can add and remove event handlers at runtime.

If you have configured outbound threading, it means a thread from the outbound thread pool delivers results to the subscriber and event handlers instead of the processing or event-sending thread.

If outbound threading is turned on, we recommend turning off the engine setting preserving the order of events delivered to event handlers as described in Section 13.4.9.1, “Preserving the order of events delivered to listeners”. If outbound threading is turned on statement execution is not blocked for the configured time in the case a

subscriber or listener takes too much time.

12.3.3. Setting a Subscriber Object

A subscriber object is a direct binding of query results to a CLR object. The object receives statement results via method invocation. The subscriber class does not need to implement an interface or extend a superclass. Only one subscriber object may be set for a statement.

Subscriber objects have several advantages over event handlers. First, they offer a substantial performance benefit: Query results are delivered directly to your method(s) through method calls, and there is no intermediate representation (`EventBean`). Second, as subscribers receive strongly-typed parameters, the subscriber code tends to be simpler.

This chapter describes the requirements towards the methods provided by your subscriber class.

The engine can deliver results to your subscriber in two ways:

1. Each event in the insert stream results in a method invocation, and each event in the remove stream results in further method invocations. This is termed *row-by-row delivery*.
2. A single method invocation that delivers all rows of the insert and remove stream. This is termed *multi-row delivery*.

Row-By-Row Delivery

Your subscriber class must provide a method by name `update` to receive insert stream events row-by-row. The number and types of parameters declared by the `update` method must match the number and types of columns as specified in the `select` clause, in the same order as in the `select` clause.

For example, if your statement is:

```
select orderId, price, count(*) from OrderEvent
```

Then your subscriber `update` method looks as follows:

```
public class MySubscriber {  
    ...  
    public void Update(String orderId, double price, long count) {...}  
    ...  
}
```

Each method parameter declared by the `update` method must be assignable from the respective column type as listed in the `select`-clause, in the order selected. The assignability rules are:

- Widening of types follows CLR standards. For example, if your `select` clause selects an integer value, the method parameter for the same column can be typed `int`, `long`, `float` or `double` (or any equivalent boxed type).
- Auto-boxing and unboxing follows CLR standards. For example, if your `select` clause selects a `Nullable<int>` value, the method parameter for the same column can be typed `int`. Note that if your `select` clause column may generate `null` values, an exception may occur at runtime unboxing the `null` value.
- Interfaces and super-classes are honored in the test for assignability. Therefore `System.Object` can be used to accept any `select` clause column type

Wildcards

If your `select` clause contains one or more wildcards (*), then the equivalent parameter type is the underlying event type of the stream selected from.

For example, your statement may be:

```
select *, count(*) from OrderEvent
```

Then your subscriber `Update` method looks as follows:

```
public void Update(OrderEvent orderEvent, long count) {...}
```

In a join, the wildcard expands to the underlying event type of each stream in the join in the order the streams occur in the `from` clause. An example statement for a join is:

```
select *, count(*) from OrderEvent order, OrderHistory hist
```

Then your subscriber `Update` method should be:

```
public void Update(OrderEvent orderEvent, OrderHistory orderHistory, long count) {...}
```

The stream wildcard syntax and the stream name itself can also be used:

```
select hist.*, order from OrderEvent order, OrderHistory hist
```

The matching `Update` method is:

```
public void Update(OrderHistory orderHistory, OrderEvent orderEvent) {...}
```

Row Delivery as Map and Object Array

Alternatively, your `Update` method may simply choose to accept `System.Collections.Generic.IDictionary` as a representation for each row. Each column in the `select` clause is then made an entry in the resulting `Map`. The `Map` keys are the column name if supplied, or the expression string itself for columns without a name.

The `Update` method for `Map` delivery is:

```
public void Update(IDictionary<string, object> row) {...}
```

The engine also supports delivery of `select` clause columns as an object array. Each item in the object array represents a column in the `select` clause. The `Update` method then looks as follows:

```
public void Update(Object[] row) {...}
```

Delivery of Remove Stream Events

Your subscriber receives remove stream events if it provides a method named `UpdateRStream`. The method must accept the same number and types of parameters as the `Update` method.

An example statement:

```
select orderId, count(*) from OrderEvent.win:time(20 sec) group by orderId
```

Then your subscriber `Update` and `UpdateRStream` methods should be:

```
public void Update(String, long count) {...}
```

```
public void UpdateRStream(String orderId, long count) {...}
```

Delivery of Begin and End Indications

If your subscriber requires a notification for begin and end of event delivery, it can expose methods by name `UpdateStart` and `UpdateEnd`.

The `UpdateStart` method must take two integer parameters that indicate the number of events of the insert stream and remove stream to be delivered. The engine invokes the `UpdateStart` method immediately prior to delivering events to the `Update` and `UpdateRStream` methods.

The `UpdateEnd` method must take no parameters. The engine invokes the `UpdateEnd` method immediately after delivering events to the `Update` and `UpdateRStream` methods.

An example set of delivery methods:

```
// Called by the engine before delivering events to update methods
public void UpdateStart(int insertStreamLength, int removeStreamLength)

// To deliver insert stream events
public void Update(String orderId, long count) {...}

// To deliver remove stream events
public void UpdateRStream(String orderId, long count) {...}

// Called by the engine after delivering events
public void UpdateEnd() {...}
```

Multi-Row Delivery

In place of row-by-row delivery, your subscriber can receive all events in the insert and remove stream via a single method invocation.

The event delivery follow the scheme as described earlier in Section 12.3.3.1.2, “Row Delivery as Map and Object Array”. The subscriber class must provide one of the following methods:

Table 12.3. Update Method for Multi-Row Delivery of Underlying Events

Method	Description
<code>Update(Object[][] insertStream, Object[][] removeStream)</code>	The first dimension of each Object array is the event row, and the second dimension is the column matching the column order of the statement <code>select</code> clause
<code>Update(IDictionary<string,object>[] insertStream, IDictionary<string,object>[] removeStream)</code>	Each map represents one event, and Map entries represent columns of the statement <code>select</code> clause

Wildcards

If your `select` clause contains a single wildcard (*) or wildcard stream selector, the subscriber object may also directly receive arrays of the underlying events. In this case, the subscriber class should provide a method `update(Underlying[] insertStream, Underlying[] removeStream)`, such that *Underlying* represents the class of the underlying event.

For example, your statement may be:

```
select * from OrderEvent.win:time(30 sec)
```

Your subscriber class exposes the method:

```
public void Update(OrderEvent[] insertStream, OrderEvent[] removeStream) { ... }
```

12.3.4. Adding Event Handlers

Your application can subscribe to updates posted by a statement via the `Events` event on `EPStatement`. Your application must to provide an implementation of the `UpdateEventHandler` to the statement:

```
public void HandleEvents(Object sender, UpdateEventArgs updateEventArgs) { ... }
countStmt.Events += HandleEvents;
```

The following adds and event handler using an embedded lambda expression.

```
countStmt.Events += (sender, updateEventArgs) => MyMethod(updateEventArgs);
```

EPL statements and event patterns publish old data and new data to registered `UpdateEventHandlers`. New data published by statements is the events representing the new values of derived data held by the statement. Old data published by statements consists of the events representing the prior values of derived data held by the statement.

It is important to understand that `UpdateEventHandlers` receive multiple result rows in one invocation by the engine: the new data and old data parameters to your event handler are array parameters. For example, if your application uses one of the batch data windows, or your application creates a pattern that matches multiple times when a single event arrives, then the engine indicates such multiple result rows in one invocation and your new data array carries two or more rows.

The `UpdateEventArgs` object that is passed as part of the event handler is especially useful when the same event handler is registered with multiple statements, as the event arguments contain the statement and engine instance in addition to the new and old data when the engine indicates new results to a event handler.

Subscription Snapshot and Atomic Delivery

The `AddEventHandlerWithReplay` method provided by `EPStatement` makes it possible to send a snapshot of current statement results to an event handler when the event handler is added.

When using the `AddEventHandlerWithReplay` method to register an event handler, the event handler receives current statement results as the first invocation of the delegate, passing in the `newEvents` parameter the current statement results as an array of zero or more events. Subsequent calls to the delegate of the event handler are statement results.

Current statement results are the events returned by the `GetEnumerator` or `GetSafeEnumerator` methods.

Delivery is atomic: Events occurring during delivery of current results to the event handler are guaranteed to be delivered in a separate call and not lost. The event handler implementation should thus minimize long-running or blocking operations to reduce lock times held on statement-level resources.

12.3.5. Using Enumerators

Subscribing to events posted by a statement is following a push model. The engine pushes data to event handlers when events are received that cause data to change or patterns to match. Alternatively, you need to know that statements serve up data that your application can obtain via the `GetSafeEnumerator` and `GetEnumerator` methods on `EPStatement`. This is called the pull API and can come in handy if your application is not interested in all new updates, and only needs to perform a frequent or infrequent poll for the latest data.

The `GetSafeEnumerator` method on `EPStatement` returns a concurrency-safe enumerator returning current statement results, even while concurrent threads may send events into the engine for processing. The safe enumerator guarantees correct results even as events are being processed by other threads. The cost is that the enumerator obtains and holds a statement lock that must be released via the `Dispose` method on the `GetSafeEnumerator` instance.

The `GetEnumerator` method on `EPStatement` returns a concurrency-unsafe enumerator. This enumerator is only useful for applications that are single-threaded, or applications that themselves perform coordination between the iterating thread and the threads that send events into the engine for processing. The advantage to this enumerator is that it does not hold a lock.

The next code snippet shows a short example of use of safe iterators:

```
EPStatement statement = epAdmin.CreateEPL("select avg(price) as avgPrice from MyTick");
// .. send events into the engine
// then use the pull API...
IEnumerator<EventBean> safeEnum = statement.GetSafeEnumerator();
while(safeEnum.MoveNext()) {
    // .. process event ..
    EventBean @event = safeEnum.Current;
    Console.WriteLine("avg:" + @event.Get("avgPrice"));
}
```

This is a short example of use of the regular iterator that is not safe for concurrent event processing:

```
double averagePrice = (double) eplStatement.FirstOrDefault().Get("average");
```

The `GetSafeEnumerator` and `GetEnumerator` methods can be used to pull results out of all statements, including statements that join streams, contain aggregation functions, pattern statements, and statements that contain a `where` clause, `group by` clause, `having` clause or `order by` clause.

For statements without an `order by` clause, the `GetEnumerator` method returns events in the order maintained by the data window. For statements that contain an `order by` clause, the `GetEnumerator` method returns events in the order indicated by the `order by` clause.

Consider using the `on-select` clause and a named window if your application requires iterating over a partial result set or requires indexed access for fast iteration; Note that `on-select` requires that you sent a trigger event, which may contain the key values for indexed access.

Esper places the following restrictions on the pull API and usage of the `GetSafeEnumerator` and `GetEnumerator` methods:

1. In multithreaded applications, use the `GetSafeEnumerator` method. Note: make sure your application closes the iterator via the `Close` method when done, otherwise the iterated statement stays locked and event processing for that statement does not resume.
2. In multithreaded applications, the `GetEnumerator` method does not hold any locks. The iterator returned by this method does not make any guarantees towards correctness of results and fail-behavior, if your application processes events into the engine instance by multiple threads. Use the `GetSafeEnumerator` method for concurrency-safe iteration instead.
3. Since the `GetSafeEnumerator` and `GetEnumerator` methods return events to the application immediately,

the iterator does not honor an output rate limiting clause, if present. That is, the iterator returns results as if there is no output-rate clause for the statement in statements without grouping or aggregation. For statements with grouping or aggregation, the iterator in combination with an output clause returns last output group and aggregation results. Use a separate statement and the `insert into` clause to control the output rate for iteration, if so required.

4. When iterating a statement that selects an unbound stream (no data window declared), the iterator returns the last event. When iterating a statement that groups and aggregates values from an unbound stream, the iterated result contains only the last updated group.

12.3.6. Managing Statements

The `EPAdministrator` interface provides the facilities for managing statements:

- Use the `GetStatement` method to obtain an existing started or stopped statement by name
- Use the `StatementNames` property to obtain a list of started and stopped statement names
- Use the `StartAllStatements`, `StopAllStatements` and `DestroyAllStatements` methods to manage all statements in one operation

12.3.7. Runtime Configuration

Certain configuration changes are available to perform on an engine instance while in operation. Such configuration operations are available via the `GetConfiguration` method on `EPAdministrator`, which returns a `ConfigurationOperations` object.

Please consult the SDK documentation of `ConfigurationOperations` for further information. The section Section 13.6, “Runtime Configuration” provides a summary of available configurations.

In summary, the configuration operations available on a running engine instance are as follows:

- Add new event types for all event representations, check if an event type exists, update an existing event type, remove an event type, query a list of types and obtain a type by name.
- Add and remove variables (get and set variable values is done via the runtime API).
- Add a variant stream.
- Add a revision event type.
- Add event types for all event classes in a given namespace, using the simple type name as the event name.
- Add import for user-defined functions.
- Add a plug-in aggregation function, plug-in single row function, plug-in event type, plug-in event type resolution URIs.
- Control metrics reporting.
- Additional items please see the `ConfigurationOperations` interface.

For examples of above runtime configuration API functions please consider the Configuration chapter, which applies to both static configuration and runtime configuration as the `ConfigurationOperations` interface is the same.

12.4. The Runtime Interface

The `EPRuntime` interface is used to send events for processing into an Esper engine, set and get variable values and execute on-demand queries.

The below code snippet shows how to send a CLR object event to the engine. Note that the `SendEvent` method is overloaded. As events can take on different representation types, the `SendEvent` takes parameters to reflect

the different types of events that can be send into the engine. The Chapter 2, *Event Representations* section explains the types of events accepted.

```
EPServiceProvider epService = EPServiceProviderManager.GetDefaultProvider();
EPRuntime runtime = epService.EPRuntime;

// Send an example event containing stock market data
runtime.SendEvent(new MarketDataBean('IBM', 75.0));
```

Events, in theoretical terms, are observations of a state change that occurred in the past. Since one cannot change an event that happened in the past, events are best modelled as immutable objects.

Note that the Esper engine relies on events that are sent into an engine to not change their state. Typically, applications create a new event object for every new event, to represent that new event. Application should not modify an existing event that was sent into the engine.

Another important method in the runtime interface is the `Route` method. This method is designed for use by `UpdateEventHandler` and subscriber implementations that need to send events into an engine instance to avoid the possibility of a stack overflow due to nested calls to `SendEvent`.

12.4.1. Event Sender

The `EventSender` interface processes event objects that are of a known type. This facility can reduce the overhead of event object reflection and type lookup as an event sender is always associated to a single concrete event type.

Use the method `GetEventSender(String eventName)` to obtain an event sender for processing events of the named type:

```
EventSender sender = epService.EPRuntime.GetEventSender("MyEvent");
sender.SendEvent(myEvent);
```

For events backed by a CLR type, the event sender ensures that the event object equals the underlying class, or implements or extends the underlying class for the given event type name.

For events backed by a `System.Collections.Generic.IDictionary` (Map events), the event sender does not perform any checking other then checking that the event object implements Map.

For events backed by a `System.Xml.XmlNode` (XML DOM events), the event sender checks that the root element name equals the root element name for the event type.

A second method to obtain an event sender is the method `GetEventSender(Uri[])`, which takes an array of URIs. This method is for use with plug-in event representations. The event sender returned by this method processes event objects that are of one of the types of one or more plug-in event representations. Please consult Section 15.8, “Event Type And Event Object” for more information.

12.4.2. Receiving Unmatched Events

Your application can register an implementation of the `UnmatchedListener` interface with the `EPRuntime` runtime via the `SetUnmatchedListener` method to receive events that were not matched by any statement.

Events that can be unmatched are all events that your application sends into the runtime via one of the `SendEvent` or `Route` methods, or that have been generated via an `insert into` clause.

For an event to become unmatched by any statement, the event must not match any statement's event stream fil-

ter criteria. Note that the EPL `where` clause or `having` clause are not considered part of the filter criteria for a stream, as explained by example below.

In the next statement all `MyEvent` events match the statement's event stream filter criteria, regardless of the value of the 'quantity' property. As long as the below statement remains started, the engine would not deliver `MyEvent` events to your registered `UnmatchedListener` instance:

```
select * from MyEvent where quantity > 5
```

In the following statement a `MyEvent` event with a 'quantity' property value of 5 or less does not match this statement's event stream filter criteria. The engine delivers such an event to the registered `UnmatchedListener` instance provided no other statement matches on the event:

```
select * from MyEvent(quantity > 5)
```

For patterns, if no pattern sub-expression is active for an event type, an event of that type also counts as unmatched in regards to the pattern statement.

12.4.3. On-Demand Snapshot Query Execution

As your application may not require streaming results and may not know each query in advance, the on-demand query facility provides for ad-hoc execution of an EPL expression.

On-demand queries are not continuous in nature: The query engine executes the query once and returns all result rows to the application. On-demand query execution is very lightweight as the engine performs no statement creation and the query leaves no traces within the engine.

Esper also provides the facility to explicitly index named windows to speed up queries. Please consult Section 4.15.10, “Explicitly Indexing Named Windows” for more information.

The following limitations apply:

- An on-demand EPL expression only evaluates against the named windows that your application creates. On-demand queries may not specify any other streams or application event types.
- The following clauses are not allowed in on-demand EPL: `insert into` and `output`.
- Views and patterns are not allowed to appear in on-demand queries.
- On-demand EPL may not perform subqueries.
- The `previous` and `prior` functions may not be used.

On-Demand Query API

The `EPRuntime` provides two ways to run on-demand queries:

1. Dynamic on-demand queries are executed once through the `ExecuteQuery` method.
2. Prepared on-demand queries: The `PrepareQuery` method returns an `EPOnDemandPreparedQuery` representing the query, and the query can be performed repeatedly via the `Execute` method.

Prepared on-demand queries are designed for repeated execution and may perform better than the dynamic

queries if running the same query multiple times. Placeholders are not allowed in prepared on-demand queries.

The next program listing runs an on-demand query against a named window `MyNamedWindow` and prints a column of each row result of the query:

```
String query = "select * from MyNamedWindow";
EPOnDemandQueryResult result = epRuntime.executeQuery(query);
for (EventBean row : result.getArray()) {
    System.out.println("name=" + row.get("name"));
}
```

The next code snippet demonstrates prepared on-demand queries:

```
EPOnDemandPreparedQuery prepared = epRuntime.prepareQuery(query);
EPOnDemandQueryResult result = prepared.execute();
// ...later ...
prepared.execute(); // execute a second time
```

Esper also provides the facility to explicitly index named windows to speed up queries. Please consult Section 4.15.10, “Explicitly Indexing Named Windows” for more information.

12.5. Event and Event Type

An `EventBean` object represents a row (event) in your continuous query's result set. Each `EventBean` object has an associated `EventType` object providing event metadata.

An `UpdateEventHandler` implementation receives one or more `UpdateEventArgs` events with each invocation. Via the `GetEnumerator` method on `EPStatement` your application can poll or read data out of statements. Statement enumerators also return `EventBean` instances.

Each statement provides the event type of the events it produces, available via the `EventType` property on `EPStatement`.

12.5.1. Event Type Metadata

An `EventType` object encapsulates all the metadata about a certain type of events. As Esper supports an inheritance hierarchy for event types, it also provides information about super-types to an event type.

An `EventType` object provides the following information:

- For each event property, it lists the property name and type as well as flags for indexed or mapped properties and whether a property is a fragment.
- The direct and indirect super-types to the event type.
- Value getters for property expressions.
- Underlying class of the event representation.

For each property of an event type, there is an `EventPropertyDescriptor` object that describes the property. The `EventPropertyDescriptor` contains flags that indicate whether a property is an indexed (array) or a mapped property and whether access to property values require an integer index value (indexed properties only) or string key value (mapped properties only). The descriptor also contains a fragment flag that indicates whether a property value is available as a fragment.

The term *fragment* means an event property value that is itself an event, or a property value that can be represented as an event. The `getFragmentType` on `EventType` may be used to determine a fragment's event type in ad-

vance.

A fragment event type and thereby fragment events allow navigation over a statement's results even if the statement result contains nested events or a graph of events. There is no need to use the reflection API to navigate events, since fragments allow the querying of nested event properties or array values, including nested CLR types.

When using the Map event representation, any named Map type nested within a Map as a simple or array property is also available as a fragment. When using CLR objects either directly or within Map events, any object that is neither a primitive or boxed built-in type, and that is not an enumeration and does not implement the Map interface is also available as a fragment.

The nested, indexed and mapped property syntax can be combined to a property expression that may query an event property graph. Most of the methods on the `EventType` interface allow a property expression to be passed.

Your application may use an `EventType` object to obtain special getter-objects. A getter-object is a fast accessor to a property value of an event of a given type. All getter objects implement the `EventPropertyGetter` interface. Getter-objects work only for events of the same type or sub-types as the `EventType` that provides the `EventPropertyGetter`. The performance section provides additional information and samples on using getter-objects.

12.5.2. Event Object

An event object is an `EventBean` that provides:

- The property value for a property given a property name or property expression that may include nested, indexed or mapped properties in any combination.
- The event type of the event.
- Access to the underlying event object.
- The `EventBean` fragment or array of `EventBean` fragments given a property name or property expression.

The `GetFragment` method on `EventBean` and `EventPropertyGetter` return the fragment `EventBean` or array of `EventBean`, if the property is itself an event or can be represented as an event. Your application may use `EventPropertyDescriptor` to determine which properties are also available as fragments.

The underlying event object of an `EventBean` can be obtained via the `Underlying` property. Please see Chapter 2, *Event Representations* for more information on different event representations.

From a threading perspective, it is safe to retain and query `EventBean` and `EventType` objects in multiple threads.

12.5.3. Query Example

Consider a statement that returns the symbol, count of events per symbol and average price per symbol for tick events. Our sample statement may declare a fully-qualified type name as the event type: `org.sample.StockTickEvent`. Assume that this type exists and exposes a `symbol` property of type `String`, and a `price` property of type (primitive) `double`.

```
select symbol, avg(price) as avgprice, count(*) as mycount
from org.sample.StockTickEvent
group by symbol
```

The next table summarizes the property names and types as posted by the statement above:

Table 12.4. Properties offered by sample statement aggregating price

Name	Type	Description	Code snippet
symbol	System.String	Value of symbol event property	<code>eventBean.Get["symbol"]</code>
avgprice	System.Double	Average price per symbol	<code>eventBean.Get["avgprice"]</code>
mycount	System.Int64	Number of events per symbol	<code>eventBean["mycount"]</code>

A code snippet out of a possible `UpdateEventHandler` implementation to this statement may look as below:

```
String symbol = (String) newEvents[0][ "symbol" ];
double? price = (double?) newEvents[0][ "avgprice" ];
long? count = (long?) newEvents[0][ "mycount" ];
```

The engine supplies the boxed `System.Double?` and `System.Int64?` types as property values rather than primitive types. This is because aggregated values can return a `null` value to indicate that no data is available for aggregation. Also, in a select statement that computes expressions, the underlying event objects to `EventBean` instances are of type `System.Collections.Generic.IDictionary`.

Consider the next statement that specifies a wildcard selecting the same type of event:

```
select * from org.sample.StockTickEvent where price > 100
```

The property names and types provided by an `EventBean` query result row, as posted by the statement above are as follows:

Table 12.5. Properties offered by sample wildcard-select statement

Name	Type	Description	Code snippet
symbol	System.String	Value of symbol event property	<code>eventBean["symbol"]</code>
price	double	Value of price event property	<code>eventBean["price"]</code>

As an alternative to querying individual event properties via the indexer or `Get` methods, the `Underlying` property on `EventBean` returns the underlying object representing the query result. In the sample statement that features a wildcard-select, the underlying event object is of type `org.sample.StockTickEvent`:

```
StockTickEvent tick = (StockTickEvent) newEvents[0].Underlying;
```

12.5.4. Pattern Example

Composite events are events that aggregate one or more other events. Composite events are typically created by the engine for statements that join two event streams, and for event patterns in which the causal events are retained and reported in a composite event. The example below shows such an event pattern.

```
// Look for a pattern where BEvent follows AEvent
String pattern = "a=AEvent -> b=BEvent";
EPStatement stmt = epService.EPAdministrator.CreatePattern(pattern);
stmt.Events += HandleEvent;
```

```
// Example event handler code
public void HandleEvent(Object sender, UpdateEventArgs e) {
    Console.WriteLine("a event={0}", e.NewData[0]["a"]);
    Console.WriteLine("b event={0}", e.NewData[0]["b"]);
}
```

Note that the `HandleEvent` method can receive multiple events at once encapsulated in the `UpdateEventArgs`. For example, a time batch window may post multiple events to event handlers representing a batch of events received during a given time period.

Pattern statements can also produce multiple events delivered to update event handlers in one invocation. The pattern statement below, for instance, delivers an event for each A event that was not followed by a B event with the same `id` property within 60 seconds of the A event. The engine may deliver all matching A events as an array of events in a single invocation of the `UpdateEventHandler` delegate associated with the statement:

```
select * from pattern[
    every a=A -> (timer:interval(60 sec) and not B(id=a.id))]
```

A code snippet out of a possible `UpdateEventHandler` implementation to this statement that retrieves the events as fragments may look as below:

```
EventBean a = (EventBean) newEvents[0].GetFragment("a");
// ... or using a nested property expression to get a value out of A event...
double value = (double) newEvent[0]["a.value"];
```

Some pattern objects return an array of events. An example is the unbound repeat operator. Here is a sample pattern that collects all A events until a B event arrives:

```
select * from pattern [a=A until b=B]
```

A possible code to retrieve different fragments or property values:

```
EventBean[] a = (EventBean[]) newEvents[0].GetFragment("a");
// ... or using a nested property expression to get a value out of A event...
double value = (double) newEvent[0]["a[0].value"];
```

12.6. Engine Threading and Concurrency

Esper is designed from the ground up to operate as a component to multi-threaded, highly-concurrent applications that require efficient use of resources. In addition, multi-threaded execution requires guarantees in predictability of results and deterministic processing. This section discusses these concerns in detail.

In Esper, an engine instance is a unit of separation. Applications can obtain and discard (initialize) one or more engine instances within the same application domain and can provide the same or different engine configurations to each instance. An engine instance efficiently shares resources between statements. For example, consider two statements that declare the same data window. The engine matches up view declarations provided by each statement and can thus provide a single data window representation shared between the two statements.

Applications can use Esper APIs to concurrently, by multiple threads of execution, perform such functions as creating and managing statements, or sending events into an engine instance for processing. Applications can

use application-managed threads or thread pools or any set of same or different threads of execution with any of the public Esper APIs. There are no restrictions towards threading other than those noted in specific sections of this document.

Esper does not prescribe a specific threading model. Applications using Esper retain full control over threading, allowing an engine to be easily embedded and used as a component or library in your favorite container or process.

In the default configuration it is up to the application code to use multiple threads for processing events by the engine, if so desired. All event processing takes places within your application thread call stack. The exception is timer-based processing if your engine instance relies on the internal timer (default). If your application relies on external timer events instead of the internal timer then there need not be any Esper-managed internal threads.

The fact that event processing can take place within your application thread's call stack makes developing applications with Esper easier: Any .NET integrated development environment (IDE) can host an Esper engine instance. This allows developers to easily set up test cases, debug through listener code and inspect input or output events, or trace their call stack.

In the default configuration, each engine instance maintains a single timer thread (internal timer) providing for time or schedule-based processing within the engine. The default resolution at which the internal timer operates is 100 milliseconds. The internal timer thread can be disabled and applications can instead send external time events to an engine instance to perform timer or scheduled processing at the resolution required by an application.

Each engine instance performs minimal locking to enable high levels of concurrency. An engine instance locks on a statement level to protect statement resources.

For an engine instance to produce predictable results from the viewpoint of event handlers to statements, an engine instance by default ensures that it dispatches statement result events to event handlers in the order in which a statement produced result events. Applications that require the highest possible concurrency and do not require predictable order of delivery of events to event handlers, this feature can be turned off via configuration, see Section 13.4.9.1, "Preserving the order of events delivered to listeners". For example, assume thread T1 processes an event applied to statement S producing output event O1. Assume thread T2 processes another event applied to statement S and produces output event O2. The engine employs a configurable latch system to ensure that event handlers to statement S receive and may complete processing of O1 before receiving O2. When using outbound threading (advanced threading options) or changing the configuration this guarantee is weakened or removed.

In multithreaded environments, when one or more statements make result events available via the `insert into` clause to further statements, the engine preserves the order of events inserted into the generated insert-into stream, allowing statements that consume other statement's events to behave deterministic. This feature can also be turned off via configuration, see , see Section 13.4.9.2, "Preserving the order of events for insert-into streams". For example, assume thread T1 processes an event applied to statement S and thread T2 processes another event applied to statement S. Assume statement S inserts into into stream ST. T1 produces an output event O1 for processing by consumers of ST and T2 produces an output event O2 for processing by consumers of ST. The engine employs a configurable latch system such that O1 is processed before O2 by consumers of ST. When using route execution threading (advanced threading options) or changing the configuration this guarantee is weakened or removed.

We generally recommended that listener implementations block minimally or do not block at all. By implementing listener code as non-blocking code execution threads can often achieve higher levels of concurrency.

We recommended that, when using a single listener or subscriber instance to receive output from multiple state-

ments, that the listener or subscriber code is multithread-safe. If your application has shared state between listener or subscriber instances then such shared state should be thread-safe.

12.6.1. Advanced Threading

In the default configuration the same application thread that invokes any of the `SendEvent` methods will process the event fully and also deliver output events to event handlers and subscribers. By default the single internal timer thread based on system time performs time-based processing and delivery of time-based results.

This default configuration reduces the processing overhead associated with thread context switching, is light-weight and fast and works well in many environments such as J2EE, server or client. Latency and throughput requirements are largely use case dependant, and Esper provides engine-level facilities for controlling concurrency that are described next.

Inbound Threading queues all incoming events: A pool of engine-managed threads performs the event processing. The application thread that sends an event via any of the `SendEvent` methods returns without blocking.

Outbound Threading queues events for delivery to event handlers and subscribers, such that slow or blocking event handlers or subscribers do not block event processing.

Timer Execution Threading means time-based event processing is performed by a pool of engine-managed threads. With this option the internal timer thread (or external timer event) serves only as a metronome, providing units-of-work to the engine-managed threads in the timer execution pool, pushing threading to the level of each statement for time-based execution.

Route Execution Threading means that the thread sending in an event via any of the `SendEvent` methods (or the inbound threading pooled thread if inbound threading is enabled) only identifies and pre-processes an event, and a pool of engine-managed threads handles the actual processing of the event for each statement, pushing threading to the level of each statement for event-arrival-based execution.

The engine starts engine-managed threads as daemon threads when the engine instance is first obtained. The engine stops engine-managed threads when the engine instance is destroyed via the `Dispose` method. When the engine is initialized via the `Initialize` method the existing engine-managed threads are stopped and new threads are created. When shutting down your application, use the `Dispose` method to stop engine-managed threads.

Note that the options discussed herein may introduce additional processing overhead into your system, as each option involves work queue management and thread context switching.

If your use cases require ordered processing of events or do not tolerate disorder, the threading options described herein may not be the right choice.

If your use cases require loss-less processing of events, wherein the threading options mean that events are held in an in-memory queue, the threading options described herein may not be the right choice.

Care should be taken to consider arrival rates and queue depth. Threading options utilize unbound queues or capacity-bound queues with blocking-put, depending on your configuration, and may therefore introduce an overload or blocking situation to your application. You may use the service provider interface as outlined below to manage queue sizes, if required, and to help tune the engine to your application needs. Consider throttling down the event send rate when the API (see below) indicates that events are getting queued.

All threading options are on the level of an engine. If you require different threading behavior for certain statements then consider using multiple engine instances, consider using the `Route` method or consider using application threads instead.

Please consult Section 13.4.9, “Engine Settings related to Concurrency and Threading” for instructions on how to configure threading options. Threading options take effect at engine initialization time.

Inbound Threading

With inbound threading an engine places inbound events in a queue for processing by one or more engine-managed threads other than the delivering application threads.

The delivering application thread uses one of the `SendEvent` methods on `EPRuntime` to deliver events or may also use the `SendEvent` method on a `EventSender`. The engine receives the event and places the event into a queue, allowing the delivering thread to continue and not block while the event is being processed and results are delivered.

Events that are sent into the engine via one of the `Route` methods are not placed into queue but processed by the same thread invoking the `Route` operation.

Outbound Threading

With outbound threading an engine places outbound events in a queue for delivery by one or more engine-managed threads other than the processing thread originating the result.

With outbound threading your listener or subscriber class receives statement results from one of the engine-managed threads in the outbound pool of threads. This is useful when you expect your listener or subscriber code to perform significantly blocking operations and you do not want to hold up event processing.

Timer Execution Threading

With timer execution threading an engine places time-based work units into a queue for processing by one or more engine-managed threads other than the internal timer thread or the application thread that sends an external timer event.

Using timer execution threading the internal timer thread (or thread delivering an external timer event) serves to evaluate which time-based work units must be processed. A pool of engine-managed threads performs the actual processing of time-based work units and thereby offloads the work from the internal timer thread (or thread delivering an external timer event).

Enable this option as a tuning parameter when your statements utilize time-based patterns or data windows. Timer execution threading is fine grained and works on the level of a time-based schedule in combination with a statement.

Route Execution Threading

With route execution threading an engine identifies event-processing work units based on the event and statement combination. It places such work units into a queue for processing by one or more engine-managed threads other than the thread that originated the event.

While inbound threading works on the level of an event, route execution threading is fine grained and works on the level of an event in combination with a statement.

Threading Service Provider Interface

The service-provider interface `EPServiceProviderSPI` is an extension API that allows to manage engine-level queues and thread pools .

The service-provider interface `EPServiceProviderSPI` is considered an extension API and subject to change between release versions.

The following code snippet shows how to obtain the `BlockingQueue<Runnable>` and the `ThreadPoolExecutor` for the managing the queue and thread pool responsible for inbound threading:

```
EPServiceProviderSPI spi = (EPServiceProviderSPI) epService;  
int queueSize = spi.getThreadingService().getInboundQueue().size();  
ThreadPoolExecutor threadpool = spi.getThreadingService().getInboundThreadPool();
```

12.6.2. Processing Order

Competing Statements

This section discusses the order in which N competing statements that all react to the same arriving event execute.

The engine, by default, does not guarantee to execute competing statements in any particular order unless using `@Priority`. We therefore recommend that an application does not rely on the order of execution of statements by the engine, since that best shields the behavior of an application from changes in the order that statements may get created by your application or by threading configurations that your application may change at will.

If your application requires a defined order of execution of competing statements, use the `@Priority` EPL syntax to make the order of execution between statements well-defined (requires that you set the prioritized-execution configuration setting). And the `@Drop` can make a statement preempt all other lowered priority ones that then won't get executed for any matching events.

Competing Events in a Work Queue

This section discusses the order of event evaluation when multiple events must be processed, for example when multiple statements use insert-into to generate further events upon arrival of an event.

The engine processes an arriving event completely before considering output events generated by insert-into or routed events inserted by event handlers or subscribers.

For example, assume three statements (1) `select * from MyEvent` and (2) `insert into ABCStream select * from MyEvent`. (3) `select * from ABCStream`. When a `MyEvent` event arrives then the event handlers to statements (1) and (2) execute first (default threading model). Event handlers to statement (3) which receive the inserted-in-to stream events are always executed after delivery of the triggering event.

Among all events generated by insert-into of statements and the events routed into the engine via the `Route` method, all events that insert-into a named window are processed first in the order generated. All other events are processed thereafter in the order they were generated.

When enabling timer or route execution threading as explained under advanced threading options then the engine does not make any guarantee to the processing order except that it will prioritize events inserted into a named window.

12.7. Controlling Time-Keeping

There are two modes for an engine to keep track of time: The internal timer based on JVM system time (the default), and externally-controlled time giving your application full control over the concept of time within an en-

gine or isolated service.

An isolated service is an execution environment separate from the main engine runtime, allowing full control over the concept of time for a group of statements, as further described in Section 12.9, “Service Isolation”.

By default the internal timer provides time and evaluates schedules. External clocking can be used to supply time ticks to the engine instead. The latter is useful for testing time-based event sequences or for synchronizing the engine with an external time source.

The internal timer relies on the windows multimedia timer (default) for time tick events. The windows multimedia timer provides the finest granularity available on a Windows based platform (1 millisecond) but requires integration of the multimedia DLL. The next section describes timer resolution for the internal timer, by default set to 100 milliseconds but is configurable via the threading options. When using externally-controlled time the timer resolution is in your control.

To disable the internal timer and use externally-provided time instead, there are two options. The first option is to use the configuration API at engine initialization time. The second option toggles on and off the internal timer at runtime, via special timer control events that are sent into the engine like any other event.

If using a timer execution thread pool as discussed above, the internal timer or external time event provide the schedule evaluation however do not actually perform the time-based processing. The time-based processing is performed by the threads in the timer execution thread pool.

This code snippet shows the use of the configuration API to disable the internal timer and thereby turn on externally-provided time (see the Configuration section for configuration via XML file):

```
Configuration config = new Configuration();
config.EngineDefaults.Threading.IsInternalTimerEnabled = false;
EPServiceProvider epService = EPServiceProviderManager.GetDefaultProvider(config);
```

After disabling the internal timer, it is wise to set a defined time so that any statements created thereafter start relative to the time defined. Use the `CurrentTimeEvent` class to indicate current time to the engine and to move time forward for the engine.

This code snippet obtains the current time and sends a timer event in:

```
long timeInMillis = DateTimeHelper.GetCurrentTimeMillis();
CurrentTimeEvent timeEvent = new CurrentTimeEvent(timeInMillis);
epService.EPRuntime.SendEvent(timeEvent);
```

Alternatively, you can use special timer control events to enable or disable the internal timer. Use the `TimerControlEvent` class to control timer operation at runtime.

The next code snippet demonstrates toggling to external timer at runtime, by sending in a `TimerControlEvent` event:

```
EPServiceProvider epService = EPServiceProviderManager.GetDefaultProvider();
EPRuntime runtime = epService.EPRuntime;
// switch to external clocking
runtime.SendEvent(new TimerControlEvent(TimerControlEvent.ClockType.CLOCK_EXTERNAL));
```

Your application sends a `CurrentTimeEvent` event when it desires to move the time forward. All aspects of Esper engine time related to EPL statements and patterns are driven by the time provided by the `CurrentTimeEvent` that your application sends in.

The next example sequence of instructions sets time to zero, then creates a statement, then moves time forward

to 1 seconds later and then 6 seconds later:

```
// Set start time at zero.
runtime.SendEvent(new CurrentTimeEvent(0));

// create a statement here
epAdministrator.CreateEPL("select * from MyEvent output every 5 seconds");

// move time forward 1 second
runtime.SendEvent(new CurrentTimeEvent(1000));

// move time forward 5 seconds
runtime.SendEvent(new CurrentTimeEvent(6000));
```

When sending external timer events, your application should make sure that `long`-type time values are ascending. That is, each `long`-type value should be either the same value or a larger value than the prior value provided by a `CurrentTimeEvent`. The engine outputs a warning if time events move back in time.

Your application may use the `NextScheduledTime` property in `EPRuntime` to determine the earliest time a schedule for any statement requires evaluation.

The following code snippet sets the current time, creates a statement and prints the next scheduled time which is 1 minute later than the current time:

```
// Set start time to the current time.
runtime.SendEvent(new CurrentTimeEvent(System.currentTimeMillis()));

// Create a statement.
epService.EPAdministrator.CreateEPL("select * from pattern[timer:interval(1 minute)]");

// Print next schedule time
Console.WriteLine("Next schedule at {0}", new Date(runtime.NextScheduledTime));
```

12.7.1. Controlling Time Using Time Span Events

With `CurrentTimeEvent`, as described above, your application can advance engine time to a given point in time. In addition, the `NextScheduledTime` property in `EPRuntime` returns the next scheduled time according to started statements. You would typically use `CurrentTimeEvent` to advance time at a relatively high resolution.

To advance time for a span of time without sending individual `CurrentTimeEvent` events to the engine, the API provides the class `CurrentTimeSpanEvent`. You may use `CurrentTimeSpanEvent` with or without a resolution.

If your application only provides the target end time of time span to `CurrentTimeSpanEvent` and no resolution, the engine advances time up to the target time by stepping through all relevant times according to started statements.

If your application provides the target end time of time span and in addition a `long`-typed resolution, the engine advances time up to the target time by incrementing time according to the resolution (regardless of next scheduled time according to started statements).

Consider the following example:

```
// Set start time to Jan.1, 2010, 00:00 am for this example
DateTime startTime = DateTime.Parse("2010 01 01 00:00:00 000", "yyyy MM dd HH:mm:ss FFF");
runtime.SendEvent(new CurrentTimeEvent(startTime.InMillis()));

// Create a statement.
EPStatement stmt = epService.EPAdministrator.CreateEPL("select current_timestamp() as ct " +
    "from pattern[every timer:interval(1 minute)]");
stmt.Events += ...; // add an event handler
```

```
// Advance time to 10 minutes after start time
runtime.SendEvent(new CurrentTimeSpanEvent(startTime.InMillis() + 10*60*1000));
```

The above example advances time to 10 minutes after the time set using `CurrentTimeSpanEvent`. As the example does not pass a resolution, the engine advances time according to statement schedules. Upon sending the `CurrentTimeSpanEvent` the listener sees 10 invocations for minute 1 to minute 10.

To advance time according to a given resolution, you may provide the resolution as shown below:

```
// Advance time to 10 minutes after start time at 100 msec resolution
runtime.SendEvent(new CurrentTimeSpanEvent(startTime.getTime() + 10*60*1000, 100));
```

12.7.2. Additional Time-Related APIs

Consider using the service-provider interface `EPRuntimeSPI` `EPRuntimeIsolatedSPI`. The two interfaces are service-provider interfaces that expose additional function to manage statement schedules. However the SPI interfaces should be considered an extension API and are subject to change between release versions.

Additional engine-internal SPI interfaces can be obtained by downcasting `EPServiceProvider` to `EPServiceProviderSPI`. For example the `SchedulingServiceSPI` exposes schedule information per statement (downcast from `SchedulingService`). Engine-internal SPI are subject to change between versions.

12.8. Time Resolution

The minimum resolution that all data windows, patterns and output rate limiting operate at is the millisecond. Parameters to time window views, pattern operators or the `output` clause that are less than 1 millisecond are not allowed. As stated earlier, the default frequency at which the internal timer operates is 100 milliseconds (configurable).

The internal timer thread, by default, uses the call `DateTime.Now` to obtain system time. Please see the JIRA issue [ESPER-191 Support nano/microsecond resolution](#) for more information on system time-call performance, accuracy and drift.

The internal timer thread can be configured to use nano-second time as returned by `com.espertech.esper.compat.HighResolutionTimeProvider`. If configured for nano-second time, the engine computes an offset of the nano-second ticks to wall clock time upon startup to present back an accurate millisecond wall clock time. Please see section [Section 13.4.15, "Engine Settings related to Time Source"](#) to configure the internal timer thread to use `HighPerformanceTimer`.

Your application can achieve a higher tick rate than 1 tick per millisecond by sending external timer events that carry a long-value which is not based on milliseconds since January 1, 1970, 00:00:00 GMT. In this case, your time interval parameters need to take consideration of the changed use of engine time.

Thus, if your external timer events send long values that represents microseconds (1E-6 sec), then your time window interval must be 1000-times larger, i.e. "win:time(1000)" becomes a 1-second time window.

And therefore, if your external timer events send long values that represents nanoseconds (1E-9 sec), then your time window interval must be 1000000-times larger, i.e. "win:time(1000000)" becomes a 1-second time window.

12.9. Service Isolation

12.9.1. Overview

An *isolated service* allows an application to control event visibility and the concept of time as desired on a statement level: Events sent into an isolated service are visible only to those statements that currently reside in the isolated service and are not visible to statements outside of that isolated service. Within an isolated service an application can control time independently, start time at a point in time and advance time at the resolution and pace suitable for the statements added to that isolated service.

As discussed before, a single application domain may hold multiple Esper engine instances unique by engine URI. Within an Esper engine instance the default execution environment for statements is the `EPRuntime` engine runtime, which coordinates all statement's reaction to incoming events and to time passing (via internal or external timer).

Subordinate to an Esper engine instance, your application can additionally allocate multiple isolated services (or execution environments), uniquely identified by a name and represented by the `EPServiceProviderIsolated` interface. In the isolated service, time passes only when you application sends timer events to the `EPRuntimeIsolated` instance. Only events explicitly sent to the isolated service are visible to statements added.

Your application can create new statements that start in an isolated service. You can also move existing statements back and forth between the engine and an isolated service.

Isolation does not apply to variables: Variables are global in nature. Also, as named windows are globally visible data windows, consumers to named windows see changes in named windows even though a consumer or the named window (through the create statement) may be in an isolated service.

An isolated service allows an application to:

1. Suspend a statement without losing its statement state that may have accumulated for the statement.
2. Control the concept of time separately for a set of statements, for example to simulate, backtest, adjust arrival order or compute arrival time.
3. Initialize statement state by replaying events, without impacting already running statements, to catch-up statements from historical events for example.

While a statement resides in an isolated runtime it receives only those events explicitly sent to the isolated runtime, and performs time-based processing based on the timer events provided to that isolated runtime.

Use the `GetEPServiceIsolated` method on `EPServiceProvider` passing a name to obtain an isolated runtime:

```
EPServiceProviderIsolated isolatedService = epServiceManager.GetEPServiceIsolated("name");
```

Set the start time for your isolated runtime via the `CurrentTimeEvent` timer event:

```
// In this example start the time at the system time
long startInMillis = DateTimeHelper.GetCurrentTimeMillis();
isolatedService.EPRuntime.SendEvent(new CurrentTimeEvent(startInMillis));
```

Use the `AddStatement` method on `EPAdministratorIsolated` to move an existing statement out of the engine runtime into the isolated runtime:

```
// look up the existing statement
EPStatement stmt = epServiceManager.EPAdministrator.GetStatement("MyStmt");

// move it to an isolated service
isolatedService.EPAdministrator.AddStatement(stmt);
```

To remove the statement from isolation and return the statement back to the engine runtime, use the `RemoveStatement` method on `EPAdministratorIsolated`:

```
isolatedService.EPAdministrator.RemoveStatement(stmt);
```

To create a new statement in the isolated service, use the `CreateEPL` method on `EPAdministratorIsolated`:

```
isolatedService.EPAdministrator.CreateEPL(
    "@Name('MyStmt') select * from Event", null, null);
// the example is passing the statement name in an annotation and no user object
```

The `Dispose` method on `EPServiceProviderIsolated` moves all currently-isolated statements for that isolated service provider back to engine runtime.

When moving a statement between engine runtime and isolated service or back, the algorithm ensures that events are aged according to the time that passed and time schedules stay intact.

To use isolated services, your configuration must have view sharing disabled as described in Section 13.4.11.1, “Sharing View Resources between Statements”.

12.9.2. Example: Suspending a Statement

By adding an existing statement to an isolated service, the statement's processing effectively becomes suspended. Time does not pass for the statement and it will not process events, unless your application explicitly moves time forward or sends events into the isolated service.

First, let's create a statement and send events:

```
EPStatement stmt = epServiceManager.EPAdministrator.CreateEPL("select * from TemperatureEvent.win:time");
epServiceManager.EPRuntime.Send(new TemperatureEvent(...));
// send some more events over time
```

The steps to suspend the previously created statement are as follows:

```
EPServiceProviderIsolated isolatedService = epServiceManager.GetEPServiceIsolated("suspendedStmts");
isolatedService.EPAdministrator.AddStatement(stmt);
```

To resume the statement, move the statement back to the engine:

```
isolatedService.EPAdministrator.RemoveStatement(stmt);
```

If the statement employed a time window, the events in the time window did not age. If the statement employed patterns, the pattern's time-based schedule remains unchanged. This is because the example did not advance time in the isolated service.

12.9.3. Example: Catching up a Statement from Historical Data

This example creates a statement in the isolated service, replays some events and advances time, then merges back the statement to the engine to let it participate in incoming events and engine time processing.

First, allocate an isolated service and explicitly set it to a start time. Assuming that `myStartTime` is a long milli-second time value that marks the beginning of the data to replay, the sequence is as follows:

```
EPServiceProviderIsolated isolatedService = epServiceManager.GetEPServiceIsolated("suspendedStmts");
isolatedService.EPRuntime.SendEvent(new CurrentTimeEvent(myStartTime));
```

Next, create the statement. The sample statement is a pattern statement looking for temperature events following each other within 60 seconds:

```
EPStatement stmt = epAdmin.CreateEPL(
    "select * from pattern[every a=TemperatureEvent -> b=TemperatureEvent where timer:within(60)]");
```

For each historical event to be played, advance time and send an event. This code snippet assumes that `currentTime` is a time greater than `myStartTime` and reflects the time that the historical event should be processed at. It also assumes `historyEvent` is the historical event object.

```
isolatedService.EPRuntime.SendEvent(new CurrentTimeEvent(currentTime));
isolatedService.EPRuntime.Send(historyEvent);
// repeat the above advancing time until no more events
```

Finally, when done replaying events, merge the statement back with the engine:

```
isolatedService.EPAdministrator.RemoveStatement(stmt);
```

12.9.4. Isolation for Insert-Into

When isolating statements, events that are generated by `insert into` are visible within the isolated service that currently holds that `insert into` statement.

For example, assume the below two statements named A and B:

```
@Name('A') insert into MyStream select * from MyEvent
@Name('B') select * from MyStream
```

When adding statement A to an isolated service, and assuming a `MyEvent` is sent to either the engine runtime or the isolated service, a listener to statement B does not receive that event.

When adding statement B to an isolated service, and assuming a `MyEvent` is sent to either the engine runtime or the isolated service, a listener to statement B does not receive that event.

12.9.5. Isolation for Named Windows

When isolating named windows, the event visibility of events entering and leaving from a named window is not limited to the isolated service. This is because named windows are global data windows (a relation in essence).

For example, assume the below three statements named A, B and C:

```
@Name('A') create window MyNamedWindow.win:time(60) as select * from MyEvent
@Name('B') insert into MyNamedWindow select * from MyEvent
@Name('C') select * from MyNamedWindow
```

When adding statement A to an isolated service, and assuming a `MyEvent` is sent to either the engine runtime or the isolated service, a listener to statement A and C does not receive that event.

When adding statement B to an isolated service, and assuming a `MyEvent` is sent to either the engine runtime or the isolated service, a listener to statement A and C does not receive that event.

When adding statement C to an isolated service, and assuming a `MyEvent` is sent to the engine runtime, a listener to statement A and C does receive that event.

12.9.6. Runtime Considerations

Moving statements between an isolated service and the engine is an expensive operation and should not be performed with high frequency.

When using multiple threads to send events and at the same time moving a statement to an isolated service, it is undefined whether events will be delivered to a listener of the isolated statement until all threads completed sending events.

Metrics reporting is not available for statements in an isolated service. Advanced threading options are also not available in the isolated service, however it is thread-safe to send events including timer events from multiple threads to the same or different isolated service.

12.10. Exception Handling

You may register one or more exception handlers for the engine to invoke in the case it encounters an exception processing a continuously-executing statement. By default and without exception handlers the engine cancels execution of the current EPL statement that encountered the exception, logs the exception and continues to the next statement, if any. The configuration is described in Section 13.4.20, “Engine Settings related to Exception Handling”.

If your application registers exception handlers as part of engine configuration, the engine invokes the exception handlers in the order they are registered passing relevant exception information such as EPL statement name, expression and the exception itself.

Exception handlers receive any EPL statement unchecked exception such as internal exceptions or exceptions thrown by plug-in aggregation functions or plug-in views. The engine does not provide to exception handlers any exceptions thrown by static method invocations for function calls, method invocations in joins, methods on event classes and event handlers or subscriber exceptions.

An exception handler can itself throw a runtime exception to cancel execution of the current event against any further statements.

For on-demand queries the API indicates any exception directly back to the caller without the exception handlers being invoked, as exception handlers apply to continuous queries only. The same applies to any API calls other than `SendEvent` and the `EventSender` methods.

As the configuration section describes, your application registers one or more classes that implement the `ExceptionHandlerFactory` interface in the engine configuration. Upon engine initialization the engine obtains a factory instance from the class name that then provides the exception handler instance. The exception handler class must implement the `ExceptionHandler` interface.

12.11. Condition Handling

You may register one or more condition handlers for the engine to invoke in the case it encounters certain con-

ditions, as outlined below, when executing a statement. By default and without condition handlers the engine logs the condition at informational level and continues processing. The configuration is described in Section 13.4.21, “Engine Settings related to Condition Handling”.

If your application registers condition handlers as part of engine configuration, the engine invokes the condition handlers in the order they are registered passing relevant condition information such as EPL statement name, expression and the condition information itself.

Currently the only condition indicated by this facility is raised by the pattern followed-by operator when used with the limit expression and when the limit is reached, see Section 5.5.8.1, “Limiting Sub-Expression Count”.

A condition handler may not itself throw a runtime exception or return any value.

As the configuration section describes, your application registers one or more classes that implement the `ConditionHandlerFactory` interface in the engine configuration. Upon engine initialization the engine obtains a factory instance from the class name that then provides the condition handler instance. The condition handler class must implement the `ConditionHandler` interface.

12.12. Statement Object Model

The statement object model is a set of classes that provide an object-oriented representation of an EPL or pattern statement. The object model classes are found in package `com.espertech.esper.client.soda`. An instance of `EPStatementObjectModel` represents a statement's object model.

The statement object model classes are a full and complete specification of a statement. All EPL and pattern constructs including expressions and sub-queries are available via the statement object model.

In conjunction with the administrative API, the statement object model provides the means to build, change or interrogate statements beyond the EPL or pattern syntax string representation. The object graph of the statement object model is fully navigable for easy querying by code, and is also serializable allowing applications to persist or transport statements in object form, when required.

The statement object model supports full round-trip from object model to EPL statement string and back to object model: A statement object model can be rendered into an EPL string representation via the `toEPL` method on `EPStatementObjectModel`. Further, the administrative API allows to compile a statement string into an object model representation via the `CompileEPL` method on `EPAdministrator`.

The `Create` method on `EPAdministrator` creates and starts a statement as represented by an object model. In order to obtain an object model from an existing statement, obtain the statement expression text of the statement via the `Text` property on `EPStatement` and use the `CompileEPL` method to obtain the object model.

The following limitations apply:

- Statement object model classes are not safe for sharing between threads other than for read access.
- Between versions of Esper, the serialized form of the object model is subject to change. Esper makes no guarantees that the serialized object model of one version will be fully compatible with the serialized object model generated by another version of Esper. Please consider this issue when storing Esper object models in persistent store.

12.12.1. Building an Object Model

A `EPStatementObjectModel` consists of an object graph representing all possible clauses that can be part of an

EPL statement.

Among all clauses, the `SelectClause` and `FromClause` objects are required clauses that must be present, in order to define what to select and where to select from.

Table 12.6. Required Statement Object Model Instances

Class	Description
<i>EPStatementObjectModel</i>	All statement clauses for a statement, such as the select-clause and the from-clause, are specified within the object graph of an instance of this class
<i>SelectClause</i>	A list of the selection properties or expressions, or a wildcard
<i>FromClause</i>	A list of one or more streams; A stream can be a filter-based, a pattern-based or a SQL-based stream; Views are added to streams to provide data window or other projections

Part of the statement object model package are convenient builder classes that make it easy to build a new object model or change an existing object model. The `SelectClause` and `FromClause` are such builder classes and provide convenient `Create` methods.

Within the from-clause we have a choice of different streams to select on. The `FilterStream` class represents a stream that is filled by events of a certain type and that pass an optional filter expression.

We can use the classes introduced above to create a simple statement object model:

```
EPStatementObjectModel model = new EPStatementObjectModel();
model.SelectClause = SelectClause.CreateWildcard();
model.FromClause = FromClause.Create(FilterStream.Create("com.chipmaker.ReadyEvent"));
```

The model as above is equivalent to the EPL :

```
select * from com.chipmaker.ReadyEvent
```

Last, the code snippet below creates a statement from the object model:

```
EPStatement stmt = epService.EPAdministrator.Create(model);
```

Notes on usage:

- Variable names can simply be treated as property names.
- When selecting from named windows, the name of the named window is the event type name for use in `FilterStream` instances or patterns.
- To compile an arbitrary sub-expression text into an `Expression` object representation, simply add the expression text to a `where` clause, compile the EPL string into an object model via the `compileEPL` on `EPAdministrator`, and obtain the compiled `where` from the `EPStatementObjectModel` via the `WhereClause` property.

12.12.2. Building Expressions

The `EPStatementObjectModel` includes an optional `where`-clause. The `where`-clause is a filter expression that the engine applies to events in one or more streams. The key interface for all expressions is the `Expression` in-

terface.

The `Expressions` class provides a convenient way of obtaining `Expression` instances for all possible expressions. Please consult the SDK documentatin for detailed method information. The next example discusses sample where-clause expressions.

Use the `Expressions` class as a service for creating expression instances, and add additional expressions via the `Add` method that most expressions provide.

In the next example we add a simple where-clause to the EPL as shown earlier:

```
select * from com.chipmaker.ReadyEvent where line=8
```

And the code to add a where-clause to the object model is below.

```
model.WhereClause = Expressions.Eq("line", 8);
```

The following example considers a more complex where-clause. Assume we need to build an expression using logical-and and logical-or:

```
select * from com.chipmaker.ReadyEvent
where (line=8) or (line=10 and age<5)
```

The code for building such a where-clause by means of the object model classes is:

```
model.WhereClause = Expressions.Or()
    .Add(Expressions.Eq("line", 8))
    .Add(Expressions.And()
        .Add(Expressions.Eq("line", 10))
        .Add(Expressions.Lt("age", 5))
    );
```

12.12.3. Building a Pattern Statement

The `Patterns` class is a factory for building pattern expressions. It provides convenient methods to create all pattern expressions of the pattern language.

Patterns in EPL are seen as a stream of events that consist of patterns matches. The `PatternStream` class represents a stream of pattern matches and contains a pattern expression within.

For instance, consider the following pattern statement.

```
select * from pattern [every a=MyAEvent and not b=MyBEvent]
```

The next code snippet outlines how to use the statement object model and specifically the `Patterns` class to create a statement object model that is equivalent to the pattern statement above.

```
EPStatementObjectModel model = new EPStatementObjectModel();
model.SelectClause = SelectClause.CreateWildcard();
PatternExpr pattern = Patterns.And()
    .Add(Patterns.EveryFilter("MyAEvent", "a"))
    .Add(Patterns.NotFilter("MyBEvent", "b"));
model.FromClause = FromClause.Create(PatternStream.Create(pattern));
```

12.12.4. Building a Select Statement

In this section we build a complete example statement and include all optional clauses in one EPL statement, to demonstrate the object model API.

A sample statement:

```
insert into ReadyStreamAvg(line, avgAge)
select line, avg(age) as avgAge
from com.chipmaker.ReadyEvent(line in (1, 8, 10)).win:time(10) as RE
where RE.waverId != null
group by line
having avg(age) < 0
output every 10.0 seconds
order by line
```

Finally, this code snippet builds the above statement from scratch:

```
EPStatementObjectModel model = new EPStatementObjectModel();
model.InsertInto = InsertIntoClause.create("ReadyStreamAvg", "line", "avgAge");
model.SelectClause = SelectClause.Create()
    .Add("line")
    .Add(Expressions.Avg("age"), "avgAge");
Filter filter = Filter.Create("com.chipmaker.ReadyEvent", Expressions.In("line", 1, 8, 10));
model.FromClause = FromClause.Create(
    FilterStream.Create(filter, "RE").AddView("win", "time", 10));
model.WhereClause = Expressions.IsNotNull("RE.waverId");
model.GroupByClause = GroupByClause.Create("line");
model.HavingClause = Expressions.Lt(Expressions.Avg("age"), Expressions.Constant(0));
model.OutputLimitClause = OutputLimitClause.Create(OutputLimitSelector.DEFAULT, Expressions.TimePeriod);
model.OrderByClause = OrderByClause.Create("line");
```

12.12.5. Building a Create-Variable and On-Set Statement

This sample statement creates a variable:

```
create variable integer var_output_rate = 10
```

The code to build the above statement using the object model:

```
EPStatementObjectModel model = new EPStatementObjectModel();
model.CreateVariable = CreateVariableClause.Create("integer", "var_output_rate", 10));
epService.EPAdministrator.Create(model);
```

A second statement sets the variable to a new value:

```
on NewValueEvent set var_output_rate = new_rate
```

The code to build the above statement using the object model:

```
EPStatementObjectModel model = new EPStatementObjectModel();
model.OnExpr = OnClause.CreateOnSet("var_output_rate", Expressions.Property("new_rate"));
model.FromClause = FromClause.Create(FilterStream.Create("NewValueEvent"));
EPStatement stmtSet = epService.EPAdministrator.Create(model);
```

12.12.6. Building Create-Window, On-Delete and On-Select Statements

This sample statement creates a named window:

```
create window OrdersTimeWindow.win:time(30 sec) as select symbol as sym, volume as vol, price from Orders
```

The is the code that builds the create-window statement as above:

```
EPStatementObjectModel model = new EPStatementObjectModel();
model.CreateWindow = CreateWindowClause.Create("OrdersTimeWindow");
model.CreateWindow.AddView("win", "time", 30);
model.SelectClause = SelectClause.Create()
    .AddWithName("symbol", "sym")
    .AddWithName("volume", "vol")
    .Add("price");
model.FromClause = FromClause.Create(FilterStream.Create("OrderEvent"));
```

A second statement deletes from the named window:

```
on NewOrderEvent as myNewOrders
delete from AllOrdersNamedWindow as myNamedWindow
where myNamedWindow.symbol = myNewOrders.symbol
```

The object model is built by:

```
EPStatementObjectModel model = new EPStatementObjectModel();
model.OnExpr = OnClause.CreateOnDelete("AllOrdersNamedWindow", "myNamedWindow");
model.FromClause = FromClause.Create(FilterStream.Create("NewOrderEvent", "myNewOrders"));
model.WhereClause = Expressions.EqProperty("myNamedWindow.symbol", "myNewOrders.symbol");
EPStatement stmtOnDelete = epService.EPAdministrator.Create(model);
```

A third statement selects from the named window using the non-continuous on-demand selection via on-select:

```
on QueryEvent(volume>0) as query
select count(*) from OrdersNamedWindow as win
where win.symbol = query.symbol
```

The on-select statement is built from scratch via the object model as follows:

```
EPStatementObjectModel model = new EPStatementObjectModel();
model.OnExpr = OnClause.CreateOnSelect("OrdersNamedWindow", "win");
model.WhereClause = Expressions.EqProperty("win.symbol", "query.symbol");
model.FromClause = FromClause.Create(FilterStream.Create("QueryEvent", "query",
    Expressions.Gt("volume", 0)));
model.SelectClause = SelectClause.Create().Add(Expressions.CountStar());
EPStatement stmtOnSelect = epService.EPAdministrator.Create(model);
```

12.13. Prepared Statement and Substitution Parameters

The `Prepare` method that is part of the administrative API pre-compiles an EPL statement and stores the pre-compiled statement in an `EPPreparedStatement` object. This object can then be used to efficiently start the parameterized statement multiple times.

Substitution parameters are inserted into an EPL statement as a single question mark character '?'. The engine assigns the first substitution parameter an index of 1 and subsequent parameters increment the index by one.

Substitution parameters can be inserted into any EPL construct that takes an expression. They are therefore valid in any clauses such as the select-clause, from-clause filters, where-clause, group-by-clause, having-clause or order-by-clause, including view parameters and pattern observers and guards. Substitution parameters cannot be used where a numeric constant is required rather than an expression and in SQL statements.

All substitution parameters must be replaced by actual values before a statement with substitution parameters can be started. Substitution parameters can be replaced with an actual value using the `SetObject` method for each index. Substitution parameters can be set to new values and new statements can be created from the same

`EPPreparedStatement` object more than once.

While the `SetObject` method allows substitution parameters to assume any actual value including application CLR objects or enumeration values, the application must provide the correct type of substitution parameter that matches the requirements of the expression the parameter resides in.

In the following example of setting parameters on a prepared statement and starting the prepared statement, `epService` represents an engine instance:

```
String stmt = "select * from com.chipmaker.ReadyEvent(line=?)";
EPPreparedStatement prepared = epService.EPAdministrator.PrepareEPL(stmt);
prepared.SetObject(1, 8);
EPStatement statement = epService.EPAdministrator.Create(prepared);
```

12.14. Engine and Statement Metrics Reporting

Metrics reporting is a feature that allows an application to receive ongoing reports about key engine-level and statement-level metrics. Examples are the number of incoming events, the CPU time and wall time taken by statement executions or the number of output events per statement.

Metrics reporting is, by default, disabled. To enable reporting, please follow the steps as outlined in Section 13.4.16, “Engine Settings related to Metrics Reporting”. Metrics reporting must be enabled at engine initialization time. Reporting intervals can be controlled at runtime via the `ConfigurationOperations` interface available from the administrative API.

Your application receives metrics at configurable intervals via EPL statement. A metric datapoint is simply a well-defined event. The events are `EngineMetric` and `StatementMetric` and the type representing the events can be found in the client API in package `com.espertech.esper.client.metric`.

Since metric events are processed by the engine the same as application events, your EPL may use any construct on such events. For example, your application may select, filter, aggregate properties, sort or insert into a stream or named window all metric events the same as application events.

This example statement selects all engine metric events:

```
select * from com.espertech.esper.client.metric.EngineMetric
```

The next statement selects all statement metric events:

```
select * from com.espertech.esper.client.metric.StatementMetric
```

Make sure to have metrics reporting enabled since only then do event handlers or subscribers to a statement such as above receive metric events.

The engine provides metric events after the configured interval of time has passed. By default, only started statements that have activity within an interval (in the form of event or timer processing) are reported upon.

The default configuration performs the publishing of metric events in an Esper daemon thread under the control of the engine instance. Metrics reporting honors externally-supplied time, if using external timer events.

Via runtime configuration options provided by `ConfigurationOperations`, your application may enable and disable metrics reporting globally, provided that metrics reporting was enabled at initialization time. Your application may also enable and disable metrics reporting for individual statements by statement name.

Statement groups is a configuration feature that allows to assign reporting intervals to statements. Statement

groups are described further in the Section 13.4.16, “Engine Settings related to Metrics Reporting” section. Statement groups cannot be added or removed at runtime.

The following limitations apply:

- High-performance numbers are obtained through the use of a PerformanceObserver that uses the `Kernel32` DLL. While this method has been exhaustively tested on Windows, we are aware that this will not work in non-Windows environments. To ensure compatability with Mono, we have included a model based on `DateTime`. This implementation will not provide the level of granularity or clarity that the Windows implementation will.
- There is a performance cost to collecting and reporting metrics.
- Not all statements may report metrics: The engine performs certain runtime optimizations sharing resources between similar statements, thereby not reporting on certain statements unless resource sharing is disabled through configuration.

12.14.1. Engine Metrics

Engine metrics are properties of `EngineMetric` events:

Table 12.7. EngineMetric Properties

Name	Description
<code>engineURI</code>	The URI of the engine instance.
<code>timestamp</code>	The current engine time.
<code>inputCount</code>	Cumulative number of input events since engine initialization time. Input events are defined as events send in via application threads as well as <code>insert into</code> events.
<code>inputCountDelta</code>	Number of input events since last reporting period.
<code>scheduleDepth</code>	Number of outstanding schedules.

12.14.2. Statement Metrics

Statement metrics are properties of `StatementMetric`. The properties are:

Table 12.8. StatementMetric Properties

Name	Description
<code>engineURI</code>	The URI of the engine instance.
<code>timestamp</code>	The current engine time.
<code>statementName</code>	Statement name, if provided at time of statement creation, otherwise a generated name.
<code>cpuTime</code>	Statement processing CPU time (system and user) in nanoseconds.
<code>wallTime</code>	Statement processing wall time in nanoseconds (based on <code>System.nanoTime</code>).
<code>numInput</code>	Number of input events to the statement.
<code>numOutputIStream</code>	Number of insert stream rows output to event handlers or the subscriber, if any.

Name	Description
numOutputRStream	Number of remove stream rows output to event handlers or the subscriber, if any.

The totals reported are cumulative relative to the last metric report.

12.15. Event Rendering to XML and JSON

Your application may use the built-in XML and JSON formatters to render output events into a readable textual format, such as for integration or debugging purposes. This section introduces the utility classes in the client `util` package for rendering events to strings. Further API information can be found in the SDK documentation.

The `IEventRenderer` interface accessible from the runtime interface via the `GetEventRenderer` method provides access to JSON and XML rendering. For repeated rendering of events of the same event type or sub-types, it is recommended to obtain a `JSONEventRenderer` or `XMLEventRenderer` instance and use the `Render` method provided by the interface. This allows the renderer implementations to cache event type metadata for fast rendering.

In this example we show how one may obtain a renderer for repeated rendering of events of the same type, assuming that `statement` is an instance of `EPStatement`:

```
JSONEventRenderer jsonRenderer = epService.EPRuntime.  
    EventRenderer.GetJSONRenderer(statement.EventType);
```

Assuming that `event` is an instance of `EventBean`, this code snippet renders an event into the JSON format:

```
String jsonEventText = jsonRenderer.Render("MyEvent", @event);
```

The XML renderer works the same:

```
XMLEventRenderer xmlRenderer = epService.EPRuntime.  
    EventRenderer.GetXMLRenderer(statement.EventType);
```

...and...

```
String xmlEventText = xmlRenderer.Render("MyEvent", @event);
```

If the event type is not known in advance or if your application does not want to obtain a renderer instance per event type for fast rendering, your application can use one of the following methods to render an event to a XML or JSON textual format:

```
String json = epService.EPRuntime.EventRenderer.RenderJSON(@event);  
String xml = epService.EPRuntime.EventRenderer.RenderXML(@event);
```

Use the `JSONRenderingOptions` or `XMLRenderingOptions` classes to control how events are rendered.

12.15.1. JSON Event Rendering Conventions and Options

The JSON renderer produces JSON text according to the standard documented at <http://www.json.org>.

The renderer formats simple properties as well as nested properties and indexed properties according to the JSON string encoding, array encoding and nested object encoding requirements.

The renderer does render indexed properties, it does not render indexed properties that require an index, i.e. if your event representation is backed by native objects and your getter method is `getValue(int index)`, the indexed property values are not part of the JSON text. This is because the implementation has no way to determine how many index keys there are. A workaround is to have a method such as `Object[] getValue()` instead.

The same is true for mapped properties that the renderer also renders. If a property requires a Map key for access, i.e. your getter method is `getValue(String key)`, such property values are not part of the result text as there is no way for the implementation to determine the key set.

12.15.2. XML Event Rendering Conventions and Options

The XML renderer produces well-formed XML text according to the XML standard.

The renderer can be configured to format simple properties as attributes or as elements. Nested properties and indexed properties are always represented as XML sub-elements to the root or parent element.

The root element name provided to the XML renderer must be the element name of the root in the XML document and may include namespace instructions.

The renderer does render indexed properties, it does not render indexed properties that require an index, i.e. if your event representation is backed by native objects and your getter method is `GetValue(int index)`, the indexed property values are not part of the XML text. This is because the implementation has no way to determine how many index keys there are. A workaround is to have a method such as `Object[] GetValue()` instead.

The same is true for mapped properties that the renderer also renders. If a property requires a Map key for access, i.e. your getter method is `GetValue(String key)`, such property values are not part of the result text as there is no way for the implementation to determine the key set.

12.16. Plug-in Loader

A plug-in loader is for general use with input adapters, output adapters or EPL code deployment or any other task that can benefit from being part of an Esper configuration file and that follows engine lifecycle.

A plug-in loader implements the `com.espertech.esper.plugin.PluginLoader` interface and can be listed in the configuration.

Each configured plug-in loader follows the engine instance lifecycle: When an engine instance initializes, it instantiates each `PluginLoader` implementation class listed in the configuration. The engine then invokes the lifecycle methods of the `PluginLoader` implementation class before and after the engine is fully initialized and before an engine instance is destroyed.

Declare a plug-in loader in your configuration XML as follows:

```
...
<plugin-loader name="MyLoader" class-name="org.mypackage.MyLoader">
  <init-arg name="property1" value="value1"/>
</plugin-loader>
...
```

Alternatively, add the plug-in loader via the configuration API:

```
Configuration config = new Configuration();
var props = new Dictionary<string, string>();
props["property1"] = "value1";
config.AddPluginLoader("MyLoader", "org.mypackage.MyLoader", props);
```

Implement the `Init` method of your `PluginLoader` implementation to receive initialization parameters. The engine invokes this method before the engine is fully initialized, therefore your implementation should not yet rely on the engine instance within the method body:

```
public class MyPluginLoader : PluginLoader {
    public void Init(String loaderName, Properties properties, EPServiceProviderSPI epService) {
        // save the configuration for later, perform checking
    }
    ...
}
```

The engine calls the `PostInitialize` method once the engine completed initialization and to indicate the engine is ready for traffic.

```
public void PostInitialize() {
    // Start the actual interaction with external feeds or the engine here
}
...
```

The engine calls the `Dispose` method once the engine is destroyed or initialized for a second time.

```
public void Dispose() {
    // Destroy resources allocated as the engine instance is being destroyed
}
```

12.17. Interrogating EPL Annotations

As discussed in Section 4.2.6, “Annotation” an EPL annotation is an addition made to information in an EPL statement. The API and examples to interrogate annotations are described here.

You may use the `getAnnotations` method of `EPStatement` to obtain annotations specified for an EPL statement. Or when compiling an EPL expression to a `EPStatementObjectModel` statement object model you may also query, change or add annotations.

The following example code demonstrates iterating over an `EPStatement` statement's annotations and retrieving values:

```
String exampleEPL = "@Tag(name='direct-output', value='sink 1') select * from RootEvent";
EPStatement stmt = epService.EPAdministrator.CreateEPL(exampleEPL);
foreach (Attribute attribute in stmt.Attributes) {
    TagAttribute tag = attribute as TagAttribute;
    if (tag != null) {
        Console.WriteLine("Tag name {0}, value {1}", tag.Name, tag.Value);
    }
}
```

The output of the sample code shown above is Tag name direct-output value sink 1.

12.18. CLR-Java Differences

When using Esper under .NET, one is sure to run into some differences that make it noticeably different than Java. For example, Java does not have an explicit notion of properties; however, this concept is native to the CLR. Another difference is in how events are handled; in Java, it is common to expose an interface that is implemented by the event receiver; in the CLR, it is common to expose an event handler that requires a delegate that the event receiver provides.

Chapter 13. Configuration

Esper engine configuration is entirely optional. Esper has a very small number of configuration parameters that can be used to simplify event pattern and EPL statements, and to tune the engine behavior to specific requirements. The Esper engine works out-of-the-box without configuration.

An application can supply configuration at the time of engine allocation using the `Configuration` class, and can also use XML files to hold configuration. Configuration can be changed at runtime via the `ConfigurationOperations` interface available from `EPAdministrator` via the `GetConfiguration` method.

The difference between using a `Configuration` object and the `ConfigurationOperations` interface is that for the latter, all configuration including event types added through that interface are considered runtime configurations. This means they will be discarded when calling the `Initialize` method on an `EPServiceProvider` instance.

13.1. Programmatic Configuration

An instance of `com.espertech.esper.client.Configuration` represents all configuration parameters. The `Configuration` is used to build an `EPServiceProvider`, which provides the administrative and runtime interfaces for an Esper engine instance.

You may obtain a `Configuration` instance by instantiating it directly and adding or setting values on it. The `Configuration` instance is then passed to `EPServiceProviderManager` to obtain a configured Esper engine.

```
Configuration configuration = new Configuration();
configuration.AddEventType<PriceLimit>("PriceLimit");
configuration.AddEventType<StockTick>("StockTick");
configuration.AddImport("org.mycompany.mypackage.MyUtility");
configuration.AddImport("org.mycompany.util");

EPServiceProvider epService = EPServiceProviderManager.GetProvider("sample", configuration);
```

Note that `Configuration` is meant only as an initialization-time object. The Esper engine represented by an `EPServiceProvider` does not retain any association back to the `Configuration`.

The `ConfigurationOperations` interface provides runtime configuration options as further described in Section 12.3.7, “Runtime Configuration”. Through this interface applications can, for example, add new event types at runtime and then create new statements that rely on the additional configuration. The `GetConfiguration` method on `EPAdministrator` allows access to `ConfigurationOperations`.

13.2. Configuration via XML File

An alternative approach to configuration is to specify a configuration in a XML file.

The default name for the XML configuration file is `esper.cfg.xml`. Esper reads this file from the root of the `CLASSPATH` as an application resource via the `Configure` method.

```
Configuration configuration = new Configuration();
configuration.Configure();
```

The `Configuration` class can read the XML configuration file from other sources as well. The `Configure` method accepts `URL`, `File` and `String` filename parameters.

```
Configuration configuration = new Configuration();
configuration.Configure("myengine.esper.cfg.xml");
```

13.3. XML Configuration File

Here is an example configuration file. The schema for the configuration file can be found in the `etc` folder and is named `esper-configuration-4-0.xsd`. It is also available online at <http://www.espertech.com/schema/esper/esper-configuration-2.0.xsd> so that IDE can fetch it automatically. The namespace used is `http://www.espertech.com/schema/esper`.

```
<?xml version="1.0" encoding="UTF-8"?>
<esper-configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.espertech.com/schema/esper"
  xsi:schemaLocation="
http://www.espertech.com/schema/esper
http://www.espertech.com/schema/esper/esper-configuration-2.0.xsd">
  <event-type name="StockTick" class="com.espertech.esper.example.stockticker.event.StockTick"/>
  <event-type name="PriceLimit" class="com.espertech.esper.example.stockticker.event.PriceLimit"/>
  <auto-import import-name="org.mycompany.mypackage.MyUtility"/>
  <auto-import import-name="org.mycompany.util.*"/>
</esper-configuration>
```

The example above is only a subset of the configuration items available. The next chapters outline the available configuration in greater detail.

13.4. Configuration Items

13.4.1. Events represented by CLR types

Package of CLR Event Classes

Via this configuration an application can make the Java package or packages that contain an application's Java event classes known to an engine. Thereby an application can simply refer to event types in statements by using the simple class name of each CLR type representing an event type.

For example, consider an order-taking application that places all event classes in package `com.mycompany.order.event`. One CLR type representing an event is the class `OrderEvent`. The application can simply issue a statement as follows to select `OrderEvent` events:

```
select * from OrderEvent
```

The XML configuration for defining the namespaces that contain CLR event types is:

```
<event-type-auto-name package-name="com.mycompany.order.event"/>
```

The same configuration but using the `Configuration` class:

```
Configuration config = new Configuration();
config.AddEventTypeAutoName("com.mycompany.order.events");
// ... or ...
config.AddEventTypeAutoName(typeof(MyEvent).Namespace);
```

Event type name to CLR type mapping

This configuration item can be used to allow event pattern statements and EPL statements to use an event type name rather than the fully qualified CLR type name. Note that interface classes and abstract classes are also supported as event types via the fully qualified CLR type name, and an event type name can also be defined for such classes.

The example pattern statement below first shows a pattern that uses the name `StockTick`. The second pattern statement is equivalent but specifies the fully-qualified CLR type name.

```
every StockTick(symbol='IBM')"
```

```
every com.espertech.esper.example.stockticker.event.StockTick(symbol='IBM')
```

The event type name can be listed in the XML configuration file as shown below. The Configuration API can also be used to programmatically specify an event type name, as shown in an earlier code snippet.

```
<event-type name="StockTick" class="com.espertech.esper.example.stockticker.event.StockTick"/>
```

Legacy Event Classes

Esper can process CLR types that provide event properties through other means than through JavaBean-style getter methods. It is not necessary that the method and member variable names in your CLR type adhere to the JavaBean convention - any public methods and public member variables can be exposed as event properties via the below configuration.

A CLR type can optionally be configured with an accessor style attribute. This attribute instructs the engine how it should expose methods and fields for use as event properties in statements.

Table 13.1. Accessor Styles

Style Name	Description
native	As the default setting, the engine exposes an event property for each public readable property and for all methods that expose properties according to the simple property publisher specification.
public	The engine exposes an event property for each public method and public member variable of the given class
explicit	The engine exposes an event property only for the explicitly configured public methods and public member variables

Using the `public` setting for the `accessor-style` attribute instructs the engine to expose an event property for each public method and public member variable of a CLR type. The engine assigns event property names of the same name as the name of the method or member variable in the CLR type.

For example, assuming the class `MyLegacyEvent` exposes a method named `readValue` and a member variable named `myField`, we can then use properties as shown.

```
select readValue, myField from MyLegacyEvent
```

Using the `explicit` setting for the `accessor-style` attribute requires that event properties are declared via configuration. This is outlined in the next chapter.

When configuring an engine instance from a XML configuration file, the XML snippet below demonstrates the use of the `legacy-type` element and the `accessor-style` attribute.

```
<event-type name="MyLegacyEvent" class="com.mycompany.mypackage.MyLegacyEventClass">
  <legacy-type accessor-style="public"/>
</event-type>
```

When configuring an engine instance via Configuration API, the sample code below shows how to set the accessor style.

```
Configuration configuration = new Configuration();
ConfigurationEventTypeLegacy legacyDef = new ConfigurationEventTypeLegacy();
legacyDef.setAccessorStyle(ConfigurationEventTypeLegacy.AccessorStyle.PUBLIC);
config.AddEventType("MyLegacyEvent", MyLegacyEventClass.class.getName(), legacyDef);

EPServiceProvider epService = EPServiceProviderManager.getProvider("sample", configuration);
```

Specifying Event Properties for CLR Types

Sometimes it may be convenient to use event property names in pattern and EPL statements that are backed up by a given public method or member variable (field) in a CLR type. And it can be useful to declare multiple event properties that each map to the same method or member variable.

We can configure properties of events via `method-property` and `field-property` elements, as the next example shows.

```
<event-type name="StockTick" class="com.espertech.esper.example.stockticker.event.StockTickEvent">
  <legacy-type accessor-style="native" code-generation="enabled">
    <method-property name="price" accessor-method="getCurrentPrice" />
    <field-property name="volume" accessor-field="volumeField" />
  </legacy-type>
</event-type>
```

The XML configuration snippet above declared an event property named `price` backed by a getter-method named `getCurrentPrice`, and a second event property named `volume` that is backed by a public member variable named `volumeField`. Thus the price and volume properties can be used in a statement:

```
select avg(price * volume) from StockTick
```

As with all configuration options, the API can also be used:

```
Configuration configuration = new Configuration();
ConfigurationEventTypeLegacy legacyDef = new ConfigurationEventTypeLegacy();
legacyDef.AddMethodProperty("price", "GetCurrentPrice");
legacyDef.AddFieldProperty("volume", "_volumeField");
config.AddEventType<StockTickEvent>("StockTick", legacyDef);
```

Turning off Code Generation

Esper employs the `CGLIB` library for very fast read access to event property values. For certain legacy CLR types it may be desirable to disable the use of this library and instead use CLR reflection to obtain event property values from event objects.

In the XML configuration, the optional `code-generation` attribute in the `legacy-type` section can be set to disabled as shown next.

```
<event-type name="MyLegacyEvent" class="com.mycompany.package.MyLegacyEventClass">
  <legacy-type accessor-style="native" code-generation="disabled" />
</event-type>
```



```
</event-type>
```

The sample below shows how to configure this option via the API.

```
Configuration configuration = new Configuration();
ConfigurationEventTypeLegacy legacyDef = new ConfigurationEventTypeLegacy();
legacyDef.CodeGeneration = ConfigurationEventTypeLegacy.CodeGeneration.DISABLED;
config.AddEventType<MyLegacyEventClass>("MyLegacyEvent", legacyDef);
```

Case Sensitivity and Property Names

By default the engine resolves Java event properties case sensitive. That is, property names in statements must match property names in name and case. This option controls case sensitivity per CLR type.

In the configuration XML, the optional `property-resolution-style` attribute in the `legacy-type` element can be set to any of these values:

Table 13.2. Property Resolution Case Sensitivity Styles

Style Name	Description
<code>case_sensitive</code> (default)	As the default setting, the engine matches property names for the exact name and case only.
<code>case_insensitive</code>	Properties are matched if the names are identical. A case insensitive search is used and will choose the first property that matches the name exactly or the first property that matches case insensitively should no match be found.
<code>distinct_case_insensitive</code>	Properties are matched if the names are identical. A case insensitive search is used and will choose the first property that matches the name exactly case insensitively. If more than one 'name' can be mapped to the property an exception is thrown.

The sample below shows this option in XML configuration, however the setting can also be changed via API:

```
<event-type name="MyLegacyEvent" class="com.mycompany.package.MyLegacyEventClass">
  <legacy-type property-resolution-style="case_insensitive"/>
</event-type>
```

Factory and Copy Method

The `insert into` clause and directly instantiate and populate your event object. By default the engine invokes the default constructor to instantiate an event object. To change this behavior, you may configure a factory method. The factory method is a method name or a class name plus a method name (in the format `class.method`) that returns an instance of the class.

The `update` clause can change event properties on an event object. For the purpose of maintaining consistency, the engine may have to copy your event object via serialization (annotated with the `System.Serializable` attribute). If instead you do not want any copy operations to occur, or your application needs to control the copy operation, you may configure a copy method. The copy method is the name of a method on the event object that copies the event object.

The sample below shows this option in XML configuration, however the setting can also be changed via API:

```
<event-type name="MyLegacyEvent" class="com.mycompany.package.MyLegacyEventClass"
  factory-method="com.mycompany.myapp.MySampleEventFactory.createMyLegacyTypeEvent" copy-method="myCop
</event-type>
```

13.4.2. Events represented by `System.Collection.Generic.IDictionary`

The engine can process `System.Collection.Generic.IDictionary` events via the `SendEvent(Map map, String eventName)` method on the `EPRuntime` interface. Entries in the `Map` represent event properties. Keys must be of type `System.String` for the engine to be able to look up event property names in pattern or EPL statements. Values can be of any type. CLR objects as values in a `Map` can be processed by the engine, and strongly-typed nested maps are also supported. Please see the Chapter 2, *Event Representations* section for details on how to use `Map` events with the engine.

Via configuration we provide an event type name for `Map` events for use in statements, and the event property names and types enabling the engine to validate properties in statements.

The below snippet of XML configuration configures an event named `MyMapEvent`.

```
<event-type name="MyMapEvent">
  <clr-map>
    <map-property name="carId" class="int"/>
    <map-property name="carType" class="string"/>
    <map-property name="assembly" class="com.mycompany.Assembly"/>
  </clr-map>
</event-type>
```

This configuration defines the `carId` property of `MyMapEvent` events to be of type `int`, and the `carType` property to be of type `System.String`. The `assembly` property of the `Map` event will contain instances of `com.mycompany.Assembly` for the engine to query.

The valid types for the `class` attribute are listed in Section 13.5, “Type Names”. In addition, any fully-qualified CLR type name that can be resolved via `Class.forName` is allowed.

You can also use the configuration API to configure `Map` event types, as the short code snippet below demonstrates:

```
IDictionary<String, Object> properties = new Dictionary<String, Object>();
properties["carId"] = "int";
properties["carType"] = "string";
properties["assembly"] = typeof(Assembly).FullName;

Configuration configuration = new Configuration();
configuration.AddEventType("MyMapEvent", properties);
```

For strongly-typed nested maps (maps-within-maps), the configuration API method `addEventType` can also be used to define the nested types. The XML configuration does not provide the capability to configure nested maps.

Finally, here is a sample EPL statement that uses the configured `MyMapEvent` map event. This statement uses the `chassisTag` and `numParts` properties of `Assembly` objects in each map.

```
select carType, assembly.chassisTag, count(assembly.numParts) from MyMapEvent.win:time(60 sec)
```

A `Map` event type may also become a subtype of one or more supertypes that must also be `Map` event types. The `clr-map` element provides an optional attribute `supertype-names` that accepts a comma-separated list of names of `Map` event types that are supertypes to the type:

```
<event-type name="AccountUpdate">
  <clr-map supertype-names="BaseUpdate, AccountEvent">
    ...
```

For initialization time configuration, the `AddMapSuperType` method can be used to add Map hierarchy information. For runtime configuration, pass the supertype names to the `AddEventType` method in `ConfigurationOperations`.

13.4.3. Events represented by `System.Xml.XmlNode`

Via this configuration item the Esper engine can natively process `System.Xml.XmlNode` instances, i.e. XML document object model (DOM) nodes. Please see the Chapter 2, *Event Representations* section for details on how to use Node events with the engine.

Esper allows configuring XPath expressions as event properties. You can specify arbitrary XPath functions or expressions and provide a property name by which their result values will be available for use in expressions.

For XML documents that follow a XML schema, Esper can load and interrogate your schema and validate event property names and types against the schema information.

Nested, mapped and indexed event properties are also supported in expressions against `System.Xml.XmlNode` events. Thus XML trees can conveniently be interrogated using the existing event property syntax for querying JavaBean objects, JavaBean object graphs or `System.Collection.Generic.IDictionary` events.

In the simplest form, the Esper engine only requires a configuration entry containing the root element name and the event type name in order to process `System.Xml.XmlNode` events:

```
<event-type name="MyXMLNodeEvent">
  <xml-dom root-element-name="myevent" />
</event-type>
```

You can also use the configuration API to configure XML event types, as the short example below demonstrates. In fact, all configuration options available through XML configuration can also be provided via setter methods on the `ConfigurationEventTypeXMLDOM` class.

```
Configuration configuration = new Configuration();
ConfigurationEventTypeXMLDOM desc = new ConfigurationEventTypeXMLDOM();
desc.RootElementName = "myevent";
desc.AddXPathProperty("name1", "/element/@attribute", XPathResultType.String);
desc.AddXPathProperty("name2", "/element/subelement", XPathResultType.Number);
configuration.addEventType("MyXMLNodeEvent", desc);
```

The next example presents configuration options in a sample configuration entry.

```
<event-type name="AutoIdRFIDEvent">
  <xml-dom root-element-name="Sensor" schema-resource="data/AutoIdPmlCore.xsd"
    default-namespace="urn:autoId:specification:interchange:PMLCore:xml:schema:1">
    <namespace-prefix prefix="pmlcore"
      namespace="urn:autoId:specification:interchange:PMLCore:xml:schema:1"/>
    <xpath-property property-name="countTags"
      xpath="count(/pmlcore:Sensor/pmlcore:Observation/pmlcore:Tag)" type="number"/>
  </xml-dom>
</event-type>
```

This example configures an event property named `countTags` whose value is computed by an XPath expression. The namespace prefixes and default namespace are for use with XPath expressions and must also be made known to the engine in order for the engine to compile XPath expressions. Via the `schema-resource` attribute

we instruct the engine to load a schema file. You may also use `schema-text` instead to provide the actual text of the schema.

Here is an example EPL statement using the configured event type named `AutoIdRFIDEvent`.

```
select ID, countTags from AutoIdRFIDEvent.win:time(30 sec)
```

Schema Resource

The `schema-resource` attribute takes a schema resource URL or classpath-relative filename. The engine attempts to resolve the schema resource as an URL. If the schema resource name is not a valid URL, the engine attempts to resolve the resource from resource manager (`com.espertech.esper.compat.ResourceManager`). Use the `schema-text` attribute instead when it is more practical to provide the actual text of the schema.

By configuring a schema file for the engine to load, the engine performs these additional services:

- Validates the event properties in a statement, ensuring the event property name matches an attribute or element in the XML
- Determines the type of the event property allowing event properties to be used in type-sensitive expressions such as expressions involving arithmetic (Note: XPath properties are also typed)
- Matches event property names to either element names or attributes

If no schema resource is specified, none of the event properties specified in statements are validated at statement creation time and their type defaults to `System.String`. Also, attributes are not supported if no schema resource is specified and must thus be declared via XPath expression.

Explicit XPath Property

The `xpath-property` element adds explicitly-names event properties to the event type that are computed via an XPath expression. In order for the XPath expression to compile, be sure to specify the `default-namespace` attribute and use the `namespace-prefix` to declare namespace prefixes.

XPath expression properties are strongly typed. The `type` attribute allows the following values. These values correspond to those declared by `System.Xml.XPath.XPathResultType`.

- number (Note: resolves to a double)
- string
- boolean
- any
- nodeset

In case you need your XPath expression to return a type other than the types listed above, an optional cast-to type can be specified. If specified, the operation first obtains the result of the XPath expression as the defined type (number, string, boolean) and then casts or parses the returned type to the specified cast-to-type. At runtime, a warning message is logged if the XPath expression returns a result object that cannot be casted or parsed.

The next line shows how to return a long-type property for an XPath expression that returns a string:

```
desc.AddXPathProperty("name", "/element/sub", XPathConstants.STRING, "long");
```

The equivalent configuration XML is:

```
<xpath-property property-name="name" xpath="/element/sub" type="string" cast="long"/>
```

See Section 13.5, “Type Names” for a list of cast-to type names.

Absolute or Deep Property Resolution

This setting indicates that when properties are compiled to XPath expressions that the compilation should generate an absolute XPath expression or a deep (find element) XPath expression.

For example, consider the following statement against an event type that is represented by a XML DOM document, assuming the event type GetQuote has been configured with the engine as a XML DOM event type:

```
select request, request.symbol from GetQuote
```

By default, the engine compiles the "request" property name to an XPath expression `"/GetQuote/request"`. It compiles the nested property named "request.symbol" to an XPath expression `"/GetQuote/request/symbol"`, wherein the root element node is "GetQuote".

By setting absolute property resolution to false, the engine compiles the "request" property name to an XPath expression `"//request"`. It compiles the nested property named "request.symbol" to an XPath expression `"//request/symbol"`. This enables these elements to be located anywhere in the XML document.

The setting is available in XML via the attribute `resolve-properties-absolute`.

The configuration API provides the above settings as shown here in a sample code:

```
ConfigurationEventTypeXMLDOM desc = new ConfigurationEventTypeXMLDOM();
desc.RootElementName = "GetQuote";
desc.DefaultNamespace = "http://services.samples/xsd";
desc.RootElementNamespace = "http://services.samples/xsd";
desc.AddNamespacePrefix("m0", "http://services.samples/xsd");
desc.IsXPathResolvePropertiesAbsolute = false;
configuration.AddEventType("GetQuote", desc);
```

XPath Variable and Function Resolver

If your XPath expressions require variables or functions, your application may provide the class name of an `XPathVariableResolver` and `XPathFunctionResolver`. At type initialization time the engine instantiates the resolver instances and provides these to the `XPathFactory`.

This example shows the API to set this configuration.

```
ConfigurationEventTypeXMLDOM desc = new ConfigurationEventTypeXMLDOM();
desc.XPathFunctionResolver = typeof(MyXPathFunctionResolver).FullName;
desc.XPathVariableResolver = typeof(MyXPathVariableResolver).FullName;
```

Auto Fragment

This option is for use when a XSD schema is provided and determines whether the engine automatically creates an event type when a property expression transposes a property that is a complex type according to the schema.

An example:

```
ConfigurationEventTypeXMLDOM desc = new ConfigurationEventTypeXMLDOM();
desc.IsAutoFragment = false;
```

XPath Property Expression

By default Esper employs the built-in DOM walker implementation to evaluate XPath expressions, which is not namespace-aware.

This configuration setting, when set to true, instructs the engine to rewrite property expressions into XPath.

An example:

```
ConfigurationEventTypeXMLDOM desc = new ConfigurationEventTypeXMLDOM();
desc.IsXPathPropertyExpr = true;
```

Event Sender Setting

By default an `EventSender` for a given XML event type validates the root element name for which the type has been declared against the one provided by the `System.Xml.XmlNode` sent into the engine.

This configuration setting, when set to false, instructs an `EventSender` to not validate.

An example:

```
ConfigurationEventTypeXMLDOM desc = new ConfigurationEventTypeXMLDOM();
desc.IsEventSenderValidatesRoot = false;
```

13.4.4. Events represented by Plug-in Event Representations

As part of the extension API plug-in event representations allows an application to create new event types and event instances based on information available elsewhere. Please see Section 15.8, “Event Type And Event Object” for details.

The configuration examples shown next use the configuration API to select settings. All options are also configurable via XML, please refer to the sample configuration XML in file `esper.sample.cfg.xml`.

Enabling an Custom Event Representation

Use the method `AddPlugInEventRepresentation` to enable a custom event representation, like this:

```
URI rootURI = new URI("type://mycompany/myproject/myname");
config.AddPlugInEventRepresentation(rootURI,
    typeof(MyEventRepresentation).FullName, null);
```

The `type://` part of the URI is an optional convention for the scheme part of an URI.

If your event representation takes initialization parameters, these are passed in as the last parameter. Initialization parameters can also be stored in the configuration XML, in which case they are passed as an XML string to the plug-in class.

Adding Plug-in Event Types

To register event types that your plug-in event representation creates in advance of creating an EPL statement (either at runtime or at configuration time), use the method `AddPlugInEventType`:

```
Uri childURI = new Uri("type://mycompany/myproject/myname/MyEvent");
configuration.AddPlugInEventType("MyEvent", new Uri[] {childURI}, null);
```

Your plug-in event type may take initialization parameters, these are passed in as the last parameter. Initializa-

tion parameters can also be stored in the configuration XML.

Setting Resolution URIs

The engine can invoke your plug-in event representation when an EPL statement is created with an event type name that the engine has not seen before. Plug-in event representations can resolve such names to an actual event type. In order to do this, you need to supply a list of resolution URIs. Use the `PlugInEventTypeResolutionURIs` property, at runtime or at configuration time:

```
Uri childURI = new Uri("type://mycompany/myproject/myname");
configuration.PlugInEventTypeResolutionURIs = new[] {childURI};
```

13.4.5. Class and package imports

Esper allows invocations of static library functions in expressions, as outlined in Section 8.1, “Single-row Function Reference”. This configuration item can be set to allow a partial rather than a fully qualified class name in such invocations. The imports work in the same way as in Java files, so both packages and classes can be imported.

```
select Math.Max(priceOne, PriceTwo)
// via configuration equivalent to
select System.Math.Max(priceOne, priceTwo)
```

Esper auto-imports the following library packages if no other configuration is supplied. This list is replaced with any configuration specified in a configuration file or through the API.

- System
- System.Collections
- System.Text

In a XML configuration file the auto-import configuration may look as below:

```
<auto-import import-name="com.mycompany.mypackage"/>
<auto-import import-name="com.mycompany.myapp.MyUtilityClass"/>
```

Here is an example of providing imports via the API:

```
Configuration config = new Configuration();
config.AddImport("com.mycompany.mypackage"); // namespace import
config.AddImport("com.mycompany.mypackage.MyLib"); // type import
```

13.4.6. Cache Settings for From-Clause Method Invocations

Method invocations are allowed in the `from` clause in EPL, such that your application may join event streams to the data returned by a web service, or to data read from a distributed cache or object-oriented database, or obtain data by other means. A local cache may be placed in front of such method invocations through the configuration settings described herein.

The LRU cache is described in detail in Section 13.4.8.6.1, “LRU Cache”. The expiry-time cache documentation can be found in Section 13.4.8.6.2, “Expiry-time Cache”

The next XML snippet is a sample cache configuration that applies to methods provided by the classes 'MyFromClauseLookupLib' and 'MyFromClauseWebServiceLib'. The XML and API configuration understand both the fully-qualified CLR type name, as well as the simple class name:

```
<method-reference class-name="com.mycompany.MyFromClauseLookupLib">
  <expiry-time-cache max-age-seconds="10" purge-interval-seconds="10" ref-type="weak" />
</method-reference>
<method-reference class-name="MyFromClauseWebServiceLib">
  <lru-cache size="1000" />
</method-reference>
```

13.4.7. Variables

Variables can be created dynamically in EPL via the `create variable` syntax but can also be configured at runtime and at configuration time.

A variable is declared by specifying a variable name, the variable type and an optional initialization value. The initialization value can be of the same or compatible type as the variable type, or can also be a String value that, when parsed, is compatible to the type declared for the variable.

In a XML configuration file the variable configuration may look as below. The Configuration API can also be used to configure variables.

```
<variable name="var_threshold" type="long" initialization-value="100" />
<variable name="var_key" type="string" />
```

Please find the list of valid values for the `type` attribute in Section 13.5, “Type Names”.

13.4.8. Relational Database Access

Esper has the capability to join event streams against historical data sources, such as a relational database. This section describes the configuration entries that the engine requires to access data stored in your database. Please see Section 4.13, “Accessing Relational Data via SQL” for information on the use of EPL queries that include historical data sources.

EPL queries that poll data from a relational database specify the name of the database as part of the EPL statement. The engine uses the configuration information described here to resolve the database name in the statement to database settings. The required and optional database settings are summarized below.

- Database connections can be obtained via JDBC `javax.xml.DataSource`, via `java.sql.DriverManager` and via data source factory. Either one of these methods to obtain database connections is a required configuration.
- Optionally, JDBC connection-level settings such as auto-commit, transaction isolation level, read-only and the catalog name can be defined.
- Optionally, a connection lifecycle can be set to indicate to the engine whether the engine must retain connections or must obtain a new connection for each lookup and close the connection when the lookup is done (pooled).
- Optionally, define a cache policy to allow the engine to retrieve data from a query cache, reducing the number of query executions.

Some of the settings can have important performance implications that need to be carefully considered in relationship to your database software, JDBC driver and runtime environment. This section attempts to outline such implications where appropriate.

The sample XML configuration file in the "etc" folder can be used as a template for configuring database settings. All settings are also available by means of the configuration API through the classes `Configuration` and `ConfigurationDBRef`.

Connections obtained via DataSource

This configuration causes Esper to obtain a database connection from a `javax.sql.DataSource` available from your JNDI provider.

The setting is most useful when running within an application server or when a JNDI directory is otherwise present in your Java VM. If your application environment does not provide an available `DataSource`, the next section outlines how to use Apache DBCP as a `DataSource` implementation with connection pooling options and outlines how to use a custom factory for `DataSource` implementations.

If your `DataSource` provides connections out of a connection pool, your configuration should set the collection lifecycle setting to `pooled`.

The snippet of XML below configures a database named `mydb1` to obtain connections via a `javax.sql.DataSource`. The `datasource-connection` element instructs the engine to obtain new connections to the database `mydb1` by performing a lookup via `javax.naming.InitialContext` for the given object lookup name. Optional environment properties for the `InitialContext` are also shown in the example.

```
<database-reference name="mydb1">
  <datasource-connection context-lookup-name="java:comp/env/jdbc/mydb">
    <env-property name="java.naming.factory.initial" value="com.myclass.CtxFactory"/>
    <env-property name="java.naming.provider.url" value="iiop://localhost:1050"/>
  </datasource-connection>
</database-reference>
```

To help you better understand how the engine uses this information to obtain connections, we have included the logic below.

```
if (envProperties.size() > 0) {
    initialContext = new InitialContext(envProperties);
}
else {
    initialContext = new InitialContext();
}
DataSource dataSource = (DataSource) initialContext.lookup(lookupName);
Connection connection = dataSource.getConnection();
```

In order to plug-in your own implementation of the `DataSource` interface, your application may use an existing JNDI provider as provided by an application server if running in a J2EE environment.

In case your application does not have an existing JNDI implementation to register a `DataSource` to provide connections, you may set the `java.naming.factory.initial` property in the configuration to point to your application's own implementation of the `javax.naming.spi.InitialContextFactory` interface that can return your application `DataSource` through the `javax.naming.Context` provided by the factory implementation. Please see Java Naming and Directory Interface (JNDI) API documentation for further information.

Connections obtained via DataSource Factory

This section describes how to use *Apache Commons Database Connection Pooling (Apache DBCP)* [<http://commons.apache.org/dbcp>] with Esper. We also explain how to provide a custom application-specific `DataSource` factory if not using Apache DBCP.

If your `DataSource` provides connections out of a connection pool, your configuration should set the collection lifecycle setting to `pooled`.

Apache DBCP provides comprehensive means to test for dead connections or grow and shrink a connection

pool. Configuration properties for Apache DBCP can be found at *Apache DBCP configuration* [<http://commons.apache.org/dbcp/configuration.html>]. The listed properties are passed to Apache DBCP via the properties list provided as part of the Esper configuration.

The snippet of XML below is an example that configures a database named `mydb3` to obtain connections via the pooling `DataSource` provided by Apache DBCP `BasicDataSourceFactory`.

The listed properties are passed to DBCP to instruct DBCP how to manage the connection pool. The settings below initialize the connection pool to 2 connections and provide the validation query `select 1 from dual` for DBCP to validate a connection before providing a connection from the pool to Esper:

```
<database-reference name="mydb3">
  <!-- For a complete list of properties see Apache DBCP. -->
  <datasourcefactory-connection class-name="org.apache.commons.dbcp.BasicDataSourceFactory">
    <env-property name="username" value="myusername"/>
    <env-property name="password" value="mypassword"/>
    <env-property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <env-property name="url" value="jdbc:mysql://localhost/test"/>
    <env-property name="initialSize" value="2"/>
    <env-property name="validationQuery" value="select 1 from dual"/>
  </datasourcefactory-connection>
  <connection-lifecycle value="pooled"/>
</database-reference>
```

The same configuration options provided through the API:

```
Properties props = new Properties();
props.put("username", "myusername");
props.put("password", "mypassword");
props.put("driverClassName", "com.mysql.jdbc.Driver");
props.put("url", "jdbc:mysql://localhost/test");
props.put("initialSize", 2);
props.put("validationQuery", "select 1 from dual");

ConfigurationDBRef configDB = new ConfigurationDBRef();
// BasicDataSourceFactory is an Apache DBCP import
configDB.setDataSourceFactory(props, BasicDataSourceFactory.class.getName());
configDB.setConnectionLifecycleEnum(ConfigurationDBRef.ConnectionLifecycleEnum.POOLED);

Configuration configuration = new Configuration();
configuration.addDatabaseReference("mydb3", configDB);
```

Apache Commons DBCP is a separate download and not provided as part of the Esper distribution. The Apache Commons DBCP jar file requires the Apache Commons Pool jar file.

Your application can provide its own factory implementation for `DataSource` instances: Set the class name to the name of the application class that provides a public static method named `createDataSource` which takes a single `Properties` object as parameter and returns a `DataSource` implementation. For example:

```
configDB.setDataSourceFactory(props, MyOwnDataSourceFactory.class.getName());
...
class MyOwnDataSourceFactory {
  public static DataSource CreateDataSource(Properties properties) {
    return new MyDataSourceImpl(properties);
  }
}
```

Connections obtained via DriverManager

The next snippet of XML configures a database named `mydb2` to obtain connections via `java.sql.DriverManager`. The `drivermanager-connection` element instructs the engine to obtain new con-

nections to the database mydb2 by means of `Class.forName` and `DriverManager.getConnection` using the class name, URL and optional username, password and connection arguments.

```
<database-reference name="mydb2">
  <drivermanager-connection class-name="my.sql.Driver"
    url="jdbc:mysql://localhost/test?user=root&password=mypassword"
    user="myuser" password="mypassword">
    <connection-arg name="user" value="myuser"/>
    <connection-arg name="password" value="mypassword"/>
    <connection-arg name="somearg" value="someargvalue"/>
  </drivermanager-connection>
</database-reference>
```

The username and password are shown in multiple places in the XML only as an example. Please check with your database software on the required information in URL and connection arguments.

Connections-level settings

Additional connection-level settings can optionally be provided to the engine which the engine will apply to new connections. When the engine obtains a new connection, it applies only those settings to the connection that are explicitly configured. The engine leaves all other connection settings at default values.

The below XML is a sample of all available configuration settings. Please refer to the Java API JavaDocs for `System.Data.DbConnection` for more information to each option or check the documentation of your driver and database software.

```
<database-reference name="mydb2">
... configure data source or driver manager settings...
  <connection-settings auto-commit="true" catalog="mycatalog"
    read-only="true" transaction-isolation="1" />
</database-reference>
```

The `read-only` setting can be used to indicate to your database engine that SQL statements are read-only. The `transaction-isolation` and `auto-commit` help you database software perform the right level of locking and lock release. Consider setting these values to reduce transactional overhead in your database queries.

Connections lifecycle settings

By default the engine retains a separate database connection for each started EPL statement. However, it is possible to override this behavior and require the engine to obtain a new database connection for each lookup, and to close that database connection after the lookup is completed. This often makes sense when you have a large number of EPL statements and require pooling of connections via a connection pool.

In the `pooled` setting, the engine obtains a database connection from the data source or driver manager for every query, and closes the connection when done, returning the database connection to the pool if using a pooling data source.

In the `retain` setting, the engine retains a separate dedicated database connection for each statement and does not close the connection between uses.

The XML for this option is below. The connection lifecycle allows the following values: `pooled` and `retain`.

```
<database-reference name="mydb2">
... configure data source or driver manager settings...
  <connection-lifecycle value="pooled"/>
</database-reference>
```

Cache settings

Cache settings can dramatically reduce the number of database queries that the engine executes for EPL statements. If no cache setting is specified, the engine does not cache query results and executes a separate database query for every event.

Caches store the results of database queries and make these results available to subsequent queries using the exact same query parameters as the query for which the result was stored. If your query returns one or more rows, the cache keep the result rows of the query keyed to the parameters of the query. If your query returns no rows, the cache also keeps the empty result. Query results are held by a cache until the cache entry is evicted. The strategies available for evicting cached query results are listed next.

LRU Cache

The least-recently-used (LRU) cache is configured by a maximum size. The cache discards the least recently used query results first once the cache reaches the maximum size.

The XML configuration entry for a LRU cache is as below. This entry configures an LRU cache holding up to 1000 query results.

```
<database-reference name="mydb">
... configure data source or driver manager settings...
  <lru-cache size="1000" />
</database-reference>
```

Expiry-time Cache

The expiry time cache is configured by a maximum age in seconds, a purge interval and an optional reference type. The cache discards (on the get operation) any query results that are older then the maximum age so that stale data is not used. If the cache is not empty, then every purge interval number of seconds the engine purges any expired entries from the cache.

The XML configuration entry for an expiry-time cache is as follows. The example configures an expiry time cache in which prior query results are valid for 60 seconds and which the engine inspects every 2 minutes to remove query results older then 60 seconds.

```
<database-reference name="mydb">
... configure data source or driver manager settings...
  <expiry-time-cache max-age-seconds="60" purge-interval-seconds="120" />
</database-reference>
```

By default, the expiry-time cache is backed by a `java.util.WeakHashMap` and thus relies on weak references. That means that cached SQL results can be freed during garbage collection.

Via XML or using the configuration API the type of reference can be configured to not allow entries to be garbage collected, by setting the `ref-type` property to `hard`:

```
<database-reference name="mydb">
... configure data source or driver manager settings...
  <expiry-time-cache max-age-seconds="60" purge-interval-seconds="120" ref-type="hard" />
</database-reference>
```

The last setting for the cache reference type is `soft`: This strategy allows the garbage collection of cache entries only when all other weak references have been collected.

Column Change Case

This setting instructs the engine to convert to lower- or uppercase any output column names returned by your

database system. When using Oracle relational database software, for example, column names can be changed to lowercase via this setting.

A sample XML configuration entry for this setting is:

```
<column-change-case value="lowercase"/>
```

SQL Types Mapping

By providing a mapping of SQL types (`java.sql.Types`) to Java built-in types your code can avoid using sometimes awkward default database types and can easily change the way Esper returns Java types for columns returned by a SQL query.

The mapping maps a constant as defined by `java.sql.Types` to a Java built-in type of any of the following Java type names: `String`, `BigDecimal`, `Boolean`, `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, `ByteArray`, `SqlDate`, `SqlTime`, `SqlTimestamp`. The Java type names are not case-sensitive.

A sample XML configuration entry for this setting is shown next. The sample maps `Types.NUMERIC` which is a constant value of 2 per JDBC API to the Java `int` type.

```
<sql-types-mapping sql-type="2" java-type="int" />
```

Metadata Origin

This setting controls how the engine retrieves SQL statement metadata from JDBC prepared statements.

Table 13.3. Syntax and results of aggregate functions

Option	Description
default	By default, the engine detects the driver name and queries prepared statement metadata if the driver is not an Oracle database driver. For Oracle drivers, the engine uses lexical analysis of the SQL statement to construct a sample SQL statement and then fires that statement to retrieve statement metadata.
metadata	The engine always queries prepared statement metadata regardless of the database driver used.
sample	The engine always uses lexical analysis of the SQL statement to construct a sample SQL statement, and then fires that statement to retrieve statement metadata.

13.4.9. Engine Settings related to Concurrency and Threading

Preserving the order of events delivered to listeners

In multithreaded environments, this setting controls whether dispatches of statement result events to listeners preserve the ordering in which a statement processes events. By default the engine guarantees that it delivers a statement's result events to statement listeners in the order in which the result is generated. This behavior can be turned off via configuration as below.

The next code snippet shows how to control this feature:

```
Configuration config = new Configuration();
config.EngineDefaults.Threading.IsListenerDispatchPreserveOrder = false;
engine = EPServiceProviderManager.GetDefaultProvider(config);
```

And the XML configuration file can also control this feature by adding the following elements:

```
<engine-settings>
  <defaults>
    <threading>
      <listener-dispatch preserve-order="true" timeout-msec="1000" locking="spin"/>
    </threading>
  </defaults>
</engine-settings>
```

As discussed, by default the engine can temporarily block another processing thread when delivering result events to listeners in order to preserve the order in which results are delivered to a given statement. The maximum time the engine blocks a thread can also be configured, and by default is set to 1 second.

As such delivery locks are typically held for a very short amount of time, the default blocking technique employs a spin lock (There are two techniques for implementing blocking; having the operating system suspend the thread until it is awakened later or using spin locks). While spin locks are CPU-intensive and appear inefficient, a spin lock can be more efficient than suspending the thread and subsequently waking it up, especially if the lock in question is held for a very short time. That is because there is significant overhead to suspending and rescheduling a thread.

The locking technique can be changed to use a blocking strategy that suspends the thread, by means of setting the locking property to 'suspend'.

Preserving the order of events for insert-into streams

In multithreaded environments, this setting controls whether statements producing events for other statements via insert-into preserve the order of delivery within the producing and consuming statements, allowing statements that consume other statement's events to behave deterministic in multithreaded applications, if the consuming statement requires such determinism. By default, the engine makes this guarantee (the setting is on).

Take, for example, an application where a single statement (S1) inserts events into a stream that another statement (S2) further evaluates. A multithreaded application may have multiple threads processing events into statement S1. As statement S1 produces events for consumption by statement S2, such results may need to be delivered in the exact order produced as the consuming statement may rely on the order received. For example, if the first statement counts the number of events, the second statement may employ a pattern that inspects counts and thus expect the counts posted by statement S1 to continuously increase by 1 even though multiple threads process events.

The engine may need to block a thread such that order of delivery is maintained, and statements that require order (such as pattern detection, previous and prior functions) receive a deterministic order of events. The settings available control the blocking technique and parameters. As described in the section immediately prior, the default blocking technique employs spin locks per statement inserting events for consumption, as the locks in questions are typically held a very short time. The 'suspend' blocking technique can be configured and a timeout value can also be defined.

The XML configuration file may change settings via the following elements:

```
<engine-settings>
  <defaults>
```

```

    <threading>
      <insert-into-dispatch preserve-order="true" timeout-msec="100" locking="spin"/>
    </threading>
  </defaults>
</engine-settings>

```

Internal Timer Settings

This option can be used to disable the internal timer thread and such have the application supply external time events, as well as to set a timer resolution.

The next code snippet shows how to disable the internal timer thread via the configuration API:

```

Configuration config = new Configuration();
config.EngineDefaults.Threading.UsInternalTimerEnabled = false;

```

This snippet of XML configuration leaves the internal timer enabled (the default) and sets a resolution of 200 milliseconds (the default is 100 milliseconds):

```

<engine-settings>
  <defaults>
    <threading>
      <internal-timer enabled="true" msec-resolution="200"/>
    </threading>
  </defaults>
</engine-settings>

```

We recommend that when disabling the internal timer, applications send an external timer event setting the start time before creating statements, such that statement start time is well-defined.

Advanced Threading Options

The settings described herein are for enabling advanced threading options for inbound, outbound, timer and route executions.

Take the next snippet of XML configuration as an example. It configures all threading options to 2 threads, which may not be suitable to your application, however demonstrates the configuration:

```

<engine-settings>
  <defaults>
    <threading>
      <threadpool-inbound enabled="true" num-threads="2"/>
      <threadpool-outbound enabled="true" num-threads="2" capacity="1000"/>
      <threadpool-timerexec enabled="true" num-threads="2"/>
      <threadpool-routeexec enabled="true" num-threads="2"/>
    </threading>
  </defaults>
</engine-settings>

```

By default, queues are unbound and backed by `java.util.concurrent.LinkedBlockingQueue`. The optional `capacity` attribute can be set to instruct the threading option to configure a capacity-bound queue with a sender-wait (blocking put) policy, backed `ArrayBlockingQueue`.

This example uses the API for configuring inbound threading :

```

Configuration config = new Configuration();
config.EngineDefaults.Threading.IsThreadPoolInbound = true;
config.EngineDefaults.Threading.ThreadPoolInboundNumThreads = 2;

```

With a bounded work queue, the queue size and pool size should be tuned together. A large queue coupled with a small pool can help reduce memory usage, CPU usage, and context switching, at the cost of potentially constraining throughput.

13.4.10. Engine Settings related to Event Metadata

Type Property Names, Case Sensitivity and Accessor Style

The engine-wide settings discussed here are used when you want to control case sensitivity or accessor style for all event classes as a default. The two settings are found under `class-property-resolution` under `event-meta` in the XML configuration.

To control the case sensitivity as discussed in Section 13.4.1.6, “Case Sensitivity and Property Names”, add the `style` attribute in the XML configuration to set a default case sensitivity applicable to all event classes unless specifically overridden by class-specific configuration. The default case sensitivity is `case_sensitive` (case sensitivity turned on).

To control the accessor style as discussed in Section 13.4.1.3, “Legacy Event Classes”, add the `accessor-style` attribute in the XML configuration to set a default accessor style applicable to all event classes unless specifically overridden by class-specific configuration. The default accessor style is `native` accessor style.

The next code snippet shows how to control this feature via the API:

```
Configuration config = new Configuration();
config.EngineDefaults.EventMeta.ClassPropertyResolutionStyle = PropertyResolutionStyle.CASE_INSENSITIVE;
config.EngineDefaults.EventMeta.DefaultAccessorStyle = AccessorStyleEnum.PUBLIC;
```

13.4.11. Engine Settings related to View Resources

Sharing View Resources between Statements

The engine by default attempts to optimize resource usage and thus re-uses or shares views between statements that declare same views. However, in multi-threaded environments, this can lead to reduced concurrency as locking for shared view resources must take place. Via this setting this behavior can be turned off for higher concurrency in multi-threaded processing.

The next code snippet outlines the API to turn off view resource sharing between statements:

```
Configuration config = new Configuration();
config.EngineDefaults.ViewResources.ShareViews = false;
```

Configuring Multi-Expiry Policy Defaults

By default, when combining multiple data window views, Esper applies an intersection of the data windows unless the `retain-union` keyword is provided which instructs to apply an union. The setting described herein may be used primarily for backward compatibility to instruct that intersection should not be the default.

Here is a sample statement that specifies multiple expiry policies:

```
select * from MyEvent.std:unique(price).std:unique(quantity)
```

By default Esper applies intersection as described in Section 4.4.4, “Multiple Data Window Views”.

Here is the setting to allow multiple data windows without the intersection default:

```
Configuration config = new Configuration();
config.EngineDefaults.ViewResources.AllowMultipleExpiryPolicies = true;
```

When setting this option to true, and when using multiple data window views for a given stream, the behavior is as follows: The top-most data window receives an insert stream of events. It passes each insert stream event to each further data window view in the chain. Each data window view may remove events according to its expiry policy. Such remove stream events are only passed to data window views further in the chain, and are not made available to data window views earlier in the chain.

It is recommended to leave the default setting at false.

13.4.12. Engine Settings related to Logging

Execution Path Debug Logging

By default, the engine does not produce debug output for the event processing execution paths even when Log4j or Logger configurations have been set to output debug level logs. To enable debug level logging, set this option in the configuration as well as in your Log4j configuration file.

When debug-level logging is enabled by setting the flag as below and by setting DEBUG in the Log4j configuration file, then the timer processing may produce extensive debug output that you may not want to have in the log file. The `timer-debug` setting in the XML or via API as below disables timer debug output which is enabled by default.

The API to use to enable debug logging and disable timer event output is shown here:

```
Configuration config = new Configuration();
config.EngineDefaults.Logging.IsEnableExecutionDebug = true;
config.EngineDefaults.Logging.IsEnableTimerDebug = false;
```

Note: this is a configuration option that applies to all engine instances of a given Java module or VM.

The XML snippet is:

```
<esper-configuration>
  <engine-settings>
    <defaults>
      <logging>
        <execution-path enabled="true"/>
        <timer-debug enabled="false"/>
      </logging>
    </defaults>
  </engine-settings>
</esper-configuration>
```

Query Plan Logging

By default, the engine does not produce query plan output unless logging at debug-level. To enable query plan logging, set this option in the configuration. When enabled, the engine reports, at INFO level, any query plans under the log name `com.espertech.esper.queryplan`.

The API to use to enable query plan logging is shown here:

```
Configuration config = new Configuration();
config.getEngineDefaults().getLogging().setEnableQueryPlan(true);
```

The XML snippet is:

```
<esper-configuration>
  <engine-settings>
    <defaults>
      <logging>
        <query-plan enabled="true"/>
      </logging>
    </defaults>
  </engine-settings>
</esper-configuration>
```

JDBC Logging

By default, the engine does not measure JDBC query execution times or report the number of rows returned from a JDBC query through logging. To enable JDBC logging, set this option in the configuration. When enabled, the engine reports, at INFO level, any JDBC query performance and number of rows returned under the log name `com.espertech.esper.jdbc`.

The API to use to enable query plan logging is shown here:

```
Configuration config = new Configuration();
config.getEngineDefaults().getLogging().setEnableJDBC(true);
```

The XML snippet is:

```
<esper-configuration>
  <engine-settings>
    <defaults>
      <logging>
        <jdbc enabled="true"/>
      </logging>
    </defaults>
  </engine-settings>
</esper-configuration>
```

13.4.13. Engine Settings related to Variables

Variable Version Release Interval

This setting controls the length of time that the engine retains variable versions for use by statements that use variables and that execute, within the same statement for the same event, longer then the time interval. By default, the engine retains 15 seconds of variable versions.

For statements that use variables and that execute (in response to a single timer or other event) longer then the time period, the engine returns the current variable version at the time the statement executes, thereby softening the guarantee of consistency of variable values within the long-running statement. Please see Section 4.18.3, “Using Variables” for more information.

The XML configuration for this setting is shown below:

```
<engine-settings>
  <defaults>
    <variables>
      <msec-version-release value="15000"/>
    </variables>
  </defaults>
</engine-settings>
```

13.4.14. Engine Settings related to Stream Selection

Default Statement Stream Selection

Statements can produce both insert stream (new data) and remove stream (old data) results. Remember that insert stream refers to arriving events and new aggregation values, while remove stream refers to events leaving data windows and prior aggregation values. By default, the engine delivers only the insert stream to listeners and observers of a statement.

There are keywords in the `select` clause that instruct the engine to not generate insert stream and/or remove stream results if your application does not need either one of the streams. These keywords are the `istream`, `rstream` and the `irstream` keywords.

By default, the engine only generates insert stream results equivalent to using the optional `istream` keyword in the `select` clause. If your application requires insert and remove stream results for many statements, your application can add the `irstream` keyword to the `select` clause of each statement, or you can set a new default stream selector via this setting.

The XML configuration for this setting is shown below:

```
<engine-settings>
  <defaults>
    <stream-selection>
      <stream-selector value="irstream" />
    </stream-selection>
  </defaults>
</engine-settings>
```

The equivalent code snippet using the configuration API is here:

```
Configuration config = new Configuration();
config.EngineDefaults.StreamSelection.DefaultStreamSelector = StreamSelector.RSTREAM_ISTREAM_BOTH;
```

13.4.15. Engine Settings related to Time Source

Default Time Source

This setting only applies if internal timer events control engine time (default). If external timer events provide engine clocking, the setting does not apply.

By default, the internal timer uses the call `DateTimeHelper.CurrentTimeMillis` to determine engine time in milliseconds. Via this setting the internal timer can be instructed to use `HighResolutionTimeProvider` instead. Please see Section 12.8, “Time Resolution” for more information.

Note: This is a global setting. If running multiple engine instances in an `AppDomain`, the timer setting is global and applies to all engine instances in the same `AppDomain`, for performance reasons.

A sample XML configuration for this setting is shown below, whereas the sample setting sets the time source to the nanosecond time provider:

```
<engine-settings>
  <defaults>
    <time-source>
      <time-source-type value="nano" />
    </time-source>
  </defaults>
</engine-settings>
```

```
</engine-settings>
```

The equivalent code snippet using the configuration API is here:

```
Configuration config = new Configuration();
config.EngineDefaults.TimeSource.
    TimeSourceType = ConfigurationEngineDefaults.TimeSourceType.NANO;
```

13.4.16. Engine Settings related to Metrics Reporting

This section explains how to enable and configure metrics reporting, which is by default disabled. Please see Section 12.14, “Engine and Statement Metrics Reporting” for more information on the metrics data reported to your application.

The flag that enables metrics reporting is global to a Java virtual machine. If metrics reporting is enabled, the overhead incurred for reporting metrics is carried by all engine instances per Java VM.

Metrics reporting occurs by an engine-controlled separate daemon thread that each engine instance starts at engine initialization time, if metrics reporting and threading is enabled (threading enabled is the default).

Engine and statement metric intervals are in milliseconds. A negative or zero millisecond interval value may be provided to disable reporting.

To control statement metric reporting for individual statements or groups of statements, the engine provides a facility that groups statements by statement name. Each such statement group may have different reporting intervals configured, and intervals can be changed at runtime through runtime configuration. A statement group is assigned a group name at configuration time to identify the group.

Metrics reporting configuration is part of the engine default settings. All configuration options are also available via the `Configuration` API.

A sample XML configuration is shown below:

```
<engine-settings>
  <defaults>
    <metrics-reporting enabled="true" engine-interval="1000" statement-interval="1000"
      threading="true"/>
  </defaults>
</engine-settings>
```

The `engine-interval` setting (defaults to 10 seconds) determines the frequency in milliseconds at which the engine reports engine metrics, in this example every 1 second. The `statement-interval` is for statement metrics. The `threading` flag is true by default since reporting takes place by a dedicated engine thread and can be set to false to use the external or internal timer thread instead.

The next example XML declares a statement group: The statements that have statement names that fall within the group follow a different reporting frequency:

```
<metrics-reporting enabled="true" statement-interval="0">
  <stmtgroup name="MyStmtGroup" interval="2000" default-include="true" num-stmts="100"
    report-inactive="true">
    <exclude-regex>.*test.*</exclude-regex>
  </stmtgroup>
</metrics-reporting>
```

The above example configuration sets the `statement-interval` to zero to disable reporting for all statements.

It defines a statement group by name `MyStmtGroup` and specifies a 2-second interval. The example sets the `default-include` flag to true (by default false) to include all statements in the statement group. The example also sets `report-inactive` to true (by default false) to report inactive statements.

The `exclude-regex` element may be used to specify a regular expression that serves to exclude statements from the group. Any statement whose statement name matches the exclude regular expression is not included in the group. In the above example, all statements with the characters 'test' inside their statement name are excluded from the group.

Any statement not belonging to any of the statement groups follow the configured statement interval.

There are additional elements available to include and exclude statements: `include-regex`, `include-like` and `exclude-like`. The latter two apply SQL-like matching. All patterns are case-sensitive.

Here is a further example of a possible statement group definition, which includes statements whose statement name have the characters `@REPORT` or `@STREAM`, and excludes statements whose statement name have the characters `@IGNORE` or `@METRICS` inside.

```
<metrics-reporting enabled="true">
  <stmtgroup name="MyStmtGroup" interval="1000">
    <include-like>%@REPORT%</include-like>
    <include-regex>.*@STREAM.*</include-like>
    <exclude-like>%@IGNORE%</exclude-like>
    <exclude-regex>.*@METRICS.*</exclude-regex>
  </stmtgroup>
</metrics-reporting>
```

13.4.17. Engine Settings related to Language and Locale

Locale-dependence in Esper can be present in the sort order of string values by the `order by` clause and by the sort view.

By default, Esper sorts string values using the `Compare` method that is not locale dependent. To enable local dependent sorting you must set the configuration flag as described below.

The XML configuration sets the locale dependent sorting as shown below:

```
<engine-settings>
  <defaults>
    <language sort-using-collator="true"/>
  </defaults>
</engine-settings>
```

The API to change the setting:

```
Configuration config = new Configuration();
config.EngineDefaults.Language.SortUsingCollator = true;
```

13.4.18. Engine Settings related to Expression Evaluation

Integer Division and Division by Zero

By default Esper returns double-typed values for divisions regardless of operand types. Division by zero returns positive or negative double infinity.

To have Esper use CLR-standard integer division instead, use this setting as described here. In CLR integer di-

vision, when dividing integer types, the result is an integer type. This means that if you divide an integer unevenly by another integer, it returns the whole number part of the result, does not perform any rounding and the fraction part is dropped. If CLR-standard integer division is enabled, when dividing an integer numerator by an integer denominator, the result is an integer number. Thus the expression `1 / 4` results in an integer zero. Your EPL must then convert at least one of the numbers to a double value before the division, for example by specifying `1.0 / 4` or by using `cast(myint, double)`.

When using CLR Java integer division, division by zero for integer-typed operands always returns null. However division by zero for double-type operands still returns positive or negative double infinity. To also return null upon division by zero for double-type operands, set the flag to true as below (default is false).

The XML configuration is as follows:

```
<engine-settings>
  <defaults>
    <expression integer-division="false" division-by-zero-is-null="false"/>
  </defaults>
</engine-settings>
```

The API to change the setting:

```
Configuration config = new Configuration();
config.EngineDefaults.Expression.IsIntegerDivision = true;
config.EngineDefaults.Expression.IsDivisionByZeroReturnsNull = true;
```

Subselect Evaluation Order

By default Esper updates sub-selects with new events before evaluating the enclosing statement. This is relevant for statements that look for the same event in both the `from` clause and subselects.

To have Esper evaluate the enclosing clauses before updating the subselect in a subselect expression, set the flag as indicated herein.

The XML configuration as below sets the same as the default value:

```
<engine-settings>
  <defaults>
    <expression self-subselect-preeval="true"/>
  </defaults>
</engine-settings>
```

Here is a sample statement that utilizes a sub-select against the same-events:

```
select * from MyEvent where prop not in (select prop from MyEvent.std:unique(otherProp))
```

By default the subselect data window updates first before the `where` clause is evaluated, thereby above statement never returns results.

Changing the setting described here causes the `where` clause to evaluate before the subselect data window updates, thereby the statement does post results.

User-Defined Function or Static Method Cache

By default Esper caches the result of an user-defined function if the parameter set to that function is empty or all parameters are constant values. Custom plug-in single-row function results are not cached.

To have Esper evaluate the user-defined function regardless of constant parameters, set the flag as indicated herein.

The XML configuration as below sets the same as the default value:

```
<engine-settings>
  <defaults>
    <expression udf-cache="true"/>
  </defaults>
</engine-settings>
```

Extended Built-in Aggregation Functions

By default Esper provides a number of additional aggregation functions over the SQL standards. To have Esper only allow the standard SQL aggregation functions and not the additional ones, disable the setting as described here.

The XML configuration as below sets the same as the default value:

```
<engine-settings>
  <defaults>
    <expression extend-agg="true"/>
  </defaults>
</engine-settings>
```

Duck Typing

By default Esper validates method references when using the dot operator syntax at time of statement creation. With duck typing, the engine resolves method references at runtime.

The XML configuration as below sets the same as the default value:

```
<engine-settings>
  <defaults>
    <expression ducktyping="false"/>
  </defaults>
</engine-settings>
```

13.4.19. Engine Settings related to Execution of Statements

Prioritized Execution

By default Esper ignores @Priority and @Drop annotations and executes unprioritized, that is the engine does not attempt to interpret assigned priorities and reorder executions based on priority. Use this setting if your application requires prioritized execution.

By setting this configuration, the engine executes statements, when an event or schedule matches multiple statements, according to the assigned priority, starting from the highest priority value. See built-in EPL annotations in Section 4.2.6.6, “@Priority”.

By enabling this setting view sharing between statements as described in Section 13.4.11.1, “Sharing View Resources between Statements” is disabled.

The XML configuration to enable the flag, which is disabled by default, is as follows:

```
<engine-settings>
```

```
<defaults>
  <execution prioritized="true" />
</defaults>
</engine-settings>
```

The API to change the setting:

```
Configuration config = new Configuration();
config.EngineDefaults.Execution.IsPrioritized = true;
```

13.4.20. Engine Settings related to Exception Handling

Use the settings as described here to register an exception handler factory class that provides an exception handler. The engine invokes exception handlers in the order they are listed to handle a continues-query unchecked exception, as further described in Section 12.10, “Exception Handling”.

Please provide the full-qualified class name of each class that implements the `com.espertech.esper.client.hook.ExceptionHandlerFactory` interface in the engine defaults configuration as below.

The XML configuration is as follows:

```
<engine-settings>
  <defaults>
    <exceptionHandling>
      <handlerFactory class="my.company.cep.MyCEPEngineExceptionHandlerFactory" />
    </exceptionHandling>
  </defaults>
</engine-settings>
```

The API calls to register an exception handler factory are as follows:

```
Configuration config = new Configuration();
config.EngineDefaults.ExceptionHandling.AddClass<MyCEPEngineExceptionHandlerFactory>();
```

13.4.21. Engine Settings related to Condition Handling

Use the settings as described here to register a condition handler factory class that provides a condition handler. The engine invokes condition handlers in the order they are listed to indicate conditions, which is the term used for notification when certain predefined limits are reached, as further described in Section 12.11, “Condition Handling”.

Please provide the full-qualified class name of each class that implements the `com.espertech.esper.client.hook.ConditionHandlerFactory` interface in the engine defaults configuration as below.

The XML configuration is as follows:

```
<engine-settings>
  <defaults>
    <conditionHandling>
      <handlerFactory class="my.company.cep.MyCEPEngineConditionHandlerFactory" />
    </conditionHandling>
  </defaults>
</engine-settings>
```


The API calls to register a condition handler factory are as follows:

```
Configuration config = new Configuration();
config.EngineDefaults.ConditionHandling.AddClass<MyCEPEngineConditionHandlerFactory>();
```

13.4.22. Revision Event Type

Revision event types reflect a versioning relationship between events of same or different event types. Please refer to Section 2.9, “Updating, Merging and Versioning Events” and Section 4.15.11, “Versioning and Revision Event Type Use with Named Windows”.

The configuration consists of the following:

- An name of an event type whose events are *base* events.
- Zero, one or more names of event types whose events are *delta* events.
- One or more property names that supply the key values that tie base and delta events to existing revision events. Properties must exist on the event type as simple properties. Nested, indexed or mapped properties are not allowed.
- Optionally, a strategy for overlaying or merging properties. The default strategy is *Overlay Declared* as described below.

The XML configuration for this setting is shown below:

```
<revision-event-type name="UserProfileRevisions">
  <base-event-type name="ProfileCreation"/>
  <delta-event-type name="ProfileUpdate"/>
  <key-property name="userid"/>
</revision-event-type>
```

If configuring via runtime or initialization-time API, this code snippet explains how:

```
Configuration config = new Configuration();
ConfigurationRevisionEventType configRev = new ConfigurationRevisionEventType();
configRev.NameBaseEventType = "ProfileCreation";
configRev.AddNameDeltaEventType("ProfileUpdate");
configRev.KeyPropertyNames = new String[] { "userid" };
config.AddRevisionEventType("UserProfileRevisions", configRev);
```

As the configuration provides names of base and delta event types, such names must be configured for Java-Bean, Map or XML events as the previous sections outline.

The next table outlines the available strategies:

Table 13.4. Property Revision Strategies

Name	Description
Overlay Declared (default)	<p>A fast strategy for revising events that groups properties provided by base and delta events and overlays contributed properties to compute a revision.</p> <p>For use when there is a limited number of combinations of properties that change on an event, and such combinations are known in advance.</p> <p>The properties available on the output revision events are all properties of the base event type. Delta event types do not add any additional properties that are not present</p>

Name	Description
	<p>on the base event type.</p> <p>Any null values or non-existing property on a delta (or base) event results in a null values for the same property on the output revision event.</p>
Merge Declared	<p>A strategy for revising events by merging properties provided by base and delta events, considering null values and non-existing (dynamic) properties as well.</p> <p>For use when there is a limited number of combinations of properties that change on an event, and such combinations are known in advance.</p> <p>The properties available on the output revision events are all properties of the base event type plus all additional properties that any of the delta event types provide.</p> <p>Any null values or non-existing property on a delta (or base) event results in a null values for the same property on the output revision event.</p>
Merge Non-null	<p>A strategy for revising events by merging properties provided by base and delta events, considering only non-null values.</p> <p>For use when there is an unlimited number of combinations of properties that change on an event, or combinations are not known in advance.</p> <p>The properties available on the output revision events are all properties of the base event type plus all additional properties that any of the delta event types provide.</p> <p>Null values returned by delta (or base) event properties provide no value to output revision events, i.e. null values are not merged.</p>
Merge Exists	<p>A strategy for revising events by merging properties provided by base and delta events, considering only values supplied by event properties that exist.</p> <p>For use when there is an unlimited number of combinations of properties that change on an event, or combinations are not known in advance.</p> <p>The properties available on the output revision events are all properties of the base event type plus all additional properties that any of the delta event types provide.</p> <p>All properties are treated as dynamic properties: If an event property does not exist on a delta event (or base) event the property provides no value to output revision events, i.e. non-existing property values are not merged.</p>

13.4.23. Variant Stream

A *variant stream* is a predefined stream into which events of multiple disparate event types can be inserted, and which can be selected from in patterns and the `from` clause.

The name of the variant stream and, optionally, the type of events that the stream may accept, are part of the stream definition. By default, the variant stream accepts only the predefined event types. The engine validates your `insert into` clause which inserts into the variant stream against the predefined types.

A variant stream can be set to accept any type of event, in which case all properties of the variant stream are effectively dynamic properties. Set the `type variance` flag to `ANY` to indicate the variant stream accepts any type of event.

The following XML configuration defines a variant stream by name `OrderStream` that carries only `PartsOrder` and `ServiceOrder` events:

```
<variant-stream name="OrderStream">
  <variant-event-type name="PartsOrder"/>
  <variant-event-type name="ServiceOrder"/>
</variant-stream>
```

This code snippet sets up a variant stream by name `OutgoingEvent`:

```
Configuration config = new Configuration();
ConfigurationVariantStream variant = new ConfigurationVariantStream();
variant.TypeVariance = ConfigurationVariantStream.TypeVariance.ANY;
config.AddVariantStream("OutgoingEvent", variant);
```

If specifying variant event type names, make sure such names have been configured for `JavaBean`, `Map` or `XML` events.

13.5. Type Names

Certain configuration values accept type names. Type names can occur in the configuration of variable types, `Map`-event property types as well as `XPath` cast types, for example. Types names are not case-sensitive.

The table below outlines all possible type names:

Table 13.5. Variable Type Names

Type Name	Type
string, varchar, varchar2 or System.String	A string value
int, integer or int?	An integer value
long or long?	A long value
bool, bool or bool?	A boolean value
double or double?	A double value
float or float?	A float value
short or short?	A short value
char, char or char?	A character value
byte or byte?	A byte value

13.6. Runtime Configuration

Certain configuration changes are available to perform on an engine instance while in operation. Such configuration operations are available via the `GetConfiguration` method on `EPAdministrator`, which returns an `ConfigurationOperations` object. Please consult the JavaDoc documentation for more detail.

13.7. Logging Configuration

Esper logs all messages to Apache commons logging under an appropriate log level. To output log messages you can add Log4j to classpath and configure Log4j as below. Or you may use SLF4J or no logging component or implement your own logging component.

For performance reasons, Esper does not log any debug-level or informational-level messages for event execution unless explicitly configured via Section 13.4.12.1, “Execution Path Debug Logging”.

A callback API for receiving certain critical engine reports is available as described in Section 12.10, “Exception Handling”.

Apache Commons Logging uses an automatic discovery mechanism to find a log framework that it will delegate to. If your application has another component using logback, please inspect your logback configuration and add `com.espertech`.

The next table explains the log levels:

Table 13.6. Log Levels

Log Level	Use
Debug	Displays detailed engine-internal information that may not be easy to understand for application developers but are useful for engine support.
Info	Used for a few critical engine-level log messages.
Warn	Certain important warning or informational messages are displayed under the warning level.
Error	Exceptions reported within the engine or by plug-in components are reported under the error level. When users enter invalid EPL statements such validation errors are not reported as error logs and are indicated via API exception instead.

13.7.1. Log4net Logging Configuration

Log4net is the default logging component. Please find additional information for Log4net configuration and extension in <http://logging.apache.org/log4net>.

The easiest way to configure Log4net is by providing a Log4net configuration file, similar to the `log4net.xml` file shipped in the `etc` folder of the distribution.

Add the `log4j.configuration` system property to the `java` command line and provide the file name of the Log4net configuration file, making sure your classpath also includes the directory of the file:

```
java -Dlog4j.configuration=log4j.xml ...
```

Chapter 14. Development Lifecycle

This chapter presents information related to the development lifecycle for developing an event processing application with EPL. It includes information on authoring, testing, debugging, packaging and deploying.

14.1. Authoring

Enterprise Edition includes authoring tools for EPL statements and modules by providing form-based dialogs, templates, an expression builder, simulation tool and other tools. Enterprise Edition also supports hot deployment and packaging options for EPL and related code.

EPL statements can be organized into modules as described below. Any text editor can edit EPL statement and module text. A text editor or IDE that highlights SQL syntax or keywords works.

For authoring configuration files please consult the XSD schema files as provided with the distribution.

For information on authoring event classes or event definitions in general please see Chapter 2, *Event Representations* or Section 4.16, “Declaring an Event Type: Create Schema”.

14.2. Testing

We recommend testing EPL statements using a test framework such as JUnit or TestNG. Please consult the Esper test suite for extensive examples, which can be downloaded from the distribution site.

Esper's API provides test framework classes to simplify automated testing of EPL statements. Please see ??? for more information.

We recommend performing latency and throughput tests early in the development lifecycle. Please consider the performance tips in Chapter 17, *Performance* for optimal performance.

Consider engine and statement metrics reporting for identifying slow-performing statements, for example. See Section 12.14, “Engine and Statement Metrics Reporting”.

14.3. Debugging

One important tool for debugging is the parameterized `@Audit` annotation. This annotation allows to output, on statement-level, detailed information about many aspects of statement processing.

Another tool for logging engine-level detail is Section 13.4.12.1, “Execution Path Debug Logging”.

Please see Section 13.7, “Logging Configuration” for information on configuring logging in general.

14.3.1. @Audit Annotation

Use the `@Audit` annotation to have the engine output detailed information about statement processing. The engine reports, at INFO level, the information under log name `com.espertech.esper.audit`. You may define an output format for audit information via configuration.

You may provide a comma-separated list of category names to `@Audit` to output information related to specific

categories only. The table below lists all available categories. If no parameter is provided, the engine outputs information for all categories. Category names are not case-sensitive.

For the next statement the engine produces detailed processing information (all categories) for the statement:

```
@Name('All Order Events') @Audit select * from OrderEvent
```

For the next statement the engine provides information about new events and also about event property values (2 categories are listed):

```
@Name('All Order Events') @Audit('stream,property') select price from OrderEvent
```

Here is a more complete example that uses the API to create the schema, create above statement and send an event:

```
epService.getEPAdministrator().createEPL("create schema OrderEvent(price double)");
String epl = "@Name('All-Order-Events') @Audit('stream,property') select price from OrderEvent";
epService.getEPAdministrator().createEPL(epl).addListener(listener);
epService.getEPRuntime().sendEvent(Collections.singletonMap("price", 100d), "OrderEvent");
```

The output is similar to the following:

```
INFO [audit] Statement All-Order-Events stream OrderEvent inserted {price=100.0}
INFO [audit] Statement All-Order-Events property price value 100.0
```

Table 14.1. @Audit Categories

Category	Description
Exprdef	Each expression declaration name and return value.
Expression	Each top-level expression and its return value.
Expression-nested	Each expression including child or nested expressions and their return value.
Insert	Each event inserted via insert-into.
Pattern	Each pattern sub-expression and its change in truth-value.
Pattern-instances	Each pattern sub-expression and its count of active instances.
Property	Each property name and the event's property value.
Schedule	Each schedule modification and trigger received by a statement.
Stream	Each new event received by a statement.
View	Each view name and its insert and remove stream.

Note that the engine only evaluates select-clause expressions if either a listener or subscriber is attached to the statement or if used with insert-into.

14.4. Packaging and Deploying Overview

Please consider Esper Enterprise Edition as a target deployment platform. Esper alone does not ship with a server as it is designed as a core CEP engine.

To support packaging and deploying event-driven applications, Esper offers infrastructure as outlined herein:

- EPL modules to build a cohesive, easily-externalizable deployment unit out of related statements as described in Section 14.5, “EPL Modules”.
- The deployment administrative interface is described in Section 14.6, “The Deployment Administrative Interface”.
- Instructions and code for use when the deployment target is a J2EE web application server or servlet runtime, please see Section 14.7, “J2EE Packaging and Deployment”.

14.5. EPL Modules

An EPL module file is a plain text file in which EPL statements appear separated by the semicolon (;) character. It bundles EPL statements with optional deployment instructions. A service provider instance keeps track of the known and/or deployed EPL modules and makes it easy to add, remove, deploy and undeploy EPL modules.

The synopsis of an EPL module file is:

```
[module module_name;  
[uses module_name; | import import_name;] [uses module_name; | import import_name;] [...]  
[epl_statement;] [epl_statement;] [...]
```

Use the `module` keyword followed a *module_name* identifier or a package (identifiers separated by dots) to declare the name of the module. The module name declaration must be at the beginning of the file, comments and whitespace excluded. The module name serves to check uses-dependences of other modules.

If a module file requires certain constructs that may be shared by other module files, such as named windows, variables, event types, variant streams or inserted-into streams required by statements, a module file may specify zero to many dependent modules with the `uses` keyword. At deployment time the engine checks the uses-dependencies and ensures that a module of that name is already deployed or will be deployed as part of the deployments. The deployment API supports ordering modules according to their uses-relationship.

If the EPL statements in the module require Java classes such as for underlying events or user-defined functions, use the `import` keyword followed by the fully-qualified class name or package name in the format `package.*`. The `uses` and `import` keywords are optional and must occur after the `module` declaration.

Following the optional deployment instructions are any number of *epl_statement* EPL statements that are separated by semicolon (;).

The following is a sample EPL module file explained in detail thereafter:

```
// Declare the name for the module  
module org.myorganization.switchmonitor;  
  
// Declare other module(s) that this module depends on  
uses org.myorganization.common;  
  
// Import any Java/.NET classes in an application package  
import org.myorganization.events.*;  
  
// Declare an event type based on a Java class in the package that was imported as above  
create schema MySwitchEvent as MySwitchEventPOJO;  
  
// Sample statement
```

```
@Name('Off-On-Detector')
insert into MyOffOnStream
select * from pattern[every-distinct(id) a=MySwitchEvent(status='off')
-> b=MySwitchEvent(id=a.id, status='on')];

// Sample statement
@Name('Count-Switched-On')
@Description('Count per switch id of the number of Off-to-On switches in the last 1 hour')
select id, count(*) from MyOffOnStream.win:time(1 hour) group by id;
```

The example above declares a module name of `org.myorganization.switchmonitor`. As defined by the `uses` keyword, it ensures that the `org.myorganization.common` module is already deployed. The example demonstrates the `import` keyword to make a package name known to the engine for resolving POJO class names, as the example assumes that `MySwitchEventPOJO` is a POJO event class. In addition the example module contains two statements separated by semicolon characters.

Your application code may, after deployment, look up a statement and attach listeners as shown here:

```
epService.getEPAdministrator().getStatement("Count-Switched-On").addListener(...);
```

14.6. The Deployment Administrative Interface

The `com.espertech.esper.client.deploy.EPDeploymentAdmin` service available from the `EPAdministrator` interface by method `getDeploymentAdmin` provides the functionality available to manage packaging and deployment. Please consult the JavaDoc documentation for more information.

The deployment API allows to read resources and parse text strings to obtain an object representation of the EPL module, the `Module`. A `Module` object can also be simply constructed.

After your application obtains a `Module` instance it may either use `deploy` to deploy the module directly, starting all statements of the module. Alternatively your application may add a module, making it known without starting statements for later deployment. In each case the module is assigned a deployment id, which acts as a unique primary key for all known modules. Your application may assign its own deployment id or may have the engine generate a deployment id (two footprints for `add` and `deploy` methods).

A module may be in two states: undeployed or deployed. When calling `add` to add a module, it starts life in the undeployed state. When calling `deploy` to deploy a module, it starts life in the deployed state. A module may be transitioned by providing the deployment id and by calling the `deploy` or `undeploy` methods.

Your code can remove a module in undeployed state using the `remove` method or the `undeployRemove` method. If the module is in deployed state, use `undeployRemove` to undeploy and remove the module.

The `DeploymentOptions` instance that can be passed to the `deploy` method when validating or deploying modules controls validation, fail-fast, rollback and the isolated service provider, if any, for the deployment. Also use `DeploymentOptions` to set a user object per statement or to set a statement name per statement.

We also provide additional sample code to read and deploy modules as part of the J2EE considerations below.

14.6.1. Reading Module Content

Read and parse module files via the `EPDeploymentAdmin` interface `read` and `parse` methods, which returns a `Module` instance to represent the module information.

This code snippet demonstrates reading and parsing a module given a file name:


```
EPServiceProvider epService = EPServiceProviderManager.getDefaultProvider();
EPDeploymentAdmin deployAdmin = epService.getEPAdministrator().getDeploymentAdmin();
Module module = deployAdmin.read(new File("switchmonitor.epl"));
```

The service provides additional read and parse methods to read from a URL, classpath, input stream or string.

14.6.2. Ordering Multiple Modules

Since modules may have inter-dependencies as discussed under the `uses` declaration, the deployment interface provides the `getDeploymentOrder` method to order a collection of modules before deployment.

Assuming your application reads multiple modules into a `mymodules` module list, this code snippet orders the modules for deployment and validates dependency declarations for each module:

```
List<Module> mymodules = ... read modules...;
DeploymentOrder order = deployAdmin.getDeploymentOrder(mymodules, new DeploymentOrderOptions());
```

14.6.3. Deploying and Undeploying

The deployment interface returns a deployment id for each module made known by adding a module or by deploying a module. To undeploy the module your application must provide the deployment id. Your application can assign its own deployment id or obtain the module name from the `Module` and use that as the deployment id.

The `undeploy` operation removes all named windows, variables, event types or any other information associated to the statements within the module to be undeployed.

The next code snippet deploys modules, starting each module's EPL statements:

```
for (Module mymodule : order.getOrdered()) {
    DeploymentResult deployResult = deployAdmin.deploy(mymodule, new DeploymentOptions());
}
```

Undeploying a module destroys all started statements associated to the module.

To undeploy and at the same time remove the module from the list of known modules use the `undeployRemove` method and pass the deployment id:

```
deployAdmin.undeployRemove(deployResult.getDeploymentId());
```

14.6.4. Listing Deployments

The deployment interface returns all module information that allows your application to determine which modules are known and their current state.

To obtain a list of all known modules or information for a specific module, the calls are:

```
DeploymentInformation[] info = deployAdmin.getDeploymentInformation();

// Given a deployment id, return the deployment information
DeploymentInformation infoModule = deployAdmin.getDeployment(deploymentId);
```

14.6.5. State Transitioning a Module

The following sample code adds a module, transitions the module to deployed, then undeploys and removes the module entirely;

```
// This sample uses the parse method to obtain a module
Module module = deployAdmin.parse("create schema MySchema (coll int)");

// Make the module know; It now shows up in undeployed state
String moduleDeploymentId = deployAdmin.add(module);

// Start all statements, passing a null options object for default options
deployAdmin.deploy(moduleDeploymentId, null);

// Undeploy module, destroying all statements
deployAdmin.undeploy(moduleDeploymentId);

// Remove module; It will no longer be known
deployAdmin.remove(moduleDeploymentId);
```

14.6.6. Best Practices

Use the `@Name` annotation to assign a name to each statement that your application would like to attach a listener or subscriber, or look up the statement for iteration or management by the administrative API.

Use the `create schema` syntax and the `import` keyword to define event types. When sharing event types, named windows or variables between modules use the `uses` keyword to declare a separate module that holds the shared definitions.

To validate whether a set of statements is complete and can start without issues, set the following flags on a `DeploymentOptions` instance passed to the `deploy` method as the code snippet below shows:

```
DeploymentOptions options = new DeploymentOptions();
options.setIsolatedServiceProvider("validation"); // we isolate any statements
options.setValidateOnly(true); // validate leaving no started statements
options.setFailFast(false); // do not fail on first error
epService.getEPAdministrator().getDeploymentAdmin()
    .deploy(module, options);
```

14.7. J2EE Packaging and Deployment

Esper can well be deployed as part of a J2EE web or enterprise application archive to a web application server. When designing for deployment into a J2EE web application server, please consider the items discussed here.

We provide a sample servlet context listener in this section that uses the deployment API to deploy and undeploy modules as part of the servlet lifecycle.

The distribution provides a message-driven bean (MDB) example that you may find useful.

Esper does not have a dependency on any J2EE or Servlet APIs to allow the engine to run in any environment or container.

14.7.1. J2EE Deployment Considerations

As multiple web applications deployed to a J2EE web application server typically have a separate classloader per application, you should consider whether engine instances need to be shared between applications or can remain separate engine instances. Consider the `EPServiceProviderManager` a Singleton. When deploying mul-

multiple web applications, your J2EE container classloader may provide a separate instance of the Singleton `EPServiceProviderManager` to each web application resulting in multiple independent engine instances.

To share `EPServiceProvider` instances between web applications, one approach is to add the Esper jar files to the system classpath. A second approach can be to have multiple web applications share the same servlet context and have your application place the `EPServiceProvider` instance into a servlet context attribute for sharing. Architecturally you may also consider a single archived application (such as an message-driven bean) that all your web applications communicate to via the JMS broker provided by your application server or an external JMS broker.

As per J2EE standards there are restrictions in regards to starting new threads in J2EE application code. Esper adheres to these restrictions: It allows to be driven entirely by external events. To remove all Esper threads, set the internal timer off and leave the advanced threading options turned off. To provide timer events when the internal timer is turned off, you should check with your J2EE application container for support of the Java system timer or for support of batch or work loading to send timer events to an engine instance.

As per J2EE standards there are restrictions in regards to input and output by J2EE application code. Esper adheres to these restrictions: By itself it does not start socket listeners or performs any file IO.

14.7.2. Servlet Context Listener

When deploying a J2EE archive that contains EPL modules files we provide sample code to read and deploy EPL modules files packaged with the enterprise or web application archive when the servlet initializes. The sample undeploys EPL modules when the servlet context gets destroyed.

A sample `web.xml` configuration extract is:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  <listener>
    <listener-class>SampleServletListener</listener-class>
  </listener>
  <context-param>
    <param-name>eplmodules</param-name>
    <param-value>switchmonitor.epl</param-value>
  </context-param>
</web-app>
```

A servlet listener that deploys EPL module files packaged into the archive on context initialization and that undeploys when the application server destroys the context is shown here:

```
public class SampleServletListener implements ServletContextListener {

    private List<String> deploymentIds = new ArrayList<String>();

    public void contextInitialized(ServletContextEvent servletContextEvent) {
        try {
            EPServiceProvider epServiceProvider = EPServiceProviderManager.getDefaultProvider();
            String modulesList = servletContextEvent.getServletContext().getInitParameter("eplmodules");
            List<Module> modules = new ArrayList<Module>();
            if (modulesList != null) {
                String[] split = modulesList.split(",");
                for (int i = 0; i < split.length; i++) {
                    String resourceName = split[i].trim();
                    if (resourceName.length() == 0) {
                        continue;
                    }
                    String realPath = servletContextEvent.getServletContext().getRealPath(resourceName);
                    Module module = epServiceProvider.getEPAdministrator()
                        .getDeploymentAdmin().read(new File(realPath));
                    modules.add(module);
                }
            }
        } catch (Exception e) {
            // Handle exception
        }
    }

    public void contextDestroyed(ServletContextEvent servletContextEvent) {
        // Undeploy modules
    }
}
```

```
    }  
  }  
  
  // Determine deployment order  
  DeploymentOrder order = epServiceProvider.getEPAdministrator()  
    .getDeploymentAdmin().getDeploymentOrder(modules, null);  
  
  // Deploy  
  for (Module module : order.getOrdered()) {  
    DeploymentResult result = epServiceProvider.getEPAdministrator()  
      .getDeploymentAdmin().deploy(module, new DeploymentOptions());  
    deploymentIds.add(result.getDeploymentId());  
  }  
}  
catch (Exception ex) {  
  ex.printStackTrace();  
}  
}  
  
public void contextDestroyed(ServletContextEvent servletContextEvent) {  
  EPServiceProvider epServiceProvider = EPServiceProviderManager.getDefaultProvider();  
  for (String deploymentId : deploymentIds) {  
    epServiceProvider.getEPAdministrator().getDeploymentAdmin().undeployRemove(deploymentId);  
  }  
}  
}
```

Chapter 15. Extension and Plug-in

Chapter 15. Extension and Plug-in

15.1. Overview

This chapter summarizes integration and describes in detail each of the extension APIs that allow integrating external data and/or extend engine functionality.

For information on input and output adapters that connect to an event transport and perform event transformation for incoming and outgoing on-the-wire event data, for use with streaming data, please see the EsperIO reference documentation. Consider using the Plug-in Loader API for creating a new adapter that starts or stops as part of the CEP engine initialization and destroy lifecycle, see Section 12.16, “Plug-in Loader”.

To join data that resides in a relational database and that is accessible via JDBC driver and SQL statement the engine offers a syntax for using SQL within EPL, see Section 4.13, “Accessing Relational Data via SQL”. A relational database input and output adapter for streaming input from and output to a relational database also exists (EsperIO).

To join data that resides in a non-relational store the engine offers a two means: First, the virtual data window, as described below, for transparently integrating the external store as a named window. The second mechanism is a special join syntax based on static method invocation, see Section 4.14, “Accessing Non-Relational Data via Method Invocation”.

15.2. Virtual Data Window

Use a virtual data window if you have a (large) external data store that you want to access as a named window. The access is transparent: There is no need to use special syntax or join syntax. All regular queries including subqueries, joins, on-merge, on-select, on-insert, on-delete, on-update and fire-and-forget are supported with virtual data windows.

There is no need to keep any data or events in memory with virtual data windows. The only requirement for virtual data windows is that all data rows returned are `EventBean` instances.

When implementing a virtual data window it is not necessary to send any events into the engine or to use insert-into. The event content is simply assumed to exist and accessible to the engine via the API implementation you provide.

The distribution ships with a sample virtual data window in the examples folder under the name `virtualdw`. The code snippets below are extracts from the example.

We use the term *store* here to mean a source set of data that is managed by the virtual data window. We use the term *store row* or just *row* to mean a single data item provided by the store. We use the term *lookup* to mean a read operation against the store returning zero, one or many rows.

Virtual data windows allow high-performance low-latency lookup by exposing all relevant EPL query access path information. This makes it possible for the virtual data window to choose the desired access method into its store.

The following steps are required to develop and use a virtual data window:

1. Implement the interface `com.espertech.esper.client.hook.VirtualDataWindowFactory`.
2. Implement the interface `com.espertech.esper.client.hook.VirtualDataWindow`.
3. Implement the interface `com.espertech.esper.client.hook.VirtualDataWindowLookup`.
4. Register the factory class in the engine configuration.

Once you have completed above steps, the virtual data window is ready to use in EPL statements.

The following restrictions apply to virtual data windows:

1. The virtual data window may not be enumerated via the `GetEnumerator` or `GetSafeEnumerator` methods.
2. A statement consuming a stream from a virtual data window does not get up-to-date state at statement creation time. For example, the EPL `select count(*) from MySampleWindow` always starts at count zero and not the count of events that are currently held in store.
3. When a virtual data window itself originates new events, use `insert-into` to insert into the named window.

From a threading perspective, virtual data window implementation classes must be thread-safe if objects are shared between multiple named windows. If no objects are shared between multiple different named windows, thereby each object is only used for the same named window and other named windows receive a separate instance, it is no necessary that the implementation classes are thread-safe.

15.2.1. How to Use

Your application must first register the virtual data window factory as part of engine configuration:

```
Configuration config = new Configuration();
config.AddPlugInVirtualDataWindow<SampleVirtualDataWindowFactory>("sample", "samplevdw");
config.AddPlugInVirtualDataWindow("sample2", "samplevdw2",
    typeof(SampleVirtualDataWindowFactory2).FullName);
```

Your application may then create a named window backed by a virtual data window.

For example, assume that the `SampleEvent` event type is declared as follows:

```
create schema SampleEvent as (key1 string, key2 string, value1 int, value2 double)
```

The next EPL statement creates a named window `MySampleWindow` that provides `SampleEvent` events and is backed by a virtual data window provided by `SampleVirtualDataWindowFactory` as configured above:

```
create window MySampleWindow.sample:samplevdw() as SampleEvent
```

You may then access the named window, same as any other named window, for example by subquery, join, on-action, fire-and-forget query or by consuming its insert and remove stream. While this example uses Map-type events, the example code is the same for POJO or other events.

Your application may obtain a reference to the virtual data window from the engine context.

This code snippet looks up the virtual data window by the named window name:

```
try {
    return (VirtualDataWindow) epService.Context.Lookup("/virtualdw/MySampleWindow");
}
catch (NamingException e) {
    throw new ApplicationException("Failed to look up virtual data window, is it created yet?");
}
```

Query Access Path

When your application registers a subquery, join or on-action query or executes a fire-and-forget query against a virtual data window the engine interacts with the virtual data window. The interaction is a two-step process.

At time of EPL statement creation (once), the engine analyzes the EPL where-clause, if present. It then compiles a list of hash-index and binary tree (btree, i.e. sorted) index properties. It passes the property names that are queried as well as the operators (i.e. =, >, range etc.) to the virtual data window. The virtual data window returns a lookup strategy object to the engine.

At time of EPL statement execution (repeatedly as triggered), the engine uses that lookup strategy object to execute a lookup. It passes to the lookup all actual key values (hash, btree including ranges) to make fast and efficient lookup achievable.

To explain in detail, assume that your application creates an EPL statement with a subquery as follows:

```
select (select * from MySampleWindow where key1 = 'A1') from OtherEvent
```

At the time of creation of the EPL query above the engine analyzes the EPL query. It determines that the subquery queries a virtual data window. It determines from the where-clause that the lookup uses property `key1` and hash-equals semantics. The engine then provides this information as part of `VirtualDataWindowLookupContext` passed to the `getLookup` method. Your application may inspect hash and btree properties and may determine the appropriate store access method to use.

The hash and btree property lookup information is for informational purposes, to enable fast and performant queries that returns the smallest number of rows possible. Your implementation classes may use some or none of the information provided and may also instead return some or perhaps even all rows, as is practical to your implementation. The where-clause still remains in effect and gets evaluated on all rows that are returned by the lookup strategy.

Following the above example, the sub-query executes once when a `OtherEvent` event arrives. At time of execution the engine delivers the string value `A1` to the `VirtualDataWindowLookup` lookup implementation provided by your application. The lookup object queries the store and returns store rows as `EventBean` instances.

As a second example, consider an EPL join statement as follows:

```
select * from MySampleWindow, MyTriggerEvent where key1 = trigger1 and key2 = trigger2
```

The engine analyzes the query and passes to the virtual data window the information that the lookup occurs on properties `key1` and `key2` under hash-equals semantics. When a `MyTriggerEvent` arrives, it passes the actual value of the `trigger1` and `trigger2` properties of the current `MyTriggerEvent` to the lookup.

As a last example, consider an EPL fire-and-forget statement as follows:

```
select * from MySampleWindow key1 = 'A2' and value1 between 0 and 1000
```

The engine analyzes the query and passes to the virtual data window the lookup information. The lookup occurs on property `key1` under hash-equals semantics and on property `value1` under btree-open-range semantics. When your application executes the fire-and-forget query the engine passes `A2` and the range endpoints `0` and `1000` to the lookup.

For more information, please consult the JavaDoc API documentation for class `com.espertech.esper.client.hook.VirtualDataWindow`, `VirtualDataWindowLookupContext` or `VirtualDataWindowLookupFieldDesc`.

15.2.2. Implementing the Factory

A virtual data window factory class is responsible for the following functions:

- Accept the factory context object as a parameter and return the `VirtualDataWindow` implementation.

The sample code shown here can be found among the examples in the distribution under `virtualdw`:

```
public class SampleVirtualDataWindowFactory : IVirtualDataWindowFactory {
    public IVirtualDataWindow Create(VirtualDataWindowContext context) {
        return new SampleVirtualDataWindow(context);
    }
}
```

Your factory class must implement the `create` method which receives a `VirtualDataWindowContext` object. This method is called once for each EPL that creates a virtual data window (see example `create window` above).

The `VirtualDataWindowContext` provides to your application:

```
String namedWindowName; // Name of named window being created.
Object[] parameters; // Any optional parameters provided as part of create-window.
EventType eventType; // The event type of events.
EventBeanFactory eventFactory; // A factory for creating EventBean instances from store rows.
VirtualDataWindowOutputStream outputStream; // For stream output to consuming statements.
StatementContext statementContext; // Other EPL statement information.
```

15.2.3. Implementing the Virtual Data Window

A virtual data window implementation is responsible for the following functions:

- Accept the lookup context object as a parameter and return the `VirtualDataWindowLookup` implementation.
- Optionally, post insert and remove stream data.

The sample code shown here can be found among the examples in the distribution under `virtualdw`.

The implementation class must implement the `VirtualDataWindow` interface like so:

```
public class SampleVirtualDataWindow : IVirtualDataWindow {

    private readonly VirtualDataWindowContext context;

    public SampleVirtualDataWindow(VirtualDataWindowContext context) {
        this.context = context;
    } ...
}
```

When the engine compiles an EPL statement and detects a virtual data window, the engine invokes the `getLookup` method indicating hash and btree access path information by passing a `VirtualDataWindowLookupContext` context. The lookup method must return a `VirtualDataWindowLookup` implementation that the EPL statement uses for all lookups until the EPL statement is stopped or destroyed.

The sample implementation does not use the hash and btree access path information and simply returns a lookup object:

```
public VirtualDataWindowLookup GetLookup(VirtualDataWindowLookupContext desc) {

    // Place any code that interrogates the hash-index and btree-index fields here.

    // Return the lookup strategy.
}
```



```
return new SampleVirtualDataWindowLookup(context);
}
```

If your virtual data window returns null instead of a lookup object, the EPL query creation fails and throws an `EPStatementException`.

The engine calls the `update` method when data changes because of on-merge, on-delete, on-update or insert-into. For example, if you have an on-merge statement that is triggered and that updates the virtual data window, the `newData` parameter receives the new (updated) event and the `oldData` parameters receives the event prior to the update. Your code may use these events to update the store or delete from the store, if needed.

If your application plans to consume data from the virtual data window, for example via `select * from MySampleWindow`, then the code must implement the `update` method to forward insert and remove stream events, as shown below, to receive the events in consuming statements. To post insert and remove stream data, use the `VirtualDataWindowOutputStream` provided by the context object as follows.

```
public void update(Object sender, UpdateEventArgs e) {
    // This sample simply posts into the insert and remove stream what is received.
    context.OutputStream.Update(sender, e);
}
```

Your application should not use `VirtualDataWindowOutputStream` to post new events that originate from the store. The object is intended for use with on-action EPL statements. Use insert-into instead for any new events that originate from the store.

15.2.4. Implementing the Lookup

A lookup implementation is responsible for the following functions:

- Accept the lookup values as a parameter and return a set of `EventBean` instances.

The sample code shown here can be found among the examples in the distribution under `virtualdw`.

The implementation class must implement the `VirtualDataWindowLookup` interface:

```
public class SampleVirtualDataWindowLookup : IVirtualDataWindowLookup {
    private readonly VirtualDataWindowContext context;

    public SampleVirtualDataWindowLookup(VirtualDataWindowContext context) {
        this.context = context;
    } ...
}
```

When an EPL query fires, the engine invokes the lookup and provides the actual lookup values. The lookup values are provided in the same exact order as the access path information that the engine provided when obtaining the lookup.

Each store row must be wrapped as an `EventBean` instance. The context object provides an `EventBeanFactory` implementation returned by `getEventFactory()` that can be used to wrap rows.

The sample implementation does not use the lookup values and simply returns a hardcoded sample event:

```
public ICollection<EventBean> Lookup(Object[] lookupValues) {
    // Add code to interrogate lookup values here.

    // Create sample event.
    // This example uses Map events; Other underlying events such as POJO are exactly the same code.
    IDictionary<String, Object> eventData = new Dictionary<String, Object>();
}
```

```

eventData["key1" = "sample1";
eventData["key2" = "sample2";
eventData["value1" = 100;
eventData["value2"] = 1.5d;
EventBean @event = context.EventFactory.Wrap(eventData);
return new EventBean[]{ @event };
}

```

The `lookupValues` object array represents all actual joined property values or expression results if you where-clause criteria are expressions. The code may use these keys to for efficient store access.

When a key value is a range, the key value is an instance of `VirtualDataWindowKeyRange`.

15.3. Single-Row Functions

Single-row functions return a single value. They are not expected to aggregate rows but instead should be stateless functions. These functions can appear in any expressions and can be passed any number of parameters.

The following steps are required to develop and use a custom single-row function with Esper.

1. Implement a class providing one or more public static methods accepting the number and type of parameters as required.
2. Register the single-row function class and method name with the engine by supplying a function name, via the engine configuration file or the configuration API.

You may not override a built-in function with a single-row function provided by you. The single-row function you register must have a different name than any of the built-in functions.

An example single-row function can also be found in the examples under the runtime configuration example.

15.3.1. Implementing a Single-Row Function

Single-row function classes have no further requirement than provide a public static method.

The following sample single-row function simply computes a percentage value based on two number values.

This sample class provides a public static method by name `computePercent` to return a percentage value:

```

public class MyUtilityClass {
    public static double computePercent(double amount, double total) {
        return amount / total * 100;
    }
}

```

15.3.2. Configuring the Single-Row Function Name

The class name of the class, the method name and the function name of the new single-row function must be added to the engine configuration via the configuration API or using the XML configuration file. The configuration shown below is XML however the same options are available through the runtime and static configuration API:

```

<esper-configuration
  <plugin-singlerow-function name="percent"
    function-class="mycompany.MyUtilityClass" function-method="computePercent" />
</esper-configuration>

```

Note that the function name and method name need not be the same.

The new single-row function is now ready to use in a statement:

```
select percent(50,100) from MyEvent
```

If the single-row function returns an object that provides further functions, you may chain function calls.

The following demonstrates a chained single-row function. The example assumes that a single-row function by name `calculator` returns an object that provides the `add` function which accepts two parameters:

```
select calculator().add(5, amount) from MyEvent
```

15.4. Derived-value and Data Window View

Views in Esper are used to derive information from an event stream, and to represent data windows onto an event stream. This chapter describes how to plug-in a new, custom view.

The following steps are required to develop and use a custom view with Esper.

1. Implement a view factory class. View factories are classes that accept and check view parameters and instantiate the appropriate view class.
2. Implement a view class. A view class commonly represents a data window or derives new information from a stream.
3. Configure the view factory class supplying a view namespace and name in the engine configuration file.

The example view factory and view class that are used in this chapter can be found in the examples source folder in the OHLC (open-high-low-close) example. The class names are `OHLCBarPlugInViewFactory` and `OHLCBarPlugInView`.

Views can make use of the following engine services available via `StatementServiceContext`:

- The `SchedulingService` interface allows views to schedule timer callbacks to a view
- The `EventAdapterService` interface allows views to create new event types and event instances of a given type.
- The `StatementStopService` interface allows view to register a callback that the engine invokes to indicate that the view's statement has been stopped

Section 15.4.3, “View Contract” outlines the requirements for correct behavior of a your custom view within the engine.

Note that custom views may use engine services and APIs that can be subject to change between major releases. The engine services discussed above and view APIs are considered part of the engine internal public API and are stable. Any changes to such APIs are disclosed through the release change logs and history. Please also consider contributing your custom view to the Esper project team by submitting the view code through the mailing list or via a JIRA issue.

15.4.1. Implementing a View Factory

A view factory class is responsible for the following functions:

- Accept zero, one or more view parameters. View parameters are themselves expressions. The view factory must validate and evaluate these expressions.

- Instantiate the actual view class.
- Provide information about the event type of events posted by the view.

View factory classes simply subclass `com.espertech.esper.view.ViewFactorySupport`:

```
public class OHLCBarPlugInViewFactory : ViewFactorySupport { ...
```

Your view factory class must implement the `SetViewParameters` method to accept and parse view parameters. The next code snippet shows an implementation of this method. The code checks the number of parameters and retains the parameters passed to the method:

```
public class OHLCBarPlugInViewFactory : ViewFactorySupport {
    private ViewFactoryContext viewFactoryContext;
    private IList<ExprNode> viewParameters;
    private ExprNode timestampExpression;
    private ExprNode valueExpression;

    public void SetViewParameters(ViewFactoryContext viewFactoryContext, IList<ExprNode> viewParameters) {
        this.viewFactoryContext = viewFactoryContext;
        if (viewParameters.size() != 2) {
            throw new ViewParameterException(
                "View requires a two parameters: " +
                "the expression returning timestamps and the expression supplying OHLC data points");
        }
        this.viewParameters = viewParameters;
    }
    ...
}
```

After the engine supplied view parameters to the factory, the engine will ask the view to attach to its parent view and validate any parameter expressions against the parent view's event type. If the view will be generating events of a different type then the events generated by the parent view, then the view factory can create the new event type in this method:

```
public void Attach(EventType parentEventType,
    StatementContext statementContext,
    ViewFactory optionalParentFactory,
    IList<ViewFactory> parentViewFactories) {

    ExprNode[] validatedNodes = ViewFactorySupport.Validate("OHLC view",
        parentEventType, statementContext, viewParameters, false);

    timestampExpression = validatedNodes[0];
    valueExpression = validatedNodes[1];

    if ((timestampExpression.ExprEvaluator.ReturnType != typeof(long)) &&
        (timestampExpression.ExprEvaluator.ReturnType != typeof(long?))) {
        throw new ViewParameterException(
            "View requires long-typed timestamp values in parameter 1");
    }
    if ((valueExpression.ExprEvaluator.ReturnType != typeof(double)) &&
        (valueExpression.ExprEvaluator.ReturnType != typeof(double?))) {
        throw new ViewParameterException(
            "View requires double-typed values for in parameter 2");
    }
}
```

Finally, the engine asks the view factory to create a view instance, and asks for the type of event generated by the view:

```
public View MakeView(StatementContext statementContext) {
    return new OHLCBarPlugInView(statementContext, timestampExpression, valueExpression);
}

public EventType EventType {
    get { return OHLCBarPlugInView.GetEventType(viewFactoryContext.EventAdapterService) }
}
```

```
}
```

15.4.2. Implementing a View

A view class is responsible for:

- The `Parent` property informs the view of the parent view's event type
- The `Update` method receives insert streams and remove stream events from its parent view
- The `GetEnumerator` method supplies an (optional) iterator to allow an application to pull or request results from an `EPStatement`
- The `CloneView` method must make a configured copy of the view to enable the view to work in a grouping context together with a `std:groupwin` parent view

View classes simply subclass `com.espertech.esper.view.ViewSupport`:

```
public class MyTrendSpotterView : ViewSupport { ...
```

Your view's `Update` method will be processing incoming (insert stream) and outgoing (remove stream) events posted by the parent view (if any), as well as providing incoming and outgoing events to child views. The convention required of your update method implementation is that the view releases any insert stream events (`EventBean` object references) which the view generates as reference-equal remove stream events (`EventBean` object references) at a later time.

The view implementation must call the `UpdateChildren` method to post outgoing insert and remove stream events. Similar to the `Update` method, the `UpdateChildren` method takes insert and remove stream events as parameters.

A sample `Update` method implementation is provided in the OHLC example.

15.4.3. View Contract

The `Update` method must adhere to the following conventions, to prevent memory leaks and to enable correct behavior within the engine:

- A view implementation that posts events to the insert stream must post unique `EventBean` object references as insert stream events, and cannot post the same `EventBean` object reference multiple times. The underlying event to the `EventBean` object reference can be the same object reference, however the `EventBean` object reference posted by the view into the insert stream must be a new instance for each insert stream event.
- If the custom view posts a continuous insert stream, then the views must also post a continuous remove stream (second parameter to the `UpdateChildren` method). If the view does not post remove stream events, it assumes unbound keep-all semantics.
- `EventBean` events posted as remove stream events must be the same object reference as the `EventBean` events posted as insert stream by the view. Thus remove stream events posted by the view (the `EventBean` instances, does not affect the underlying representation) must be reference-equal to insert stream events posted by the view as part of an earlier invocation of the update method, or the same invocation of the update method.
- `EventBean` events represent a unique observation. The values of the observation can be the same, thus the underlying representation of an `EventBean` event can be reused, however event property values must be kept immutable and not be subject to change.
- Array elements of the insert and remove stream events must not carry null values. Array size must match the number of `EventBean` instances posted. It is recommended to use a `null` value for no insert or remove stream events rather than an empty zero-size array.

The engine can provide a callback to the view indicating when a statement using the view is stopped. The callback is available to the view via the `com.espertech.esper.view.StatementStopCallback` interface. Your view code must subscribe to the stop callback in order for the engine to invoke the callback:

```
statementContext.StatementStopService.AddSubscriber(this);
```

Please refer to the sample views for a code sample on how to implement `iterator` and `CloneView` methods.

In terms of multiple threads accessing view state, there is no need for your custom view factory or view implementation to perform any synchronization to protect internal state. The iterator of the custom view implementation does also not need to be thread-safe. The engine ensures the custom view executes in the context of a single thread at a time. If your view uses shared external state, such external state must be still considered for synchronization when using multiple threads.

15.4.4. Configuring View Namespace and Name

The view factory class name as well as the view namespace and name for the new view must be added to the engine configuration via the configuration API or using the XML configuration file. The configuration shown below is XML however the same options are available through the configuration API:

```
<esper-configuration
  <plugin-view namespace="custom" name="ohlcv"
    factory-class="com.espertech.esper.example.ohlcv.OHLCBarPlugInViewFactory" />
</esper-configuration>
```

The new view is now ready to use in a statement:

```
select * from StockTick.custom:ohlcv(timestamp, price)
```

Note that the view must implement additional interfaces if it acts as a data window view, or works in a grouping context, as discussed in detail below.

15.4.5. Requirement for Data Window Views

Your custom view may represent an expiry policy and may retain events and thus act as a data window view. In order to allow the engine to validate that your view can be used with named windows, which allow only data window views, this section documents any additional requirement that your classes must fulfill.

Your view factory class must implement the `com.espertech.esper.view.DataWindowViewFactory` interface. This marker interface (no methods required) indicates that your view factory provides only data window views.

Your view class must implement the `com.espertech.esper.view.DataWindowView` interface. This marker interface indicates that your view is a data window view and therefore eligible to be used in any construct that requires a data window view.

15.4.6. Requirement for Grouped Views

Grouped views are views that operate under the `std:groupwin` view. When operating under one or more `std:groupwin` views, the engine instantiates a single view instance when the statement starts, and a new view instance per group criteria dynamically as new group criteria become known.

The next statement shows EPL for using a view instance per grouping criteria:

```
select * from StockTick.std:groupwin(symbol).custom:trendspotter(price)
```

Your view must implement the `com.espertech.esper.view.CloneableView` interface to indicate your view may create new views. This code snippet shows a sample implementation of the `CloneView` method required by the interface:

```
public View CloneView(StatementContext statementContext) {
    return new MyPlugInView(statementContext); // pass any parameters along where
}
```

15.5. Aggregation Functions

Aggregation functions aggregate event property values or expression results obtained from one or more streams. Examples for built-in aggregation functions are `count(*)`, `sum(price * volume)` or `avg(distinct volume)`.

The optional keyword `distinct` ensures that only distinct (unique) values are aggregated and duplicate values are ignored by the aggregation function. Custom plug-in aggregation functions do not need to implement the logic to handle `distinct` values. This is because when the engine encounters the `distinct` keyword, it eliminates any non-distinct values before passing the value for aggregation to the custom aggregation function.

Custom aggregation functions can also be passed multiple parameters, as further described in Section 15.5.3, “Accepting Multiple Parameters”. In the example below the aggregation function accepts a single parameter.

The following steps are required to develop and use a custom aggregation function with Esper.

1. Implement an aggregation function class.
2. Register the aggregation function class with the engine by supplying a function name, via the engine configuration file or the runtime and static configuration API.

The code for the example aggregation function as shown in this chapter can be found in the runtime configuration example in the package `com.espertech.esper.example.runtimeconfig` by the name `MyConcatAggregationFunction`. The sample function simply concatenates string-type values.

15.5.1. Implementing an Aggregation Function

An aggregation function class is responsible for the following functions:

- Implement a `Validate` method that validates the value type of the data points that the function must process.
- Implement a `ValueType` property that returns the type of the aggregation value generated by the function. For example, the built-in `count` aggregation function returns `Long.class` as it generates `long`-typed values.
- Implement an `Enter` method that the engine invokes to add a data point into the aggregation, when an event enters a data window
- Implement a `Leave` method that the engine invokes to remove a data point from the aggregation, when an event leaves a data window
- Implement a `Value` property that returns the current value of the aggregation.

Aggregation function classes simply subclass `com.espertech.esper.epl.agg.AggregationSupport`:

```
public class MyConcatAggregationFunction : AggregationSupport { ...
```

The engine generally constructs one instance of the aggregation function class for each time the function is listed in a statement, however the engine may decide to reduce the number of aggregation class instances if it finds equivalent aggregations. The class that provides the aggregation and extends `AggregationSupport` does not have to be threadsafe.

The constructor initializes the aggregation function:

```
public class MyConcatAggregationFunction : AggregationSupport {
    private readonly static char DELIMITER = ' ';
    private StringBuilder builder;
    private String delimiter;

    public MyConcatAggregationFunction() : base()
    {
        builder = new StringBuilder();
        delimiter = "";
    }
    ...
}
```

An aggregation function must provide an implementation of the `Validate` method that is passed a `AggregationValidationContext` validation context object. Within the validation context you find the result type of each of the parameters expressions to the aggregation function as well as information about constant values and data window use. Please see the API documentation for a comprehensive list of validation context information.

Since the example concatenation function requires string types, it implements a type check:

```
public void Validate(AggregationValidationContext validationContext) {
    if ((validationContext.ParameterTypes.Length != 1) ||
        (validationContext.ParameterTypes[0] != typeof(string))) {
        throw new ArgumentException("Concat aggregation requires a single parameter of type String");
    }
}
```

The `Enter` method adds a datapoint to the current aggregation value. The example `Enter` method shown below adds a delimiter and the string value to a string buffer:

```
public void Enter(Object value) {
    if (value != null) {
        builder.Append(delimiter);
        builder.Append(value.ToString());
        delimiter = DELIMITER.ToString();
    }
}
```

Conversly, the `Leave` method removes a datapoint from the current aggregation value. The example `Leave` method removes from the string buffer:

```
public void Leave(Object value) {
    if (value != null) {
        builder.Length = 0;
    }
}
```

In order for the engine to validate the type returned by the aggregation function against the types expected by enclosing expressions, the `ValueType` must return the result type of any values produced by the aggregation function:

```
public Class ValueType {
    get { return typeof(string); }
}
```


For on-demand queries the aggregation function must support resetting its value to empty or start values. Implement the `Clear` function to reset the value as shown below:

```
public void Clear() {
    builder = new StringBuilder();
    delimiter = "";
}
```

Finally, the engine obtains the current aggregation value by means of the `Value` property:

```
public Object Value {
    get { return builder.ToString(); }
}
```

15.5.2. Configuring the Aggregation Function Name

The aggregation function class name as well as the function name for the new aggregation function must be added to the engine configuration via the configuration API or using the XML configuration file. The configuration shown below is XML however the same options are available through the configuration API:

```
<esper-configuration
  <plugin-aggregation-function name="concat "
    function-class="com.espertech.esper.example.runtimeconfig.MyConcatAggregationFunction" />
</esper-configuration>
```

The new aggregation function is now ready to use in a statement:

```
select concat(symbol) from StockTick.win:length(3)
```

15.5.3. Accepting Multiple Parameters

Your plug-in aggregation function may accept multiple parameters, simply by casting the `Object` parameter of the `Enter` and `Leave` method to `Object[]`.

For instance, assume an aggregation function `rangeCount` that counts all values that fall into a range of values. The EPL that calls this function and provides a lower and upper bounds of 1 and 10 is:

```
select rangeCount(1, 10, myValue) from MyEvent
```

The `Enter` method of the plug-in aggregation function may look as follows:

```
public void Enter(Object value) {
    Object[] params = (Object[]) value;
    int lower = (int) params[0];
    int upper = (int) params[1];
    int val = (int) params[2];
    if ((val >= lower) && (val <= upper)) {
        count++;
    }
}
```

Your plug-in aggregation function may want to validate parameter types or may want to know which parameters are constant-value expressions. Constant-value expressions are evaluated only once by the engine and could therefore be cached by your aggregation function for performance reasons. The engine provides constant-value information as part of the `AggregationValidationContext` passed to the `Validate` method.

15.6. Pattern Guard

Pattern guards are pattern objects that control the lifecycle of the guarded sub-expression, and can filter the events fired by the subexpression.

The following steps are required to develop and use a custom guard object with Esper.

1. Implement a guard factory class, responsible for creating guard object instances.
2. Implement a guard class.
3. Register the guard factory class with the engine by supplying a namespace and name, via the engine configuration file or the configuration API.

The code for the example guard object as shown in this chapter can be found in the test source folder in the package `com.espertech.esper.regression.client` by the name `MyCountToPatternGuardFactory`. The sample guard discussed here counts the number of events occurring up to a maximum number of events, and end the sub-expression when that maximum is reached.

15.6.1. Implementing a Guard Factory

A guard factory class is responsible for the following functions:

- Implement a `SetGuardParameters` method that takes guard parameters, which are themselves expressions.
- Implement a `MakeGuard` method that constructs a new guard instance.

Guard factory classes subclass `com.espertech.esper.pattern.guard.GuardFactorySupport`:

```
public class MyCountToPatternGuardFactory : GuardFactorySupport { ...
```

The engine constructs one instance of the guard factory class for each time the guard is listed in a statement.

The guard factory class implements the `SetGuardParameters` method that is passed the parameters to the guard as supplied by the statement. It verifies the guard parameters, similar to the code snippet shown next. Our example counter guard takes a single numeric parameter:

```
public void SetGuardParameters(IList<ExprNode> guardParameters,
                               MatchedEventConverter convertor) {
    String message = "Count-to guard takes a single integer-value expression as parameter";
    if (guardParameters.Count != 1) {
        throw new GuardParameterException(message);
    }

    if (guardParameters[0].ExprEvaluator.ReturnType != typeof(int)) {
        throw new GuardParameterException(message);
    }

    this.numCountToExpr = guardParameters[0];
    this.convertor = convertor;
}
```

The `MakeGuard` method is called by the engine to create a new guard instance. The example `MakeGuard` method shown below passes the maximum count of events to the guard instance. It also passes a `Quitable` implementation to the guard instance. The guard uses `Quitable` to indicate that the sub-expression contained within must stop (quit) listening for events.

```
public Guard MakeGuard(PatternContext context,
                      MatchedEventMap beginState,
                      Quitable quitable,
```

```

    Object stateNodeId,
    Object guardState) {

    Object parameter = PatternExpressionUtil.Evaluate("Count-to guard",
        beginState, numCountToExpr, convertor);
    if (parameter == null) {
        throw new EPEException("Count-to guard parameter evaluated to a null value");
    }

    int? numCountTo = (int?) parameter;
    return new MyCountToPatternGuard(numCountTo, quitable);
}

```

15.6.2. Implementing a Guard Class

A guard class has the following responsibilities:

- Provides a `StartGuard` method that initializes the guard.
- Provides a `StopGuard` method that stops the guard, called by the engine when the whole pattern is stopped, or the sub-expression containing the guard is stopped.
- Provides an `Inspect` method that the pattern engine invokes to determine if the guard lets matching events pass for further evaluation by the containing expression.

Guard classes subclass `com.espertech.esper.pattern.guard.GuardSupport` as shown here:

```
public abstract class GuardSupport : Guard { ...
```

The engine invokes the guard factory class to construct an instance of the guard class for each new sub-expression instance within a statement.

A guard class must provide an implementation of the `StartGuard` method that the pattern engine invokes to start a guard instance. In our example, the method resets the guard's counter to zero:

```
public void StartGuard() {
    counter = 0;
}
```

The pattern engine invokes the `Inspect` method for each time the sub-expression indicates a new event result. Our example guard needs to count the number of events matched, and quit if the maximum number is reached:

```
public bool Inspect(MatchedEventMap matchEvent) {
    counter++;
    if (counter > numCountTo) {
        quitable.GuardQuit();
        return false;
    }
    return true;
}
```

The `Inspect` method returns true for events that pass the guard, and false for events that should not pass the guard.

15.6.3. Configuring Guard Namespace and Name

The guard factory class name as well as the namespace and name for the new guard must be added to the engine configuration via the configuration API or using the XML configuration file. The configuration shown below is XML however the same options are available through the configuration API:

```
<esper-configuration
  <plugin-pattern-guard namespace="myplugin" name="count_to"
    factory-class="com.espertech.esper.regression.client.MyCountToPatternGuardFactory"/>
</esper-configuration>
```

The new guard is now ready to use in a statement. The next pattern statement detects the first 10 MyEvent events:

```
select * from pattern [(every MyEvent) where myplugin:count_to(10)]
```

Note that the `every` keyword was placed within parentheses to ensure the guard controls the repeated matching of events.

15.7. Pattern Observer

Pattern observers are pattern objects that are executed as part of a pattern expression and can observe events or test conditions. Examples for built-in observers are `timer:at` and `timer:interval`. Some suggested uses of observer objects are:

- Implement custom scheduling logic using the engine's own scheduling and timer services
- Test conditions related to prior events matching an expression

The following steps are required to develop and use a custom observer object within pattern statements:

1. Implement an observer factory class, responsible for creating observer object instances.
2. Implement an observer class.
3. Register an observer factory class with the engine by supplying a namespace and name, via the engine configuration file or the configuration API.

The code for the example observer object as shown in this chapter can be found in the test source folder in package `com.espertech.esper.regression.client` by the name `MyFileExistsObserver`. The sample observer discussed here very simply checks if a file exists, using the filename supplied by the pattern statement, and via the `System.IO.File` class.

15.7.1. Implementing an Observer Factory

An observer factory class is responsible for the following functions:

- Implement a `SetObserverParameters` method that takes observer parameters, which are themselves expressions.
- Implement a `MakeObserver` method that constructs a new observer instance.

Observer factory classes subclass `com.espertech.esper.pattern.observer.ObserverFactorySupport`:

```
public class MyFileExistsObserverFactory : ObserverFactorySupport { ...
```

The engine constructs one instance of the observer factory class for each time the observer is listed in a statement.

The observer factory class implements the `SetObserverParameters` method that is passed the parameters to the observer as supplied by the statement. It verifies the observer parameters, similar to the code snippet shown next. Our example file-exists observer takes a single string parameter:

```
public void SetObserverParameters(IList<ExprNode> expressionParameters,
```

```

        MatchedEventConverter convertor) {
    String message = "File exists observer takes a single string filename parameter";
    if (expressionParameters.Count != 1) {
        throw new ObserverParameterException(message);
    }
    if (expressionParameters[0].ExprEvaluator.ReturnType != typeof(string)) {
        throw new ObserverParameterException(message);
    }

    this.filenameExpression = expressionParameters[0];
    this.convertor = convertor;
}

```

The pattern engine calls the `MakeObserver` method to create a new observer instance. The example `MakeObserver` method shown below passes parameters to the observer instance:

```

public EventObserver MakeObserver(PatternContext context,
                                MatchedEventMap beginState,
                                ObserverEventEvaluator observerEventEvaluator,
                                Object stateNodeId,
                                Object observerState) {
    Object filename = PatternExpressionUtil.Evaluate("File-exists observer ", beginState, filenameExpr);
    if (filename == null) {
        throw new EPEException("Filename evaluated to null");
    }

    return new MyFileExistsObserver(beginState, observerEventEvaluator, filename.ToString());
}

```

The `ObserverEventEvaluator` parameter allows an observer to indicate events, and to indicate change of truth value to permanently false. Use this interface to indicate when your observer has received or witnessed an event, or changed its truth value to true or permanently false.

The `MatchedEventMap` parameter provides a `Map` of all matching events for the expression prior to the observer's start. For example, consider a pattern as below:

```
a=MyEvent -> myplugin:my_observer(...)
```

The above pattern tagged the `MyEvent` instance with the tag "a". The pattern engine starts an instance of `my_observer` when it receives the first `MyEvent`. The observer can query the `MatchedEventMap` using "a" as a key and obtain the tagged event.

15.7.2. Implementing an Observer Class

An observer class has the following responsibilities:

- Provides a `StartObserve` method that starts the observer.
- Provides a `StopObserve` method that stops the observer, called by the engine when the whole pattern is stopped, or the sub-expression containing the observer is stopped.

Observer classes subclass `com.espertech.esper.pattern.observer.ObserverSupport` as shown here:

```
public class MyFileExistsObserver : EventObserver { ...
```

The engine invokes the observer factory class to construct an instance of the observer class for each new sub-expression instance within a statement.

An observer class must provide an implementation of the `StartObserve` method that the pattern engine invokes to start an observer instance. In our example, the observer checks for the presence of a file and indicates the

truth value to the remainder of the expression:

```
public void startObserve() {
    File file = new File(filename);
    if (file.exists()) {
        observerEventEvaluator.observerEvaluateTrue(beginState);
    }
    else {
        observerEventEvaluator.observerEvaluateFalse();
    }
}
```

Note the observer passes the `ObserverEventEvaluator` an instance of `MatchedEventMap`. The observer can also create one or more new events and pass these events through the Map to the remaining expressions in the pattern.

15.7.3. Configuring Observer Namespace and Name

The observer factory class name as well as the namespace and name for the new observer must be added to the engine configuration via the configuration API or using the XML configuration file. The configuration shown below is XML however the same options are available through the configuration API:

```
<esper-configuration
  <plugin-pattern-observer namespace="myplugin" name="file_exists"
    factory-class="com.espertech.esper.regression.client.MyFileExistsObserverFactory" />
</esper-configuration>
```

The new observer is now ready to use in a statement. The next pattern statement checks every 10 seconds if the given file exists, and indicates to the listener when the file is found.

```
select * from pattern [every timer:interval(10 sec) -> myplugin:file_exists("myfile.txt")]
```

15.8. Event Type And Event Object

Creating a plug-in event representation can be useful under any of these conditions:

- Your application has existing types that carry event metadata and event property values and your application does not want to (or cannot) extract or transform such event metadata and event data into one of the built-in event representations (Native objects, Map or XML DOM).
- Your application must perform a network lookup or other dynamic resolution of event type and events.

Note that the classes to plug-in custom event representations are held stable between minor releases, but can be subject to change between major releases.

Currently, EsperIO provides the following additional event representations:

The source code is available for these and they are therefore excellent examples for how to implement a plug-in event representation. Please see the EsperIO documentation for usage details.

15.8.1. How It Works

Your application provides a plug-in event representation as an implementation of the `com.espertech.esper.plugin.PluginEventRepresentation` interface. It registers the implementation class in the Configuration and at the same time provides a unique URI. This URI is called the root event representation URI. An example value for a root URI is `type://xml/apacheaxiom/OMNode`.

One can register multiple plug-in event representations. Each representation has a root URI. The root URI serves to divide the overall space of different event representations and plays a role in resolving event types and event objects.

There are two situations in an Esper engine instance asks an event representation for an event type:

1. When an application registers a new event type using the method `AddPlugInEventType` on `ConfigurationOperations`, either at runtime or at configuration time.
2. When an EPL statement is created with a new event type name (a name not seen before) and the URIs for resolving such names are set beforehand via `PlugInEventTypeNameResolutionURIs` on `ConfigurationOperations`.

The implementation of the `PlugInEventRepresentation` interface must provide implementations for two key interfaces: `com.espertech.esper.client.EventType` and `EventBean`. It must also implement several other related interfaces as described below.

The `EventType` methods provide event metadata including property names and property types. They also provide instances of `EventPropertyGetter` for retrieving event property values. Each instance of `EventType` represents a distinct type of event.

The `EventBean` implementation is the event itself and encapsulates the underlying event object.

15.8.2. Steps

Follow the steps outlined below to process event objects for your event types:

1. Implement the `EventType`, `EventPropertyGetter` and `EventBean` interfaces.
2. Implement the `PlugInEventRepresentation` interface, the `PlugInEventTypeHandler` and `PlugInEventBeanFactory` interfaces, then add the `PlugInEventRepresentation` class name to configuration.
3. Register plug-in event types, and/or set the event type name resolution URIs, via configuration.
4. Obtain an `EventSender` from `EPRuntime` via the `GetEventSender(Uri[])` method and use that to send in your event objects.

Please consult the SDK documentation for further information on each of the interfaces and their respective methods.

15.8.3. URI-based Resolution

Assume you have registered event representations using the following URIs:

1. `type://myFormat/myProject/myName`
2. `type://myFormat/myProject`
3. `type://myFormat/myOtherProject`

When providing an array of child URIs for resolution, the engine compares each child URI to each of the event representation root URIs, in the order provided. Any event representation root URIs that spans the child URI space becomes a candidate event representation. If multiple root URIs match, the order is defined by the more specific root URI first, to the least specific root URI last.

During event type resolution and event sender resolution you provide a child URI. Assuming the child URI provided is `type://myFormat/myProject/myName/myEvent?param1=abc¶m2=true`. In this example both root URIs #1 (the more specific) and #1 (the less specific) match, while root URI #3 is not a match. Thus at the

time of type resolution the engine invokes the `AcceptType` method on event presentation for URI #1 first (the more specific), before asking #2 (the less specific) to resolve the type.

The `EventSender` returned by the `GetEventSender(Uri[])` method follows the same scheme. The event sender instance asks each matching event representation for each URI to resolve the event object in the order of most specific to least specific root URI, and the first event representation to return an instance of `EventBean` ends the resolution process for event objects.

The `type://` part of the URI is an optional convention for the scheme part of an URI that your application may follow. URIs can also be simple names and can include parameters, as the MSDN documents for class `System.Uri` [<http://msdn.microsoft.com/en-us/library/system.uri.aspx>].

15.8.4. Example

This section implements a minimal sample plug-in event representation. For the sake of keeping the example easy to understand, the event representation is rather straightforward: an event is a `System.Collections.Generic.IDictionary<string, string>` object that consists of key-values pairs of type `string`.

The code shown next does not document method footprints. Please consult the MSDN documentation for method details.

Sample Event Type

First, the sample shows how to implement the `EventType` interface. The event type provides information about property names and types, as well as supertypes of the event type.

Our `EventType` takes a set of valid property names:

```
public class MyPlugInPropertiesEventType : EventType {
    private readonly IDictionary<string, string> properties;

    public MyPlugInPropertiesEventType(IDictionary<string, string> properties) {
        this.properties = properties;
    }

    public Type GetPropertyType(string property) {
        if (!IsProperty(property)) {
            return null;
        }
        return typeof(string);
    }

    public Type UnderlyingType {
        get { return typeof(IDictionary<string, string>); }
    }

    //... further methods below
}
```

An `EventType` is responsible for providing implementations of `EventPropertyGetter` to query actual events. The getter simply queries the `Properties` object underlying each event:

```
public EventPropertyGetter GetGetter(string property) {
    readonly String propertyName = property;

    return new EventPropertyGetter() {
        public Object get(EventBean eventBean) throws PropertyAccessException {
            MyPlugInPropertiesEventBean propBean = (MyPlugInPropertiesEventBean) eventBean;
            return propBean.getProperties().getProperty(propertyName);
        }
    }
}
```



```

    public boolean isExistsProperty(EventBean eventBean) {
        MyPlugInPropertiesEventBean propBean = (MyPlugInPropertiesEventBean) eventBean;
        return propBean.getProperties().getProperty(propertyName) != null;
    }

    public Object getFragment(EventBean eventBean) {
        return null; // The property is not a fragment
    }
};
}

```

Our sample `EventType` does not have supertypes. Supertypes represent an extends-relationship between event types, and subtypes are expected to exhibit the same event property names and types as each of their supertypes combined:

```

public EventType[] SuperTypes {
    get { return null; } // no supertype for this example
}

public EventType[] DeepSuperTypes {
    get { return null ; }
}

public string Name {
    get { return name; }
}

public EventPropertyDescriptor[] PropertyDescriptors {
    get {
        return descriptors.Values.ToArray(); // Use of LINQ - ToArray() extension
    }
}

public EventPropertyDescriptor GetPropertyDescriptor(string propertyName) {
    return descriptors.Get(propertyName); // Use of compat extension
}

public FragmentEventType GetFragmentType(string property) {
    return null; // sample does not provide any fragments
}

```

The example event type as above does not provide fragments, which are properties of the event that can themselves be represented as an event, to keep the example simple.

Sample Event Bean

Each `EventBean` instance represents an event. The interface is straightforward to implement. In this example an event is backed by a `Properties` object:

```

public class MyPlugInPropertiesEventBean : EventBean {
    private readonly MyPlugInPropertiesEventType eventType;

    public MyPlugInPropertiesEventBean(MyPlugInPropertiesEventType eventType,
        IDictionary<string, string> properties) {
        this.eventType = eventType;
        this.Properties = properties;
    }

    public EventType EventType {
        get { return eventType; }
    }

    public Object Get(string property) {
        EventPropertyGetter getter = eventType.GetGetter(property);
        return getter.Get(this);
    }
}

```

```

}

public Object GetFragment(string property) {
    EventPropertyGetter getter = eventType.GetGetter(property);
    if (getter != null) {
        return getter.GetFragment(this);
    }
    return null;
}

public Object Underlying {
    get { return properties; }
}

protected IDictionary<string, string> Properties {
    get; private set;
}
}

```

Sample Event Representation

A `PlugInEventRepresentation` serves to create `EventType` and `EventBean` instances through its related interfaces.

The sample event representation creates `MyPlugInPropertiesEventType` and `MyPlugInPropertiesEventBean` instances. The `PlugInEventHandler` returns the `EventType` instance and an `EventSender` instance.

Our sample event representation accepts all requests for event types by returning boolean true on the `AcceptType` method. When asked for the `PlugInEventHandler`, it constructs a new `EventType`. The list of property names for the new type is passed as an initialization value provided through the configuration API or XML, as a comma-separated list of property names:

```

public class MyPlugInEventRepresentation : PlugInEventRepresentation {

    private IList<MyPlugInPropertiesEventType> types;

    public void Init(PlugInEventRepresentationContext context) {
        types = new List<MyPlugInPropertiesEventType>();
    }

    public bool AcceptsType(PlugInEventHandlerContext context) {
        return true;
    }

    public PlugInEventHandler GetTypeHandler(PlugInEventHandlerContext eventTypeContext) {
        String proplist = (String) eventTypeContext.TypeInitializer;
        String[] propertyList = proplist.Split(',');

        HashSet<String> typeProps = new HashSet<String>(propertyList);

        MyPlugInPropertiesEventType eventType = new MyPlugInPropertiesEventType(typeProps);
        types.Add(eventType);

        return new MyPlugInPropertiesEventHandler(eventType);
    }
    // ... more methods below
}

```

The `PlugInEventHandler` simply returns the `EventType` as well as an implementation of `EventSender` for processing same-type events:

```

public class MyPlugInPropertiesEventHandler : PlugInEventHandler {
    private readonly MyPlugInPropertiesEventType eventType;

    public MyPlugInPropertiesEventHandler(MyPlugInPropertiesEventType eventType) {

```

```

    this.eventType = eventType;
}

public EventSender GetSender(EPRuntimeEventSender runtimeEventSender) {
    return new MyPlugInPropertiesEventSender(eventType, runtimeEventSender);
}

public EventType EventType {
    get { return eventType; }
}
}

```

The `EventSender` returned by `PlugInEventHandler` is expected process events of the same type or any subtype:

```

public class MyPlugInPropertiesEventSender : EventSender {
    private readonly MyPlugInPropertiesEventType type;
    private readonly EPRuntimeEventSender runtimeSender;

    public MyPlugInPropertiesEventSender(MyPlugInPropertiesEventType type,
        EPRuntimeEventSender runtimeSender) {
        this.type = type;
        this.runtimeSender = runtimeSender;
    }

    public void SendEvent(Object @event) {
        if (!(@event is IDictionary<string,string>)) {
            throw new EPEException("Sender expects a properties event");
        }
        EventBean eventBean = new MyPlugInPropertiesEventBean(type, (IDictionary<string,string>) @event);
        runtimeSender.ProcessWrappedEvent(eventBean);
    }
}

```

Sample Event Bean Factory

The plug-in event representation may optionally provide an implementation of `PlugInEventBeanFactory`. `PlugInEventBeanFactory` may inspect event objects and assign an event type dynamically based on resolution URIs and event properties.

Our sample event representation accepts all URIs and returns a `MyPlugInPropertiesBeanFactory`:

```

public class MyPlugInEventRepresentation : PlugInEventRepresentation {

    // ... methods as seen earlier
    public bool AcceptsEventBeanResolution(
        PlugInEventBeanReflectorContext eventBeanContext) {
        return true;
    }

    public PlugInEventBeanFactory GetEventBeanFactory(
        PlugInEventBeanReflectorContext eventBeanContext) {
        return new MyPlugInPropertiesBeanFactory(types);
    }
}

```

Last, the sample `MyPlugInPropertiesBeanFactory` implements the `PlugInEventBeanFactory` interface. It inspects incoming events and determines an event type based on whether all properties for that event type are present:

```

public class MyPlugInPropertiesBeanFactory : PlugInEventBeanFactory {
    private readonly IList<MyPlugInPropertiesEventType> knownTypes;

    public MyPlugInPropertiesBeanFactory(IList<MyPlugInPropertiesEventType> types) {

```

```
    knownTypes = types;
}

public EventBean create(Object @event, URI resolutionURI) {
    Properties properties = (Properties) @event;

    // use the known types to determine the type of the object
    foreach (MyPlugInPropertiesEventType type in knownTypes) {
        // if there is one property the event does not contain, then its not the right type
        bool hasAllProperties = true;
        foreach (String prop in type.PropertyNames) {
            if (!properties.ContainsKey(prop)) {
                hasAllProperties = false;
                break;
            }
        }

        if (hasAllProperties) {
            return new MyPlugInPropertiesEventBean(type, properties);
        }
    }
    return null; // none match, unknown event
}
```

Chapter 16. Examples, Tutorials, Case Studies

16.1. Examples Overview

This chapter outlines the examples that come with Esper in the `examples` folder of the distribution. Each sample is in a separate folder that contains all files needed by the example, excluding jar files.

Here is an overview over the examples in alphabetical order:

Table 16.1. Examples

Name	Description
Section 16.3, “AutoID RFID Reader”	<p>An array of RFID readers sense RFID tags as pallets are coming within the range of one of the readers.</p> <p>Shows the use of an XSD schema and XML event representation. A single statement shows a rolling time window, a where-clause filter on a nested property and a group-by.</p>
Section 16.5, “Market Data Feed Monitor”	<p>Processes a raw market data feed and reports throughput statistics and detects when the data rate of a feed falls off unexpectedly.</p> <p>Demonstrates a batch time window and a rolling time window with a having-clause. Multi-threaded example with a configurable number of threads and a simulator for generating feed data.</p>
Section 16.11, “MatchMaker”	<p>In the MatchMaker example every mobile user has an X and Y location and the task of the event patterns created by this example is to detect mobile users that are within proximity given a certain range, and for which certain properties match preferences.</p> <p>Dynamically creates and removes event patterns that use range matching based on mobile user events received.</p>
Section 16.12, “Named Window Query”	<p>A mini-benchmark that handles temperature sensor events. The sample creates a named window and fills it with a large number of events. It then executes a large number of pre-defined queries as well as fire-and-forget queries and reports times.</p> <p>Study this example if you are interested in named windows, Map event type representation, fire-and-forget queries as well as pre-defined queries via on-select, and the performance aspects.</p>
Section 16.6, “OHLC Plug-in View”	<p>A plug-in custom view addressing a problem in the financial space: Computes open-high-low-close bars for minute-intervals of events that may arrive late, based on each event's timestamp.</p> <p>A custom plug-in view based on the extension API can be a convenient and</p>

Name	Description
	reusable way to express a domain-specific analysis problem as a unit, and this example includes the code for the OHLC view factory and view as well as simulator to test the view.
Section 17.3, “Using the performance kit”	A benchmark that is further described in the performance section of this document under performance kit.
Section 16.13, “Quality of Service”	<p>This example develops some code for measuring quality-of-service levels such as for a service-level agreement (SLA).</p> <p>This example combines patterns with select-statements, shows the use of the timer 'at' operator and followed-by operator ->, and uses the iterator API to poll for current results.</p>
Section 16.9, “Assets Moving Across Zones - An RFID Example”	<p>An example out of the RFID domain processes location report events. The example includes a simple Swing-based GUI for visualization allows moving tags from zone to zone visually. It also contains comprehensive simulator to generate data for a large number of asset groups and their tracking.</p> <p>The example hooks up statements that aggregate and detect patterns in the aggregated data to determine when an asset group constraint is violated.</p>
Section 16.4, “Runtime Configuration”	Example code to demonstrate various key runtime configuration options such as adding event types on-the-fly, adding new variables, adding plug-in single-row and aggregation functions and adding variant streams and revision type definition.
Section 16.10, “StockTicker”	<p>An example from the financial domain that features event patterns to filter stock tick events based on price and symbol. The example is designed to provide a high volume of events and includes multithreaded unit test code as well as a simulating data generator.</p> <p>Perhaps this is a good example to learn the API and get started with event patterns. The example dynamically creates and removes event patterns based on price limit events received.</p>
Section 16.8, “Self-Service Terminal”	<p>A J2EE-based self-service terminal managing system in an airport that gets a lot of events from connected terminals.</p> <p>Contains a message-driven bean (EJB-MDB) for use in a J2EE container, a client and a simulator, as well as EPL statements for detecting various conditions. A version that runs outside of a J2EE container is also available.</p>

16.2. Running the Examples

In order to compile and run the samples please follow the below instructions:

1. Make sure .NET 3.5 or greater is installed.
2. Open a console window and change directory to `examples/example_name/etc`.
3. Run `"setenv.bat"` (Windows) or `"setenv.sh"` (Unix) to verify your environment settings.
4. Run `"compile.bat"` (Windows) or `"compile.sh"` (Unix) to compile an example.
5. Now you are ready to run an example. Some examples require mandatory parameters that are also described in the file `"readme.txt"` in the `"etc"` folder.
6. Modify the logger logging level in the `"log4j.xml"` configuration file changing `DEBUG` to `INFO` on a class or package level to control the volume of text output.

Each example also provides Eclipse project `.classpath` and `.project` files. The Eclipse projects expect an `esper_runtime` user library that includes the runtime dependencies.

JUnit tests exist for the example code. The JUnit test source code for the examples can be found in each example's `src/test` folder. To build and run the example JUnit tests, use the Maven 2 goal `test`.

16.3. AutoID RFID Reader

In this example an array of RFID readers sense RFID tags as pallets are coming within the range of one of the readers. A reader generates XML documents with observation information such as reader sensor ID, observation time and tags observed. A statement computes the total number of tags per reader sensor ID within the last 60 seconds.

This example demonstrates how XML documents unmarshalled to `System.Xml.Node` DOM document nodes can natively be processed by the engine without requiring native object event representations. The example uses an XPath expression for an event property counting the number of tags observed by a sensor. The XML documents follow the AutoID (<http://www.autoid.org/>) organization standard.

The classes for this example can be found in package `com.espertech.esper.example.autoid`. As events are XML documents with no native object representation, the example does not have event classes.

A simulator that can be run from the command line is also available for this example. The simulator generates a number of XML documents as specified by a command line argument and prints out the totals per sensor. Run `"run_autoid.bat"` (Windows) or `"run_autoid.sh"` (Unix) to start the AutoID simulator. Please see the readme file in the same folder for build instructions and command line parameters.

The code snippet below shows the simple statement to compute the total number of tags per sensor. The statement is created by class `com.espertech.esper.example.autoid.RFIDTagsPerSensorStmt`.

```
select ID as sensorId, sum(countTags) as numTagsPerSensor
from AutoIdRFIDExample.win:time(60 seconds)
where Observation[0].Command = 'READ_PALLET_TAGS_ONLY'
group by ID
```

16.4. Runtime Configuration

This example demonstrates various key runtime configuration options such as adding event types on-the-fly, adding new variables, adding plug-in single-row and aggregation functions and adding variant streams and revision type definition.

The classes for this example live in package `com.espertech.esper.example.runtimeconfig`.

16.5. Market Data Feed Monitor

This example processes a raw market data feed. It reports throughput statistics and detects when the data rate of a feed falls off unexpectedly. A rate fall-off may mean that the data is stale and we want to alert when there is a possible problem with the feed.

The classes for this example live in package `com.espertech.esper.example.marketdatafeed`. Run "run_mktdatfeed.bat" (Windows) or "run_mktdatfeed.sh" (Unix) in the `examples/etc` folder to start the market data feed simulator.

16.5.1. Input Events

The input stream consists of 1 event stream that contains 2 simulated market data feeds. Each individual event in the stream indicates the feed that supplies the market data, the security symbol and some pricing information:

```
String symbol;  
FeedEnum feed;  
double bidPrice;  
double askPrice;
```

16.5.2. Computing Rates Per Feed

For the throughput statistics and to detect rapid fall-off we calculate a ticks per second rate for each market data feed.

We can use an EPL statement that specifies a view onto the market data event stream that batches together 1 second of events. We specify the feed and a count of events per feed as output values. To make this data available for further processing, we insert output events into the `TicksPerSecond` event stream:

```
insert into TicksPerSecond  
select feed, count(*) as cnt  
  from MarketDataEvent.win:time_batch(1 second)  
 group by feed
```

16.5.3. Detecting a Fall-off

We define a rapid fall-off by alerting when the number of ticks per second for any second falls below 75% of the average number of ticks per second over the last 10 seconds.

We can compute the average number of ticks per second over the last 10 seconds simply by using the `TicksPerSecond` events computed by the prior statement and averaging the last 10 seconds. Next, we compare the current rate with the moving average and filter out any rates that fall below 75% of the average:

```
select feed, avg(cnt) as avgCnt, cnt as feedCnt  
  from TicksPerSecond.win:time(10 seconds)  
 group by feed  
having cnt < avg(cnt) * 0.75
```

16.5.4. Event generator

The simulator generates market data events for 2 feeds, feed A and feed B. The first parameter to the simulator is a number of threads. Each thread sends events for each feed in an endless loop. Note that as the garbage collection kicks in, the example generates rate drop-offs during such pauses.

The second parameter is a rate drop probability parameter specifies the probability in percent that the simulator drops the rate for a randomly chosen feed to 60% of the target rate for that second. Thus rate fall-off alerts can be generated.

The third parameter defines the number of seconds to run the example.

16.6. OHLC Plug-in View

This example contains a fully-functional custom view based on the extension API that computes OHLC open-high-low-close bars for events that provide a long-typed timestamp and a double-typed value.

OHLC bar is a problem out of the financial domain. The "Open" refers to the first datapoint and the "Close" to the last datapoint in an interval. The "High" refers to the maximum and the "Low" to the minimum value during each interval. The term "bar" is used to describe each interval results of these 4 values.

The example provides an OHLC view that is hardcoded to 1-minute bars. It considers the timestamp value carried by each event, and not the system time. The cutoff time after which an event is no longer considered for a bar is hardcoded to 5 seconds.

The view assumes that events arrive in timestamp order: Each event's timestamp value is equal to or higher than the timestamp value provided by the prior event.

The view may also be used together with `std:groupwin` to group per criteria, such as symbol. In this case the assumption of timestamp order applies per symbol.

The view gracefully handles no-event and late-event scenarios. Interval boundaries are defined by system time, thus event timestamp and system time must roughly be in-sync, unless using external timer events.

16.7. Transaction 3-Event Challenge

The classes for this example live in package `com.espertech.esper.example.transaction`. Run "run_txnsim.bat" (Windows) or "run_txnsim.sh" (Unix) to start the transaction simulator. Please see the readme file in the same folder for build instructions and command line parameters.

16.7.1. The Events

The use case involves tracking three components of a transaction. It's important that we use at least three components, since some engines have different performance or coding for only two events per transaction. Each component comes to the engine as an event with the following fields:

- Transaction ID
- Time stamp

In addition, we have the following extra fields:

In event A:

- Customer ID

In event C:

- Supplier ID (the ID of the supplier that the order was filled through)

16.7.2. Combined event

We need to take in events A, B and C and produce a single, combined event with the following fields:

- Transaction ID
- Customer ID
- Time stamp from event A
- Time stamp from event B
- Time stamp from event C

What we're doing here is matching the transaction IDs on each event, to form an aggregate event. If all these events were in a relational database, this could be done as a simple SQL join... except that with 10,000 events per second, you will need some serious database hardware to do it.

16.7.3. Real time summary data

Further, we need to produce the following:

- Min,Max,Average total latency from the events (difference in time between A and C) over the past 30 minutes.
- Min,Max,Average latency grouped by (a) customer ID and (b) supplier ID. In other words, metrics on the the latency of the orders coming from each customer and going to each supplier.
- Min,Max,Average latency between events A/B (time stamp of B minus A) and B/C (time stamp of C minus B).

16.7.4. Find problems

We need to detect a transaction that did not make it through all three events. In other words, a transaction with events A or B, but not C. Note that, in this case, what we care about is event C. The lack of events A or B could indicate a failure in the event transport and should be ignored. Although the lack of an event C could also be a transport failure, it merits looking into.

16.7.5. Event generator

To make testing easier, standard and to demonstrate how the example works, the example is including an event generator. The generator generates events for a given number of transactions, using the following rules:

- One in 5,000 transactions will skip event A
- One in 1,000 transactions will skip event B
- One in 10,000 transactions will skip event C.
- Transaction identifiers are randomly generated
- Customer and supplier identifiers are randomly chosen from two lists
- The time stamp on each event is based on the system time. Between events A and B as well as B and C, between 0 and 999 is added to the time. So, we have an expected time difference of around 500 milliseconds between each event
- Events are randomly shuffled as described below

To make things harder, we don't want transaction events coming in order. This code ensures that they come completely out of order. To do this, we fill in a bucket with events and, when the bucket is full, we shuffle it. The buckets are sized so that some transactions' events will be split between buckets. So, you have a fairly randomized flow of events, representing the worst case from a big, distributed infrastructure.

The generator lets you change the size of the bucket (small, medium, large, larger, largerer). The larger the bucket size, the more events potentially come in between two events in a given transaction and so, the more the performance characteristics like buffers, hashes/indexes and other structures are put to the test as the bucket size increases.

16.8. Self-Service Terminal

The example is about a J2EE-based self-service terminal managing system in an airport that gets a lot of events from connected terminals. The event rate is around 500 events per second. Some events indicate abnormal situations such as 'paper low' or 'terminal out of order'. Other events observe activity as customers use a terminal to check in and print boarding tickets.

16.8.1. Events

Each self-service terminal can publish any of the 6 events below.

- Checkin - Indicates a customer started a check-in dialog
- Cancelled - Indicates a customer cancelled a check-in dialog
- Completed - Indicates a customer completed a check-in dialog
- OutOfOrder - Indicates the terminal detected a hardware problem
- LowPaper - Indicates the terminal is low on paper
- Status - Indicates terminal status, published every 1 minute regardless of activity as a terminal heartbeat

All events provide information about the terminal that published the event, and a timestamp. The terminal information is held in a property named "term" and provides a terminal id. Since all events carry similar information, we model each event as a subtype to a base class BaseTerminalEvent, which will provide the terminal information that all events share. This enables us to treat all terminal events polymorphically, that is we can treat derived event types just like their parent event types. This helps simplify our queries.

All terminals publish Status events every 1 minute. In normal cases, the Status events indicate that a terminal is alive and online. The absence of status events may indicate that a terminal went offline for some reason and that may need to be investigated.

16.8.2. Detecting Customer Check-in Issues

A customer may be in the middle of a check-in when the terminal detects a hardware problem or when the network goes down. In that situation we want to alert a team member to help the customer. When the terminal detects a problem, it issues an OutOfOrder event. A pattern can find situations where the terminal indicates out-of-order and the customer is in the middle of the check-in process:

```
select * from pattern [ every a=Checkin ->
    ( OutOfOrder(term.id=a.term.id) and not
      (Cancelled(term.id=a.term.id) or Completed(term.id=a.term.id)) ) ]
```

16.8.3. Absence of Status Events

Since Status events arrive in regular intervals of 60 seconds, we can make use of temporal pattern matching using timer to find events that didn't arrive. We can use the every operator and timer:interval() to repeat an action every 60 seconds. Then we combine this with a not operator to check for absence of Status events. A 65 second interval during which we look for Status events allows 5 seconds to account for a possible delay in transmission or processing:

```
select 'terminal 1 is offline' from pattern
  [every timer:interval(60 sec) -> (timer:interval(65 sec) and not Status(term.id = 'T1'))]
output first every 5 minutes
```

16.8.4. Activity Summary Data

By presenting statistical information about terminal activity to our staff in real-time we enable them to monitor the system and spot problems. The next example query simply gives us a count per event type every 1 minute. We could further use this data, available through the CountPerType event stream, to join and compare against a recorded usage pattern, or to just summarize activity in real-time.

```
insert into CountPerType
select type, count(*) as countPerType
from BaseTerminalEvent.win:time(10 minutes)
group by type
output all every 1 minutes
```

16.9. Assets Moving Across Zones - An RFID Example

This example out of the RFID domain processes location report events. Each location report event indicates an asset id and the current zone of the asset. The example solves the problem that when a given set of assets is not moving together from zone to zone, then an alert must be fired.

Each asset group is tracked by 2 statements. The two statements to track a single asset group consisting of assets identified by asset ids {1, 2, 3} are as follows:

```
insert into CountZone_G1
select 1 as groupId, zone, count(*) as cnt
from LocationReport(assetId in 1, 2, 3).std:unique(assetId)
group by zone

select Part.zone from pattern [
  every Part=CountZone_G1(cnt in (1,2)) ->
    (timer:interval(10 sec) and not CountZone_G1(zone=Part.zone, cnt in (0,3)))]
```

The classes for this example can be found in package `com.espertech.esper.example.rfid`.

This example provides a Swing-based GUI that can be run from the command line. The GUI allows drag-and-drop of three RFID tags that form one asset group from zone to zone. Each time you move an asset across the screen the example sends an event into the engine indicating the asset id and current zone. The example detects if within 10 seconds the three assets do not join each other within the same zone, but stay split across zones. Run "run_rfid_swing.bat" (Windows) or "run_rfid_swing.sh" (Unix) to start the example's Swing GUI.

The example also provides a simulator that can be run from the command line. The simulator generates a number of asset groups as specified by a command line argument and starts a number of threads as specified by a command line argument to send location report events into the engine. Run "run_rfid_sim.bat" (Windows) or "run_rfid_sim.sh" (Unix) to start the RFID location report event simulator. Please see the readme file in the same folder for build instructions and command line parameters.

16.10. StockTicker

The StockTicker example comes from the stock trading domain. The example creates event patterns to filter stock tick events based on price and symbol. When a stock tick event is encountered that falls outside the lower or upper price limit, the example simply displays that stock tick event. The price range itself is dynamically created and changed. This is accomplished by an event patterns that searches for another event class, the price limit event.

The classes `com.espertech.esper.example.stockticker.event.StockTick` and `PriceLimit` represent our events. The event patterns are created by the class `com.espertech.esper.example.stockticker.monitor.StockTickerMonitor`.

Summary:

- Good example to learn the API and get started with event patterns
- Dynamically creates and removes event patterns based on price limit events received
- Simple, highly-performant filter expressions for event properties in the stock tick event such as symbol and price

16.11. MatchMaker

In the MatchMaker example every mobile user has an X and Y location, a set of properties (gender, hair color, age range) and a set of preferences (one for each property) to match. The task of the event patterns created by this example is to detect mobile users that are within proximity given a certain range, and for which the properties match preferences.

The event class representing mobile users is `com.espertech.esper.example.matchmaker.event.MobileUserBean`. The `com.espertech.esper.example.matchmaker.monitor.MatchMakingMonitor` class contains the patterns for detecting matches.

Summary:

- Dynamically creates and removes event patterns based on mobile user events received
- Uses range matching for X and Y properties of mobile user events

16.12. Named Window Query

This example handles very minimal temperature sensor events which are represented by `IDictionary`. It creates a named window and fills it with a large number of events. It then executes a large number of pre-defined queries via on-select as well as performs a large number of fire-and-forget queries against the named window, and reports execution times.

16.13. Quality of Service

This example develops some code for measuring quality-of-service levels such as for a service-level agreement (SLA). A SLA is a contract between 2 parties that defines service constraints such as maximum latency for service operations or error rates.

The example measures and monitors operation latency and error counts per customer and operation. When one of our operations oversteps these constraints, we want to be alerted right away. Additionally, we would like to have some monitoring in place that checks the health of our service and provides some information on how the operations are used.

Some of the constraints we need to check are:

- That the latency (time to finish) of some of the operations is always less than X seconds.
- That the latency average is always less than Y seconds over Z operation invocations.

The `com.espertech.esper.example.qos_sla.events.OperationMeasurement` event class with its latency and status properties is the main event used for the SLA analysis. The other event `LatencyLimit` serves to set latency limits on the fly.

The `com.espertech.esper.example.qos_sla.monitor.AverageLatencyMonitor` creates an EPL statement that computes latency statistics per customer and operation for the last 100 events. The `DynaLatencySpikeMonitor` uses an event pattern to listen to spikes in latency with dynamically set limits. The `ErrorRateMonitor` uses the timer 'at' operator in an event pattern that wakes up periodically and polls the error rate within the last 10 minutes. The `ServiceHealthMonitor` simply alerts when 3 errors occur, and the `SpikeAndErrorMonitor` alerts when a fixed latency is overstepped or an error status is reported.

Summary:

- This example combines event patterns with EPL statements for event stream analysis.
- Shows the use of the timer 'at' operator and followed-by operator `->` in event patterns.
- Outlines basic EPL statements.
- Shows how to pull data out of EPL statements rather than subscribing to events a statement publishes.

Chapter 17. Performance

Esper has been highly optimized to handle very high throughput streams with very little latency between event receipt and output result posting. It is also possible to use Esper on a soft-real-time or hard-real-time JVM to maximize predictability even further.

This section describes performance best practices and explains how to assess Esper performance by using our provided performance kit.

17.1. Performance Results

For a complete understanding of those results, consult the next sections.

```
Esper exceeds over 500 000 event/s on a dual CPU 2GHz Intel based hardware,  
with engine latency below 3 microseconds average (below 10us with more than  
99% predictability) on a VWAP benchmark with 1000 statements registered in the system  
- this tops at 70 Mbit/s at 85% CPU usage.
```

```
Esper also demonstrates linear scalability from 100 000 to 500 000 event/s on this  
hardware, with consistent results accross different statements.
```

```
Other tests demonstrate equivalent performance results  
(straight through processing, match all, match none, no statement registered,  
VWAP with time based window or length based windows).
```

```
Tests on a laptop demonstrated about 5x time less performance - that is  
between 70 000 event/s and 200 000 event/s - which still gives room for easy  
testing on small configuration.
```

17.2. Performance Tips

17.2.1. Understand how to tune your application

Esper runs within application domains for a CLR instance. Some performance tuning techniques are general purpose for tuning CLR performance, however there have been specific accomodations made to improve Esper performance. Statements with time-based or length-based data windows can consume large amounts of memory as their size or length can be large.

For time-based data windows, one needs to be aware that the memory consumed depends on the actual event stream input throughput. Event pattern instances also consume memory, especially when using the "every" keyword in patterns to repeat pattern sub-expressions - which again will depend on the actual event stream input throughput.

Esper makes great use of thread-local data. This technique ensures that the application can interoperate on two threads independently without the need for a lock. Esper (CLR) provides specific techniques for selecting the thread-local implementation. The default CLR ships with thread local storage (TLS) as part of the core model and Esper leverages that through the SystemThreadLocalFactory. However, performance tests revealed that it's performance was less than optimal and a custom implementation was written for Esper. This implementation FastThreadLocalFactory is significantly faster than the TLS implementation. FastThreadLocal is the default implementation for Esper, but may be overridden through configuration.

Esper makes use of locks where necessary to ensure that threads do not collide when using critical resources.

Two types of locks are utilized, standard exclusion locks and reader-writer locks. Through configuration, implementors can choose the specific type of lock that should be utilized. When choosing locks, please ensure that are choosing the right locks. If applications have a significant number of hooks that hold locks open for extended periods of time (Esper by default will not do this), then you should use a standard monitor. However, if your locks are very tight (default), then you will want to ensure that you are using the spin lock (exclusion) or slim lock (reader-writer).

17.2.2. Compare Esper to other solutions

If you compare Esper performance to the performance of another solution, you need to ensure that your statements have truly equivalent semantics. This is because between different vendors the event processing language can seem fairly similar whoever may, for all similarities, produce different results.

For example some vendor solution mandates the use of "bounded streams". The next statement shows one vendor's event processing syntax:

```
// Other (name omitted) vendor solution statement:
select * from (select * from Market where ticker = 'GOOG') retain 1 event
// The above is NOT an Esper statement
```

The semantically equivalent statement in Esper is:

```
// Esper statement with the same semantics:
select * from MarketData(ticker='$').std:lastevent()
// ... or ...
select * from MarketData(ticker='$').win:length(1)
```

As an example, a NOT semantically equivalent statement in Esper is:

```
// Esper statement that DOES ***NOT*** HAVE the same semantics
// No length window was used
select * from MarketData(ticker='$')
```

17.2.3. Input and Output Bottlenecks

Your application receives output events from Esper statements through the `UpdateEventHandler` or via the strongly-typed subscriber object. Such output events are delivered by the application or timer thread(s) that sends an input event into the engine instance.

The processing of output events that your listener or subscriber performs temporarily blocks the thread until the processing completes, and may thus reduce throughput. It can therefore be beneficial for your application to process output events asynchronously and not block the Esper engine while an output event is being processed by your listener, especially if your listener code performs blocking IO operations.

For example, your application may want to send output events to a JMS destination or write output event data to a relational database. For optimal throughput, consider performing such blocking operations in a separate thread.

Additionally, when reading input events from a store or network in a performance test, you may find that Esper processes events faster than you are able to feed events into Esper. In such case you may want to consider an in-memory driver for use in performance testing. Also consider decoupling your read operation from the event processing operation (`sendEvent` method) by having multiple readers or by pre-fetching your data from the store.

17.2.4. Advanced Threading

Esper provides the configuration option to use engine-level queues and threadpools for inbound, outbound and internal executions. See Section 12.6.1, “Advanced Threading” for more information.

We recommend the inbound threading if your application does not already employ threading.

We recommend the outbound threading if your listeners are blocking. For outbound threading also see the section below on tuning and disabling listener delivery guarantees.

If enabling advanced threading options keep in mind that the engine will maintain a queue and thread pool. There is additional overhead associated with entering work units into the queue, maintaining the queue and the hand-off between threads. It is not necessarily true that your application will perform better with any of the advanced threading options.

17.2.5. Select the underlying event rather than individual fields

By selecting the underlying event in the select-clause we can reduce load on the engine, since the engine does not need to generate a new output event for each input event.

For example, the following statement returns the underlying event to update listeners:

```
// Better performance
select * from RFIDEvent
```

In comparison, the next statement selects individual properties. This statement requires the engine to generate an output event that contains exactly the required properties:

```
// Less good performance
select assetId, zone, xlocation, ylocation from RFIDEvent
```

17.2.6. Prefer stream-level filtering over post-data-window filtering

Esper stream-level filtering is very well optimized, while filtering via the where-clause post any data windows is not optimized. In very simple statements that don't have data windows this distinction can make a performance difference.

Consider the example below, which performs stream-level filtering:

```
// Better performance : stream-level filtering
select * from MarketData(ticker = 'GOOG')
```

The example below is the equivalent (same semantics) statement and performs post-data-window filtering without a data window. The engine does not optimize statements that filter in the where-clause for the reason that data window views are generally present.

```
// Less good performance : post-data-window filtering
select * from Market where ticker = 'GOOG'
```

Thus this optimization technique applies to statements without any data window.

When a data window is used, the semantics change. Let's look at an example to better understand the difference: In the next statement only GOOG market events enter the length window:

```
select avg(price) from MarketData(ticker = 'GOOG').win:length(100)
```

The above statement computes the average price of GOOG market data events for the last 100 GOOG market data events.

Compare the filter position to a filter in the where clause. The following statement is NOT equivalent as all events enter the data window (not just GOOG events):

```
select avg(price) from Market.win:length(100) where ticker = 'GOOG'
```

The statement above computes the average price of all market data events for the last 100 market data events, and outputs results only for GOOG.

17.2.7. Reduce the use of arithmetic in expressions

Esper does not yet attempt to pre-evaluate arithmetic expressions that produce constant results.

Therefore, a filter expression as below is optimized:

```
// Better performance : no arithmetic
select * from MarketData(price>40)
```

While the engine cannot currently optimize this expression:

```
// Less good performance : with arithmetic
select * from MarketData(price+10>50)
```

17.2.8. Remove Unnecessary Constructs

If your statement uses `order by` to order output events, consider removing `order by` unless your application does indeed require the events it receives to be ordered.

If your statement specifies `group by` but does not use aggregation functions, consider removing `group by`.

17.2.9. End Pattern Sub-Expressions

In patterns, the `every` keyword in conjunction with followed by (`->`) starts a new sub-expression per match.

For example, the following pattern starts a sub-expression looking for a B event for every A event that arrives.

```
every A -> B
```

Determine under what conditions a subexpression should end so the engine can stop looking for a B event. Here are a few generic examples:

```
every A -> (B and not C)
every A -> B where timer:within(1 sec)
```

17.2.10. Consider using EventPropertyGetter for fast access to event properties

The EventPropertyGetter interface is useful for obtaining an event property value without property name table

lookup given an EventBean instance that is of the same event type that the property getter was obtained from.

When compiling a statement, the EPStatement instance lets us know the EventType via the `getEventType()` method. From the EventType we can obtain EventPropertyGetter instances for named event properties.

To demonstrate, consider the following simple statement:

```
select symbol, avg(price) from Market group by symbol
```

After compiling the statement, obtain the EventType. The event handler can use the type to obtain fast getters for property values of events for the same type:

```
var stmt = epService.EPAdministrator.CreateEPL(stmtText);
var stmtEventType = stmt.EventType;
var symbolGetter = stmtEventType.GetGetter("symbol");
var avgPriceGetter = stmtEventType.GetGetter("avg(price)");
var eventHandler = delegate(Object sender, UpdateEventArgs e) {
    var symbol = (String) symbolGetter.get(e.NewEvents[0]);
    var volume = (long) volumeGetter.get(e.NewEvents[0]);
    // some more logic here
};

stmt.Events += eventHandler;
```

17.2.11. Consider casting the underlying event

When an application requires the value of most or all event properties, it can often be best to simply select the underlying event via wildcard and cast the received events.

Let's look at the sample statement:

```
select * from MarketData(symbol regexp 'E[a-z]')
```

An update listener to the statement may want to cast the received events to the expected underlying event class:

```
public void HandleEvents(Object sender, UpdateEventArgs e) {
    MarketData md = (MarketData) e.NewEvents[0].Underlying;
    // some more logic here
}
```

17.2.12. Turn off logging

Since Esper 1.10, even if you don't have a log4j configuration file in place, Esper will make sure to minimize execution path logging overhead. For prior versions, and to reduce logging overhead overall, we recommend the "WARN" log level or the "INFO" log level.

Please see the log4j configuration file in "etc/infoonly_log4j.xml" for example log4j settings.

17.2.13. Disable view sharing

By default, Esper compares streams and views in use with existing statement's streams and views, and then re-uses views to efficiently share resources between statements. The benefit is reduced resources usage, however the potential cost is that in multithreaded applications a shared view may mean excessive locking of multiple processing threads.

Consider disabling view sharing for better threading performance if your application overall uses fewer statements and statements have very similar streams, filters and views.

View sharing can be disabled via XML configuration or API, and the next code snippet shows how, using the API:

```
Configuration config = new Configuration();
config.EngineDefaults.ViewResources.IsShareViews = false;
```

17.2.14. Tune or disable delivery order guarantees

If your application is not a multithreaded application, or your application is not sensitive to the order of delivery of result events to your application listeners, then consider disabling the delivery order guarantees the engine makes towards ordered delivery of results to listeners:

```
Configuration config = new Configuration();
config.EngineDefaults.Threading.IsListenerDispatchPreserveOrder = false;
```

If your application is not a multithreaded application, or your application uses the `insert into` clause to make results of one statement available for further consuming statements but does not require ordered delivery of results from producing statements to consuming statements, you may disable delivery order guarantees between statements:

```
Configuration config = new Configuration();
config.EngineDefaults.Threading.IsInsertIntoDispatchPreserveOrder = false;
```

Additional configuration options are available and described in the configuration section that specify timeout values and spin or thread context switching.

Esper logging will log the following informational message when guaranteed delivery order to listeners is enabled and spin lock times exceed the default or configured timeout : `Spin wait timeout exceeded in listener dispatch`. The respective message for delivery from `insert into` statements to consuming statements is `Spin wait timeout exceeded in insert-into dispatch`.

If your application sees messages that spin lock times are exceeded, your application has several options: First, disabling preserve order is an option. Second, ensure your listener does not perform (long-running) blocking operations before returning, for example by performing output event processing in a separate thread. Third, change the timeout value to a larger number to block longer without logging the message.

17.2.15. Use a Subscriber Object to Receive Events

The subscriber object is a technique to receive result data that has performance advantages over the `UpdateEventHandler` interface. Please refer to Section 12.3.3, “Setting a Subscriber Object”.

17.2.16. High-Arrival-Rate Streams and Single Statements

A statement is associated with certain statement state that consists of current aggregation values, partial pattern matches, data windows or other view state depending on whether your statement uses such constructs. When an engine receives events it updates statement state under locking such that statement state remains consistent under concurrent multi-threaded access.

For high-volume streams, the locking required to protected statement state may slow down or introduce block-

ing for very high arrival rates of events that apply to the very same statement and its state. You may want to consider splitting a single statement into multiple statements that each look for a certain subset of the high-arrival-rate stream. There is very little cost in terms of memory or CPU resources per statement, the engine can handle larger number of statements usually as efficiently as single statements.

For example, consider the following statement:

```
// less effective in a highly threaded environment
select venue, ccyPair, side, sum(qty)
from CumulativePrice.std:groupwin(ccyPair,side) as w
where w.side='O'
```

The engine will protect state of a statements by a lock on the statement level, as discussed in the API section. In highly threaded applications threads may block on the single statement.

Consider creating a statement per currency instead, for example:

```
select venue, ccyPair, side, sum(qty) from CumulativePrice(side='O', ccyPair='USD')
select venue, ccyPair, side, sum(qty) from CumulativePrice(side='O', ccyPair='EURO')
```

17.2.17. Joins And Where-clause And Data Windows

When joining streams the engine builds a product of the joined data windows based on the `where` clause. It analyzes the `where` clause at time of statement compilation and builds the appropriate indexes and query strategy. Avoid using expressions in the join `where` clause that require evaluation, such as user-defined functions or arithmetic expressions.

When joining streams and not providing a `where` clause, consider using the `std:lastevent` data window to join only the last event of each stream.

The sample query below can produce up to 5,000 rows when both data windows are filled and an event arrives for either stream:

```
// Avoid joins between streams with data windows without where-clause
select * from StreamA.win:length(100), StreamB.win:length(50)
```

Consider using a subquery, consider using separate statements with `insert-into` and consider providing a `where` clause to limit the product of rows.

Below examples show different approaches, assuming that an `MyEvent` is defined with the properties `symbol` and `value`:

```
// Replace the following statement as it may not perform well
select a.symbol, avg(a.value), avg(b.value)
from MyEvent.win:length(100) a, MyEvent.win:length(50) b

// Fast version, first alternative
select a.symbol, avg(a.value), avg(b.value)
from MyEvent.win:length(100) a, MyEvent.win:length(50) b
where a.symbol = b.symbol

// Fast version, second alternative
select
  (select avg(value) from MyEvent.win:length(100)) as avgA,
  (select avg(value) from MyEvent.win:length(50)) as avgB,
  a.symbol
from MyEvent.std:lastevent() a, MyEvent.std:lastevent() b

// Fast version, third alternative
```

```
insert into StreamAvgA select symbol, avg(value) as avgA from MyEvent.win:length(100)
insert into StreamAvgB select symbol, avg(value) as avgB from MyEvent.win:length(50)
select a.symbol, avgA, avgB from StreamAvgA.std:lastevent() a, StreamAvgB.std:lastevent() b
```

17.2.18. Patterns and Pattern Sub-Expression Instances

The `every` and `repeat` operators in patterns control the number of sub-expressions that are active. Each sub-expression can consume memory as it may retain, depending on the use of tags in the pattern, the matching events. A large number of active sub-expressions can reduce performance or lead to out-of-memory errors.

During the design of the pattern statement consider the use of `timer:within` to reduce the amount of time a sub-expression lives, or consider the `not` operator to end a sub-expression.

The examples herein assume an `AEvent` and a `BEvent` event type that have an `id` property that may correlate between arriving events of the two event types.

In the following sample pattern the engine starts, for each arriving `AEvent`, a new pattern sub-expression looking for a matching `BEvent`. Since the `AEvent` is tagged with `a` the engine retains each `AEvent` until a match is found for delivery to listeners or subscribers:

```
every a=AEvent -> b=BEvent(b.id = a.id)
```

One way to end a sub-expression is to attach a time how long it may be active.

The next statement ends sub-expressions looking for a matching `BEvent` 10 seconds after arrival of the `AEvent` event that started the sub-expression:

```
every a=AEvent -> (b=BEvent(b.id = a.id) where timer:within(10 sec))
```

A second way to end a sub-expression is to use the `not` operator. You can use the `not` operator together with the `and` operator to end a sub-expression when a certain event arrives.

The next statement ends sub-expressions looking for a matching `BEvent` when, in the order of arrival, the next `BEvent` that arrives after the `AEvent` event that started the sub-expression does not match the `id` of the `AEvent`:

```
every a=AEvent -> (b=BEvent(b.id = a.id) and not BEvent(b.id != a.id))
```

The `every-distinct` operator can be used to keep one sub-expression alive per one or more keys. The next pattern demonstrates an alternative to `every-distinct`. It ends sub-expressions looking for a matching `BEvent` when an `AEvent` arrives that matches the `id` of the `AEvent` that started the sub-expression:

```
every a=AEvent -> (b=BEvent(b.id = a.id) and not AEvent(b.id = a.id))
```

17.2.19. The Keep-All Data Window

The `std:keepall` data window is a data window that retains all arriving events. The data window can be useful during the development phase and to implement a custom expiry policy using `on-delete` and named windows. Care should be taken to timely remove from the keep-all data window however. Use `on-select` or `on-demand` queries to count the number of rows currently held by a named window with keep-all expiry policy.

17.2.20. Statement Design for Reduced Memory Consumption

This section describes common sources of out-of-memory problems.

If using the `keep-all` data window please consider the information above. If using pattern statements please consider pattern sub-expression lifetime as discussed prior to this section.

When using the `group-by` clause or `std:groupwin` grouped data windows please consider the hints as described below.

The `std:unique` unique data window can also be a source for error. If your uniqueness criteria include a field which is never unique the memory use of the data window can grow, unless your application deletes events.

When using the `every-distinct` pattern construct parameterized by distinct value expressions that generate an unlimited number of distinct values, consider specifying a time period as part of the parameters to indicate to the pattern engine how long a distinct value should be considered.

In a match-recognize pattern consider limiting the number of optional events if optional events are part of the data reported in the `measures` clause. Also when using the `partition` clause, if your partitioning criteria include a field which is never unique the memory use of the match-recognize pattern engine can grow.

A further source of memory use is when your application creates new statements but fails to destroy created statements when they are no longer needed.

In your application design you may also want to be conscious when the application listener or subscriber objects retain output data.

17.2.21. Performance, JVM, OS and hardware

Performance will also depend on your JVM (Sun HotSpot, BEA JRockit, IBM J9), your operating system and your hardware. A JVM performance index such as specJBB at spec.org [<http://www.spec.org>] can be used. For memory intensive statement, you may want to consider 64bit architecture that can address more than 2GB or 3GB of memory, although a 64bit JVM usually comes with a slow performance penalty due to more complex pointer address management.

The choice of JVM, OS and hardware depends on a number of factors and therefore a definite suggestion is hard to make. The choice depends on the number of statements, and number of threads. A larger number of threads would benefit of more CPU and cores. If you have very low latency requirements, you should consider getting more GHz per core, and possibly soft real-time JVM to enforce GC determinism at the JVM level, or even consider dedicated hardware such as Azul. If your statements utilize large data windows, more RAM and heap space will be utilized hence you should clearly plan and account for that and possibly consider 64bit architectures or consider EsperHA [<http://www.espertech.com/products/>].

The number and type of statements is a factor that cannot be generically accounted for. The benchmark kit can help test out some requirements and establish baselines, and for more complex use cases a simulation or proof of concept would certainly works best. EsperTech' experts [<http://www.espertech.com/support/services.php>] can be available to help write interfaces in a consulting relationship.

17.2.22. Consider using Hints

The `@Hint` annotation provides a single keyword or a comma-separated list of keywords that provide instructions to the engine towards statement execution that affect runtime performance and memory-use of statements. Also see Section 4.2.6.8, “`@Hint`”.

The hint for use with `group by` to specify how state for groups is reclaimed is described in Section 4.6.2.1,

“Hints Pertaining to Group-By” and Section 11.2.2, “Grouped Data Window (std:groupwin)”.

The hint for use with `group by` to specify aggregation state reclaim for unbound streams and timestamp groups is described in Section 4.6.2.1, “Hints Pertaining to Group-By”.

The hint for use with `match_recognize` to specify iterate-only is described in Section 6.4.6, “Eliminating Duplicate Matches”.

To tune subquery performance when your subquery selects from a named window, consider the hints discussed in Section 4.11.7, “Hints Related to Subqueries”.

17.2.23. Optimizing Stream Filter Expressions

Assume your EPL statement invokes a static method in the stream filter as the below statement shows as an example:

```
select * from MyEvent(MyHelperLibrary.filter(field1, field2, field3, field4*field5))
```

As a result of starting above statement, the engine must evaluate each `MyEvent` event invoking the `MyHelperLibrary.filter` method and passing certain event properties. The same applies to pattern filters that specify functions to evaluate.

If possible, consider moving some of the checking performed by the function back into the filter or consider splitting the function into a two parts separated by `and` conjunction. In general for all expressions, the engine evaluates expressions left of the `and` first and can skip evaluation of the further expressions in the conjunction in the case when the first expression returns false. In addition the engine can build a reverse index for fields provided in stream or pattern filters.

For example, the below statement could be faster to evaluate:

```
select * from MyEvent(field1="value" and MyHelperLibrary.filter(field1, field2, field3, field4*field5))
```

17.3. Using the performance kit

17.3.1. How to use the performance kit

The benchmark application is basically an Esper event server build with Esper that listens to remote clients over TCP. Remote clients send `MarketData(ticker, price, volume)` streams to the event server. The Esper event server is started with 1000 statements of one single kind (unless otherwise written), with one statement per ticker symbol, unless the statement kind does not depend on the symbol. The statement prototype is provided along the results with a '\$' instead of the actual ticker symbol value. The Esper event server is entirely multithreaded and can leverage the full power of 32bit or 64bit underlying hardware multi-processor multi-core architecture.

The kit also prints out when starting up the event size and the theoretical maximal throughput you can get on a 100 Mbit/s and 1 Gbit/s network. Keep in mind a 100 Mbit/s network will be overloaded at about 400 000 event/s when using our kit despite the small size of events.

Results are posted on our Wiki page at <http://docs.codehaus.org/display/ESPER/Esper+performance>. Reported results do not represent best ever obtained results. Reported results may help you better compare Esper to other solutions (for latency, throughput and CPU utilization) and also assess your target hardware and JVMs.

The Esper event server, client and statement prototypes are provided in the source repository `esper/`

trunk/examples/benchmark/. Refer to <http://xircles.codehaus.org/projects/esper/repo> for source access.

A built is provided for convenience (without sources) as an attachment to the Wiki page at <http://docs.codehaus.org/pages/viewpageattachments.action?pageId=8356191>. It contains Ant script to start client, server in simulation mode and server. For real measurement we advise to start from a shell script (because Ant is pipelining stdout/stderr when you invoke a JVM from Ant - which is costly). Sample scripts are provided for you to edit and customize.

If you use the kit you should:

1. Choose the statement you want to benchmark, add it to `etc/statements.properties` under your own KEY and use the `-mode KEY` when you start the Esper event server.
2. Prepare your `runServer.sh/runServer.cmd` and `runClient.sh/runclient.cmd` scripts. You'll need to drop required jar libraries in `lib/`, make sure the classpath is configured in those script to include `build` and `etc`. The required libraries are Esper (any compatible version, we have tested started with Esper 1.7.0) and its dependencies as in the sample below (with Esper 2.1):

```
# classpath on Unix/Linux (on one single line)
etc:build:lib/esper-4.1.0.jar:lib/commons-logging-1.1.1.jar:lib/cglib-nodep-2.2.jar
:lib/antlr-runtime-3.2.jar:lib/log4j-1.2.16.jar
@rem classpath on Windows (on one single line)
etc:build;lib\esper-4.1.0.jar;lib\commons-logging-1.1.1.jar;lib\cglib-nodep-2.2.jar
:lib\antlr-runtime-3.2.jar;lib\log4j-1.2.16.jar
```

Note that `./etc` and `./build` have to be in the classpath. At that stage you should also start to set min and max JVM heap. A good start is 1GB as in `-Xms1g -Xmx1g`

3. Write the statement you want to benchmark given that client will send a stream `MarketData(String ticker, int volume, double price)`, add it to `etc/statements.properties` under your own KEY and use the `-mode KEY` when you start the Esper event server. Use '\$' in the statement to create a prototype. For every symbol, a statement will get registered with all '\$' replaced by the actual symbol value (f.e. 'GOOG')
4. Ensure client and server are using the same `-Desper.benchmark.symbol=1000` value. This sets the number of symbol to use (thus may set the number of statement if you are using a statement prototype, and governs how `MarketData` event are represented over the network. Basically all events will have the same size over the network to ensure predictability and will be ranging between `S0AA` and `S999A` if you use 1000 as a value here (prefix with S and padded with A up to a fixed length string. Volume and price attributes will be randomized.
5. By default the benchmark registers a subscriber to the statement(s). Use `-Desper.benchmark.ul` to use an `UpdateEventHandler` instead. Note that the subscriber contains suitable `Update(..)` methods for the default proposed statement in the `etc/statements.properties` file but might not be suitable if you change statements due to the strong binding with statement results. Refer to Section 12.3.2, "Receiving Statement Results".
6. Establish a performance baseline in simulation mode (without clients). Use the `-rate 1x5000` option to simulate one client (one thread) sending 5000 evt/s. You can ramp up both the number of client simulated thread and their emission rate to maximize CPU utilization. The right number should mimic the client emission rate you will use in the client/server benchmark and should thus be consistent with what your client machine and network will be able to send. On small hardware, having a lot of thread with slow rate will not help getting high throughput in this simulation mode.
7. Do performance runs with client/server mode. Remove the `-rate NxM` option from the `runServer` script or Ant task. Start the server with `-help` to display the possible server options (listen port, statistics, fan out options etc). On the remote machine, start one or more client. Use `-help` to display the possible client op-

tions (remote port, host, emission rate). The client will output the actual number of event it is sending to the server. If the server gets overloaded (or if you turned on `-queue` options on the server) the client will likely not be able to reach its target rate.

Usually you will get better performance by using server side `-queue -1` option so as to have each client connection handled by a single thread pipeline. If you change to 0 or more, there will be intermediate structures to pass the event stream in an asynchronous fashion. This will increase context switching, although if you are using many clients, or are using the `-sleep xxx` (xxx in milliseconds) to simulate a listener delay you may get better performance.

The most important server side option is `-stat xxx` (xxx in seconds) to print out throughput and latency statistics aggregated over the last xxx seconds (and reset every time). It will produce both internal Esper latency (in nanosecond) and also end to end latency (in millisecond, including network time). If you are measuring end to end latency you should make sure your server and client machine(s) are having the same time with f.e. ntpd with a good enough precision. The stat format is like:

```
---Stats - engine (unit: ns)
Avg: 2528 #4101107
  0 <      5000:  97.01%  97.01% #3978672
    5000 <   10000:   2.60%  99.62% #106669
   10000 <  15000:   0.35%  99.97% #14337
   15000 <  20000:   0.02%  99.99% #971
   20000 <  25000:   0.00%  99.99% #177
   25000 <  50000:   0.00% 100.00% #89
   50000 < 100000:   0.00% 100.00% #41
  100000 < 500000:   0.00% 100.00% #120
 500000 < 1000000:   0.00% 100.00% #2
1000000 < 2500000:   0.00% 100.00% #7
2500000 < 5000000:   0.00% 100.00% #5
5000000 <   more:   0.00% 100.00% #18
---Stats - endToEnd (unit: ms)
Avg: -2704829444341073400 #4101609
  0 <      1:  75.01%  75.01% #3076609
  1 <      5:   0.00%  75.01% #0
  5 <     10:   0.00%  75.01% #0
 10 <     50:   0.00%  75.01% #0
 50 <    100:   0.00%  75.01% #0
100 <    250:   0.00%  75.01% #0
250 <    500:   0.00%  75.01% #0
500 <   1000:   0.00%  75.01% #0
1000 <   more:  24.99% 100.00% #1025000
Throughput 412503 (active 0 pending 0 cnx 4)
```

This one reads as:

"Throughput is 412 503 event/s with 4 client connected. No `-queue` options was used thus no event is pending at the time the statistics are printed. Esper latency average is at 2528 ns (that is 2.5 us) for 4 101 107 events (which means we have 10 seconds stats here). Less than 10us latency was achieved for 106 669 events that is 99.62%. Latency between 5us and 10us was achieved for those 2.60% of all the events in the interval."

"End to end latency was ... in this case likely due to client clock difference we ended up with unusable end to end statistics."

Consider the second output paragraph on end-to-end latency:

```
---Stats - endToEnd (unit: ms)
Avg: 15 #863396
  0 <      1:   0.75%   0.75% #6434
  1 <      5:   0.99%   1.74% #8552
  5 <     10:   2.12%   3.85% #18269
 10 <     50:  91.27%  95.13% #788062
 50 <    100:   0.10%  95.22% #827
100 <    250:   4.36%  99.58% #37634
```

250 <	500:	0.42%	100.00%	#3618
500 <	1000:	0.00%	100.00%	#0
1000 <	more:	0.00%	100.00%	#0

This would read:

"End to end latency average is at 15 milliseconds for the 863 396 events considered for this statistic report. 95.13% ie 788 062 events were handled (end to end) below 50ms, and 91.27% were handled between 10ms and 50ms."

17.3.2. How we use the performance kit

We use the performance kit to track performance progress across Esper versions, as well as to implement optimizations. You can track our work on the Wiki at <http://docs.codehaus.org/display/ESPER/Home>

Chapter 18. References

18.1. Reference List

- Luckham, David. 2002. *The Power of Events*. Addison-Wesley.
- The Stanford Rapide (TM) Project. <http://pavg.stanford.edu/rapide>.
- Arasu, Arvind, et.al.. 2004. Linear Road: A Stream Data Management Benchmark, Stanford University http://www.cs.brown.edu/research/aurora/Linear_Road_Benchmark_Homepage.html.

Appendix A. Output Reference and Samples

This section specifies the output of a subset of EPL continuous queries, for two purposes: First, to help application developers understand streaming engine output in response to incoming events and in response to time passing. Second, to document and standardize output for EPL queries in a testable and trackable fashion.

The section focuses on a subset of features, namely the time window, aggregation, grouping, and output rate limiting. The section does not currently provide examples for many of the other language features, thus there is no example for other data windows (the time window is used here), joins, sub-selects or named windows etc.

Rather than just describe syntax and output, this section provides detailed examples for each of the types of queries presented. The input for each type of query is always the same set of events, and the same timing. Each event has three properties: symbol, volume and price. The property types are string, long and double, respectively.

The chapters are organized by the type of query: The presence or absence of aggregation functions, as well as the presence or absence of a `group by` clause change statement output as described in Section 3.7.2, “Output for Aggregation and Group-By”.

You will notice that some queries utilize the `order by` clause for sorting output. The reason is that when multiple output rows are produced at once, the output can be easier to read if it is sorted.

With output rate limiting, the engine invokes your listener even if there are no results to indicate when the output condition has been reached. Such is indicated as `(empty result)` in the output result columns.

The output columns show both insert and remove stream events. Insert stream events are delivered as an array of `EventBean` instances to listeners in the `newData` parameter, while remove stream events are delivered to the `oldData` parameter of listeners. Delivery to observers follows similar rules.

A.1. Introduction and Sample Data

For the purpose of illustration and documentation, the example data set demonstrates input and remove streams based on a time window of a 5.5 second interval. The statement utilizing the time window could look as follows:

```
select symbol, volume, price from MarketData.win:time(5.5 sec)
```

We have picked a time window to demonstrate the output for events entering and leaving a data window with an expiration policy. The time window provides a simple expiration policy based on time: if an event resides in the time window more than 5.5 seconds, the engine expires the event from the time window.

The input events and their timing are below. The table should be read, starting from top, as "The time starts at 0.2 seconds. Event E1 arrives at 0.2 seconds with properties [S1, 100, 25]. At 0.8 second event E2 arrives with properties [S2, 5000, 9.0]" and so on.

Input			
Time	Symbol	Volume	Price
0.2	S1	100	25.0
Event E1 arrives			

```

0.8      S2      5000      9.0      Event E2 arrives
1.0
1.2
1.5      S1      150      24.0      Event E3 arrives
      S3      10000      1.0      Event E4 arrives
2.0
2.1      S1      155      26.0      Event E5 arrives
2.2
2.5
3.0
3.2
3.5      S3      11000      2.0      Event E6 arrives
4.0
4.2
4.3      S1      150      22.0      Event E7 arrives
4.9      S3      11500      3.0      Event E8 arrives
5.0
5.2
5.7                                Event E1 leaves the time window
5.9      S3      10500      1.0      Event E9 arrives
6.0
6.2
6.3                                Event E2 leaves the time window
7.0                                Event E3 and E4 leave the time window
7.2

```

The event data set assumes a time window of 5.5 seconds. Thus at time 5.7 seconds the first arriving event (E1) leaves the time window.

The data set as above shows times between 0.2 seconds and 7.2 seconds. Only a couple of time points have been picked for the table to keep the set of time points constant between statements, and thus make the test data and output easier to understand.

A.2. Output for Un-aggregated and Un-grouped Queries

This chapter provides sample output for queries that do not have aggregation functions and do not have a `group by` clause.

A.2.1. No Output Rate Limiting

Without an `output` clause, the engine dispatches to listeners as soon as events arrive, or as soon as time passes such that events leave data windows.

The statement for this sample reads:

```
select irstream symbol, volume, price from MarketData.win:time(5.5 sec)
```

The output is as follows:

Input			Output	
			Insert Stream	Remove Stream
Time	Symbol	Volume	Price	
0.2				

	IBM	100	25.0	Event E1 arrives	[IBM, 100, 25.0]
0.8					
	MSFT	5000	9.0	Event E2 arrives	[MSFT, 5000, 9.0]
1.0					
1.2					
1.5					
	IBM	150	24.0	Event E3 arrives	[IBM, 150, 24.0]
	YAH	10000	1.0	Event E4 arrives	[YAH, 10000, 1.0]
2.0					
2.1					
	IBM	155	26.0	Event E5 arrives	[IBM, 155, 26.0]
2.2					
2.5					
3.0					
3.2					
3.5					
	YAH	11000	2.0	Event E6 arrives	[YAH, 11000, 2.0]
4.0					
4.2					
4.3					
	IBM	150	22.0	Event E7 arrives	[IBM, 150, 22.0]
4.9					
	YAH	11500	3.0	Event E8 arrives	[YAH, 11500, 3.0]
5.0					
5.2					
5.7				Event E1 leaves the time window	[IBM, 100, 25.0]
5.9					
	YAH	10500	1.0	Event E9 arrives	[YAH, 10500, 1.0]
6.0					
6.2					
6.3				Event E2 leaves the time window	[MSFT, 5000, 9.0]
7.0				Event E3 and E4 leave the time window	[IBM, 150, 24.0] [YAH, 10000, 1.0]
7.2					

A.2.2. Output Rate Limiting - Default

With an `output` clause, the engine dispatches to listeners when the output condition occurs. Here, the output condition is a 1-second time interval. The engine thus outputs every 1 second, starting from the first event, even if there are no new events or no expiring events to output.

The default (no keyword) and the `ALL` keyword result in the same output.

The statement for this sample reads:

```
select istream symbol, volume, price from MarketData.win:time(5.5 sec)
output every 1 seconds
```

The output is as follows:

Input	Output	
	Insert Stream	Remove Stream
-----	-----	-----

Time	Symbol	Volume	Price	
0.2				
	IBM	100	25.0	Event E1 arrives
0.8				
	MSFT	5000	9.0	Event E2 arrives
1.0				
1.2				
				[IBM, 100, 25.0] [MSFT, 5000, 9.0]
1.5				
	IBM	150	24.0	Event E3 arrives
	YAH	10000	1.0	Event E4 arrives
2.0				
2.1				
	IBM	155	26.0	Event E5 arrives
2.2				
				[IBM, 150, 24.0] [YAH, 10000, 1.0] [IBM, 155, 26.0]
2.5				
3.0				
3.2				
				(empty result) (empty result)
3.5				
	YAH	11000	2.0	Event E6 arrives
4.0				
4.2				
				[YAH, 11000, 2.0]
4.3				
	IBM	150	22.0	Event E7 arrives
4.9				
	YAH	11500	3.0	Event E8 arrives
5.0				
5.2				
				[IBM, 150, 22.0] [YAH, 11500, 3.0]
5.7				Event E1 leaves the time window
5.9				
	YAH	10500	1.0	Event E9 arrives
6.0				
6.2				
				[YAH, 10500, 1.0] [IBM, 100, 25.0]
6.3				Event E2 leaves the time window
7.0				Event E3 and E4 leave the time window
7.2				
				[MSFT, 5000, 9.0] [IBM, 150, 24.0] [YAH, 10000, 1.0]

A.2.3. Output Rate Limiting - Last

Using the `LAST` keyword in the `output` clause, the engine dispatches to listeners only the last event of each insert and remove stream.

The statement for this sample reads:

```
select istream symbol, volume, price from MarketData.win:time(5.5 sec)
output last every 1 seconds
```

The output is as follows:

Input				Output	
				Insert Stream	Remove Stream
Time	Symbol	Volume	Price		
0.2					

0.8	IBM	100	25.0	Event E1 arrives		
1.0	MSFT	5000	9.0	Event E2 arrives		
1.2						
					[MSFT, 5000, 9.0]	
1.5	IBM	150	24.0	Event E3 arrives		
2.0	YAH	10000	1.0	Event E4 arrives		
2.1						
2.2	IBM	155	26.0	Event E5 arrives		
					[IBM, 155, 26.0]	
2.5						
3.0						
3.2					(empty result)	(empty result)
3.5						
	YAH	11000	2.0	Event E6 arrives		
4.0						
4.2					[YAH, 11000, 2.0]	
4.3						
	IBM	150	22.0	Event E7 arrives		
4.9						
	YAH	11500	3.0	Event E8 arrives		
5.0						
5.2					[YAH, 11500, 3.0]	
5.7				Event E1 leaves the time window		
5.9						
	YAH	10500	1.0	Event E9 arrives		
6.0						
6.2					[YAH, 10500, 1.0]	[IBM, 100, 25.0]
6.3				Event E2 leaves the time window		
7.0				Event E3 and E4 leave the time window		
7.2						[YAH, 10000, 1.0]

A.2.4. Output Rate Limiting - First

Using the `FIRST` keyword in the `output` clause, the engine dispatches to listeners only the first event of each insert or remove stream, and does not output further events until the output condition is reached.

The statement for this sample reads:

```
select istream symbol, volume, price from MarketData.win:time(5.5 sec)
output first every 1 seconds
```

The output is as follows:

Input				Output	
				Insert Stream	Remove Stream
Time	Symbol	Volume	Price		
0.2					
	IBM	100	25.0	Event E1 arrives	
					[IBM, 100, 25.0]
0.8					
	MSFT	5000	9.0	Event E2 arrives	
1.0					
1.2					
1.5					
	IBM	150	24.0	Event E3 arrives	

```

                                [IBM, 150, 24.0]
2.0    YAH    10000    1.0    Event E4 arrives
2.1
2.2    IBM     155    26.0    Event E5 arrives
2.5
3.0
3.2
                                (empty result)    (empty result)
3.5
                                [YAH, 11000, 2.0]
    YAH    11000    2.0    Event E6 arrives
4.0
4.2
4.3
    IBM     150    22.0    Event E7 arrives
                                [IBM, 150, 22.0]
4.9
    YAH    11500    3.0    Event E8 arrives
5.0
5.2
5.7
                                Event E1 leaves the time window
                                [IBM, 100, 25.0]
5.9
    YAH    10500    1.0    Event E9 arrives
6.0
6.2
6.3
                                Event E2 leaves the time window
                                [MSFT, 5000, 9.0]
7.0
                                Event E3 and E4 leave the time window
7.2

```

A.2.5. Output Rate Limiting - Snapshot

Using the `SNAPSHOT` keyword in the `output` clause, the engine posts data window contents when the output condition is reached.

The statement for this sample reads:

```

select istream symbol, volume, price from MarketData.win:time(5.5 sec)
output snapshot every 1 seconds

```

The output is as follows:

Input				Output	
				Insert Stream	Remove Stream
Time	Symbol	Volume	Price		
0.2					
	IBM	100	25.0	Event E1 arrives	
0.8					
	MSFT	5000	9.0	Event E2 arrives	
1.0					
1.2					
				[IBM, 100, 25.0]	
				[MSFT, 5000, 9.0]	
1.5					
	IBM	150	24.0	Event E3 arrives	
	YAH	10000	1.0	Event E4 arrives	
2.0					
2.1					
	IBM	155	26.0	Event E5 arrives	
2.2					
				[IBM, 100, 25.0]	
				[MSFT, 5000, 9.0]	

```

[IBM, 150, 24.0]
[YAH, 10000, 1.0]
[IBM, 155, 26.0]

2.5
3.0
3.2

[IBM, 100, 25.0]
[MSFT, 5000, 9.0]
[IBM, 150, 24.0]
[YAH, 10000, 1.0]
[IBM, 155, 26.0]

3.5
YAH 11000 2.0 Event E6 arrives
4.0
4.2

[IBM, 100, 25.0]
[MSFT, 5000, 9.0]
[IBM, 150, 24.0]
[YAH, 10000, 1.0]
[IBM, 155, 26.0]
[YAH, 11000, 2.0]

4.3
IBM 150 22.0 Event E7 arrives
4.9
YAH 11500 3.0 Event E8 arrives
5.0
5.2

[IBM, 100, 25.0]
[MSFT, 5000, 9.0]
[IBM, 150, 24.0]
[YAH, 10000, 1.0]
[IBM, 155, 26.0]
[YAH, 11000, 2.0]
[IBM, 150, 22.0]
[YAH, 11500, 3.0]

5.7
5.9 Event E1 leaves the time window
YAH 10500 1.0 Event E9 arrives
6.0
6.2

[MSFT, 5000, 9.0]
[IBM, 150, 24.0]
[YAH, 10000, 1.0]
[IBM, 155, 26.0]
[YAH, 11000, 2.0]
[IBM, 150, 22.0]
[YAH, 11500, 3.0]
[YAH, 10500, 1.0]

6.3 Event E2 leaves the time window
7.0 Event E3 and E4 leave the time window
7.2

[IBM, 155, 26.0]
[YAH, 11000, 2.0]
[IBM, 150, 22.0]
[YAH, 11500, 3.0]
[YAH, 10500, 1.0]

```

A.3. Output for Fully-aggregated and Un-grouped Queries

This chapter provides sample output for queries that have aggregation functions, and that do not have a `group by` clause, and in which all event properties are under aggregation.

A.3.1. No Output Rate Limiting

The statement for this sample reads:

```
select irstream sum(price) from MarketData.win:time(5.5 sec)
```

The output is as follows:

Input					Output	
					Insert Stream	Remove Stream
Time	Symbol	Volume	Price			
0.2	IBM	100	25.0	Event E1 arrives	[25.0]	[null]
0.8	MSFT	5000	9.0	Event E2 arrives	[34.0]	[25.0]
1.0						
1.2						
1.5	IBM	150	24.0	Event E3 arrives	[58.0]	[34.0]
	YAH	10000	1.0	Event E4 arrives	[59.0]	[58.0]
2.0						
2.1	IBM	155	26.0	Event E5 arrives	[85.0]	[59.0]
2.2						
2.5						
3.0						
3.2						
3.5	YAH	11000	2.0	Event E6 arrives	[87.0]	[85.0]
4.0						
4.2						
4.3	IBM	150	22.0	Event E7 arrives	[109.0]	[87.0]
4.9	YAH	11500	3.0	Event E8 arrives	[112.0]	[109.0]
5.0						
5.2						
5.7				Event E1 leaves the time window	[87.0]	[112.0]
5.9	YAH	10500	1.0	Event E9 arrives	[88.0]	[87.0]
6.0						
6.2						
6.3				Event E2 leaves the time window	[79.0]	[88.0]
7.0				Event E3 and E4 leave the time window	[54.0]	[79.0]
7.2						

A.3.2. Output Rate Limiting - Default

Output occurs when the output condition is reached after each 1-second time interval. For each event arriving, the new aggregation value is output as part of the insert stream. As part of the remove stream, the prior aggregation value is output. This is useful for getting a delta-change for each event or group. If there is a `having` clause, the filter expression applies to each row.

Here also the default (no keyword) and the `ALL` keyword result in the same output.

The statement for this sample reads:

```
select irstream sum(price) from MarketData.win:time(5.5 sec)
output every 1 seconds
```

The output is as follows:

Input					Output	
					Insert Stream	Remove Stream
Time	Symbol	Volume	Price			
0.2						
	IBM	100	25.0	Event E1 arrives		
0.8						
	MSFT	5000	9.0	Event E2 arrives		
1.0						
1.2					[25.0]	[null]
					[34.0]	[25.0]
1.5						
	IBM	150	24.0	Event E3 arrives		
	YAH	10000	1.0	Event E4 arrives		
2.0						
2.1						
	IBM	155	26.0	Event E5 arrives		
2.2						
					[58.0]	[34.0]
					[59.0]	[58.0]
					[85.0]	[59.0]
2.5						
3.0						
3.2						
					[85.0]	[85.0]
3.5						
	YAH	11000	2.0	Event E6 arrives		
4.0						
4.2						
					[87.0]	[85.0]
4.3						
	IBM	150	22.0	Event E7 arrives		
4.9						
	YAH	11500	3.0	Event E8 arrives		
5.0						
5.2						
					[109.0]	[87.0]
					[112.0]	[109.0]
5.7				Event E1 leaves the time window		
5.9						
	YAH	10500	1.0	Event E9 arrives		
6.0						
6.2						
					[87.0]	[112.0]
					[88.0]	[87.0]
6.3				Event E2 leaves the time window		
7.0				Event E3 and E4 leave the time window		
7.2						
					[79.0]	[88.0]
					[54.0]	[79.0]

A.3.3. Output Rate Limiting - Last

With the `LAST` keyword, the insert stream carries one event that holds the last aggregation value, and the remove stream carries the prior aggregation value.

The statement for this sample reads:

```
select irstream sum(price) from MarketData.win:time(5.5 sec)
output last every 1 seconds
```

The output is as follows:

Input				Output	
				Insert Stream	Remove Stream
Time	Symbol	Volume	Price		
0.2					
	IBM	100	25.0	Event E1 arrives	
0.8	MSFT	5000	9.0	Event E2 arrives	
1.0					
1.2				[34.0]	[null]
1.5					
	IBM	150	24.0	Event E3 arrives	
	YAH	10000	1.0	Event E4 arrives	
2.0					
2.1					
	IBM	155	26.0	Event E5 arrives	
2.2				[85.0]	[34.0]
2.5					
3.0					
3.2				[85.0]	[85.0]
3.5					
	YAH	11000	2.0	Event E6 arrives	
4.0					
4.2				[87.0]	[85.0]
4.3					
	IBM	150	22.0	Event E7 arrives	
4.9					
	YAH	11500	3.0	Event E8 arrives	
5.0					
5.2				[112.0]	[87.0]
5.7				Event E1 leaves the time window	
5.9					
	YAH	10500	1.0	Event E9 arrives	
6.0					
6.2				[88.0]	[112.0]
6.3				Event E2 leaves the time window	
7.0				Event E3 and E4 leave the time window	
7.2				[54.0]	[88.0]

A.3.4. Output Rate Limiting - First

The statement for this sample reads:

```
select istream sum(price) from MarketData.win:time(5.5 sec)
output first every 1 seconds
```

The output is as follows:

Input				Output	
				Insert Stream	Remove Stream
Time	Symbol	Volume	Price		
0.2					
	IBM	100	25.0	Event E1 arrives	
				[25.0]	[null]
0.8	MSFT	5000	9.0	Event E2 arrives	

```

1.0
1.2
1.5      IBM      150      24.0  Event E3 arrives      [58.0]      [34.0]
      YAH      10000      1.0  Event E4 arrives
2.0
2.1      IBM      155      26.0  Event E5 arrives
2.2
2.5
3.0
3.2                                [85.0]      [85.0]
3.5      YAH      11000      2.0  Event E6 arrives      [87.0]      [85.0]
4.0
4.2
4.3      IBM      150      22.0  Event E7 arrives      [109.0]     [87.0]
4.9      YAH      11500      3.0  Event E8 arrives
5.0
5.2
5.7                                Event E1 leaves the time window
                                [87.0]      [112.0]
5.9      YAH      10500      1.0  Event E9 arrives
6.0
6.2
6.3                                Event E2 leaves the time window
                                [79.0]      [88.0]
7.0                                Event E3 and E4 leave the time window
7.2

```

A.3.5. Output Rate Limiting - Snapshot

The statement for this sample reads:

```

select irstream sum(price) from MarketData.win:time(5.5 sec)
output snapshot every 1 seconds

```

The output is as follows:

Input				Output	
				Insert Stream	Remove Stream
Time	Symbol	Volume	Price		
0.2					
	IBM	100	25.0	Event E1 arrives	
0.8					
	MSFT	5000	9.0	Event E2 arrives	
1.0					
1.2					[34.0]
1.5					
	IBM	150	24.0	Event E3 arrives	
	YAH	10000	1.0	Event E4 arrives	
2.0					
2.1					
	IBM	155	26.0	Event E5 arrives	
2.2					[85.0]
2.5					
3.0					

```

3.2
                                     [85.0]
3.5
    YAH    11000    2.0    Event E6 arrives
4.0
4.2
                                     [87.0]
4.3
    IBM     150    22.0    Event E7 arrives
4.9
    YAH    11500    3.0    Event E8 arrives
5.0
5.2
                                     [112.0]
5.7
                                     Event E1 leaves the time window
5.9
    YAH    10500    1.0    Event E9 arrives
6.0
6.2
                                     [88.0]
6.3
                                     Event E2 leaves the time window
7.0
                                     Event E3 and E4 leave the time window
7.2
                                     [54.0]

```

A.4. Output for Aggregated and Un-grouped Queries

This chapter provides sample output for queries that have aggregation functions, and that do not have a `group by` clause, and in which there are event properties that are not under aggregation.

A.4.1. No Output Rate Limiting

The statement for this sample reads:

```
select irstream symbol, sum(price) from MarketData.win:time(5.5 sec)
```

The output is as follows:

Input				Output	
				Insert Stream	Remove Stream
Time	Symbol	Volume	Price		
0.2					
	IBM	100	25.0	Event E1 arrives	
					[IBM, 25.0]
0.8					
	MSFT	5000	9.0	Event E2 arrives	
					[MSFT, 34.0]
1.0					
1.2					
1.5					
	IBM	150	24.0	Event E3 arrives	
					[IBM, 58.0]
	YAH	10000	1.0	Event E4 arrives	
					[YAH, 59.0]
2.0					
2.1					
	IBM	155	26.0	Event E5 arrives	
					[IBM, 85.0]
2.2					
2.5					
3.0					
3.2					

3.5	YAH	11000	2.0	Event E6 arrives	[YAH, 87.0]
4.0					
4.2					
4.3	IBM	150	22.0	Event E7 arrives	[IBM, 109.0]
4.9	YAH	11500	3.0	Event E8 arrives	[YAH, 112.0]
5.0					
5.2					
5.7				Event E1 leaves the time window	[IBM, 87.0]
5.9	YAH	10500	1.0	Event E9 arrives	[YAH, 88.0]
6.0					
6.2					
6.3				Event E2 leaves the time window	[MSFT, 79.0]
7.0				Event E3 and E4 leave the time window	[IBM, 54.0]
					[YAH, 54.0]
7.2					

A.4.2. Output Rate Limiting - Default

The statement for this sample reads:

```
select istream symbol, sum(price) from MarketData.win:time(5.5 sec)
output every 1 seconds
```

The output is as follows:

Input				Output	
				Insert Stream	Remove Stream
Time	Symbol	Volume	Price		
0.2					
	IBM	100	25.0	Event E1 arrives	
0.8					
	MSFT	5000	9.0	Event E2 arrives	
1.0					
1.2					
				[IBM, 25.0]	
				[MSFT, 34.0]	
1.5					
	IBM	150	24.0	Event E3 arrives	
	YAH	10000	1.0	Event E4 arrives	
2.0					
2.1					
	IBM	155	26.0	Event E5 arrives	
2.2					
				[IBM, 58.0]	
				[YAH, 59.0]	
				[IBM, 85.0]	
2.5					
3.0					
3.2					
				(empty result)	(empty result)
3.5					
	YAH	11000	2.0	Event E6 arrives	
4.0					
4.2					
				[YAH, 87.0]	

```

4.3      IBM      150      22.0    Event E7 arrives
4.9      YAH      11500     3.0    Event E8 arrives
5.0
5.2                                     [IBM, 109.0]
                                     [YAH, 112.0]
5.7      Event E1 leaves the time window
5.9      YAH      10500     1.0    Event E9 arrives
6.0
6.2                                     [YAH, 88.0]      [IBM, 87.0]
6.3      Event E2 leaves the time window
7.0      Event E3 and E4 leave the time window
7.2
                                     [MSFT, 79.0]
                                     [IBM, 54.0]
                                     [YAH, 54.0]

```

A.4.3. Output Rate Limiting - Last

The statement for this sample reads:

```

select istream symbol, sum(price) from MarketData.win:time(5.5 sec)
output last every 1 seconds

```

The output is as follows:

Input				Output	
				Insert Stream	Remove Stream
Time	Symbol	Volume	Price		
0.2					
	IBM	100	25.0	Event E1 arrives	
0.8					
	MSFT	5000	9.0	Event E2 arrives	
1.0					
1.2					[MSFT, 34.0]
1.5					
	IBM	150	24.0	Event E3 arrives	
	YAH	10000	1.0	Event E4 arrives	
2.0					
2.1					
	IBM	155	26.0	Event E5 arrives	
2.2					[IBM, 85.0]
2.5					
3.0					
3.2					(empty result) (empty result)
3.5					
	YAH	11000	2.0	Event E6 arrives	
4.0					
4.2					[YAH, 87.0]
4.3					
	IBM	150	22.0	Event E7 arrives	
4.9					
	YAH	11500	3.0	Event E8 arrives	
5.0					
5.2					[YAH, 112.0]
5.7				Event E1 leaves the time window	
5.9					

6.0	YAH	10500	1.0	Event E9 arrives	
6.2					
6.3					[YAH, 88.0] [IBM, 87.0]
7.0				Event E2 leaves the time window	
7.2				Event E3 and E4 leave the time window	
					[YAH, 54.0]

A.4.4. Output Rate Limiting - First

The statement for this sample reads:

```
select symbol, sum(price) from MarketData.win:time(5.5 sec)
output first every 1 seconds
```

The output is as follows:

Input				Output	
Time	Symbol	Volume	Price	Insert Stream	Remove Stream
0.2					
	IBM	100	25.0	Event E1 arrives	
					[IBM, 25.0]
0.8					
	MSFT	5000	9.0	Event E2 arrives	
1.0					
1.2					
1.5					
	IBM	150	24.0	Event E3 arrives	
					[IBM, 58.0]
	YAH	10000	1.0	Event E4 arrives	
2.0					
2.1					
	IBM	155	26.0	Event E5 arrives	
2.2					
2.5					
3.0					
3.2					
					(empty result) (empty result)
3.5					
	YAH	11000	2.0	Event E6 arrives	
					[YAH, 87.0]
4.0					
4.2					
4.3					
	IBM	150	22.0	Event E7 arrives	
					[IBM, 109.0]
4.9					
	YAH	11500	3.0	Event E8 arrives	
5.0					
5.2					
5.7				Event E1 leaves the time window	
					[IBM, 87.0]
5.9					
	YAH	10500	1.0	Event E9 arrives	
6.0					
6.2					
6.3				Event E2 leaves the time window	
					[MSFT, 79.0]
7.0				Event E3 and E4 leave the time window	
7.2					

A.4.5. Output Rate Limiting - Snapshot

The statement for this sample reads:

```
select istream symbol, sum(price) from MarketData.win:time(5.5 sec)
output snapshot every 1 seconds
```

The output is as follows:

Input					Output	
					Insert Stream	Remove Stream
Time	Symbol	Volume	Price			
0.2						
	IBM	100	25.0	Event E1 arrives		
0.8						
	MSFT	5000	9.0	Event E2 arrives		
1.0						
1.2						
					[IBM, 34.0]	
					[MSFT, 34.0]	
1.5						
	IBM	150	24.0	Event E3 arrives		
	YAH	10000	1.0	Event E4 arrives		
2.0						
2.1						
	IBM	155	26.0	Event E5 arrives		
2.2						
					[IBM, 85.0]	
					[MSFT, 85.0]	
					[IBM, 85.0]	
					[YAH, 85.0]	
					[IBM, 85.0]	
2.5						
3.0						
3.2						
					[IBM, 85.0]	
					[MSFT, 85.0]	
					[IBM, 85.0]	
					[YAH, 85.0]	
					[IBM, 85.0]	
3.5						
	YAH	11000	2.0	Event E6 arrives		
4.0						
4.2						
					[IBM, 87.0]	
					[MSFT, 87.0]	
					[IBM, 87.0]	
					[YAH, 87.0]	
					[IBM, 87.0]	
					[YAH, 87.0]	
4.3						
	IBM	150	22.0	Event E7 arrives		
4.9						
	YAH	11500	3.0	Event E8 arrives		
5.0						
5.2						
					[IBM, 112.0]	
					[MSFT, 112.0]	
					[IBM, 112.0]	
					[YAH, 112.0]	
					[IBM, 112.0]	
					[YAH, 112.0]	
					[IBM, 112.0]	
					[YAH, 112.0]	
5.7				Event E1 leaves the time window		
5.9						
	YAH	10500	1.0	Event E9 arrives		
6.0						

```

6.2
                                [MSFT, 88.0]
                                [IBM, 88.0]
                                [YAH, 88.0]
                                [IBM, 88.0]
                                [YAH, 88.0]
                                [IBM, 88.0]
                                [YAH, 88.0]
                                [YAH, 88.0]
6.3
7.0
7.2
                                [IBM, 54.0]
                                [YAH, 54.0]
                                [IBM, 54.0]
                                [YAH, 54.0]
                                [YAH, 54.0]
                                [YAH, 54.0]

```

Event E2 leaves the time window

Event E3 and E4 leave the time window

A.5. Output for Fully-aggregated and Grouped Queries

This chapter provides sample output for queries that have aggregation functions, and that have a `group by` clause, and in which all event properties are under aggregation or appear in the `group by` clause.

A.5.1. No Output Rate Limiting

The statement for this sample reads:

```

select istream symbol, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
order by symbol

```

The output is as follows:

Input					Output	
					Insert Stream	Remove Stream
Time	Symbol	Volume	Price			
0.2						
	IBM	100	25.0	Event E1 arrives	[IBM, 25.0]	[IBM, null]
0.8						
	MSFT	5000	9.0	Event E2 arrives	[MSFT, 9.0]	[MSFT, null]
1.0						
1.2						
1.5						
	IBM	150	24.0	Event E3 arrives	[IBM, 49.0]	[IBM, 25.0]
	YAH	10000	1.0	Event E4 arrives	[YAH, 1.0]	[YAH, null]
2.0						
2.1						
	IBM	155	26.0	Event E5 arrives	[IBM, 75.0]	[IBM, 49.0]
2.2						
2.5						
3.0						
3.2						
3.5						
	YAH	11000	2.0	Event E6 arrives	[YAH, 3.0]	[YAH, 1.0]
4.0						
4.2						

4.3	IBM	150	22.0	Event E7 arrives		
					[IBM, 97.0]	[IBM, 75.0]
4.9	YAH	11500	3.0	Event E8 arrives		
					[YAH, 6.0]	[YAH, 3.0]
5.0						
5.2						
5.7				Event E1 leaves the time window		
					[IBM, 72.0]	[IBM, 97.0]
5.9	YAH	10500	1.0	Event E9 arrives		
					[YAH, 7.0]	[YAH, 6.0]
6.0						
6.2						
6.3				Event E2 leaves the time window		
					[MSFT, null]	[MSFT, 9.0]
7.0				Event E3 and E4 leave the time window		
					[IBM, 48.0]	[IBM, 72.0]
					[YAH, 6.0]	[YAH, 7.0]
7.2						

A.5.2. Output Rate Limiting - Default

The default (no keyword) and the `ALL` keyword do not result in the same output. The default generates an output row per input event, while the `ALL` keyword generates a row for all groups.

The statement for this sample reads:

```
select istream symbol, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output every 1 seconds
```

The output is as follows:

				Input		Output
				Insert Stream	Remove Stream	
Time	Symbol	Volume	Price			
0.2						
	IBM	100	25.0	Event E1 arrives		
0.8						
	MSFT	5000	9.0	Event E2 arrives		
1.0						
1.2						
				[IBM, 25.0]	[IBM, null]	
				[MSFT, 9.0]	[MSFT, null]	
1.5						
	IBM	150	24.0	Event E3 arrives		
	YAH	10000	1.0	Event E4 arrives		
2.0						
2.1						
	IBM	155	26.0	Event E5 arrives		
2.2						
				[IBM, 49.0]	[IBM, 25.0]	
				[YAH, 1.0]	[YAH, null]	
				[IBM, 75.0]	[IBM, 49.0]	
2.5						
3.0						
3.2						
				(empty result)	(empty result)	
3.5						
	YAH	11000	2.0	Event E6 arrives		
4.0						
4.2						
				[YAH, 3.0]	[YAH, 1.0]	

4.3				Event E7 arrives		
4.9	IBM	150	22.0			
	YAH	11500	3.0	Event E8 arrives		
5.0						
5.2					[IBM, 97.0]	[IBM, 75.0]
					[YAH, 6.0]	[YAH, 3.0]
5.7				Event E1 leaves the time window		
5.9	YAH	10500	1.0	Event E9 arrives		
6.0						
6.2					[IBM, 72.0]	[IBM, 97.0]
					[YAH, 7.0]	[YAH, 6.0]
6.3				Event E2 leaves the time window		
7.0				Event E3 and E4 leave the time window		
7.2					[MSFT, null]	[MSFT, 9.0]
					[YAH, 6.0]	[YAH, 7.0]
					[IBM, 48.0]	[IBM, 72.0]

A.5.3. Output Rate Limiting - All

The statement for this sample reads:

```
select istream symbol, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output all every 1 seconds
order by symbol
```

The output is as follows:

Input				Output	
				Insert Stream	Remove Stream
Time	Symbol	Volume	Price		
0.2					
	IBM	100	25.0	Event E1 arrives	
0.8					
	MSFT	5000	9.0	Event E2 arrives	
1.0					
1.2					
					[IBM, 25.0]
					[MSFT, 9.0]
1.5					[IBM, null]
					[MSFT, null]
	IBM	150	24.0	Event E3 arrives	
	YAH	10000	1.0	Event E4 arrives	
2.0					
2.1					
	IBM	155	26.0	Event E5 arrives	
2.2					
					[IBM, 75.0]
					[MSFT, 9.0]
					[YAH, 1.0]
2.5					
3.0					
3.2					
					[IBM, 75.0]
					[MSFT, 9.0]
					[YAH, 1.0]
3.5					
	YAH	11000	2.0	Event E6 arrives	
4.0					
4.2					
					[IBM, 75.0]
					[MSFT, 9.0]
					[IBM, 75.0]
					[MSFT, 9.0]

					[YAH, 3.0]	[YAH, 1.0]
4.3						
	IBM	150	22.0	Event E7 arrives		
4.9						
	YAH	11500	3.0	Event E8 arrives		
5.0						
5.2						
					[IBM, 97.0]	[IBM, 75.0]
					[MSFT, 9.0]	[MSFT, 9.0]
					[YAH, 6.0]	[YAH, 3.0]
5.7				Event E1 leaves the time window		
5.9						
	YAH	10500	1.0	Event E9 arrives		
6.0						
6.2						
					[IBM, 72.0]	[IBM, 97.0]
					[MSFT, 9.0]	[MSFT, 9.0]
					[YAH, 7.0]	[YAH, 6.0]
6.3				Event E2 leaves the time window		
7.0				Event E3 and E4 leave the time window		
7.2						
					[IBM, 48.0]	[IBM, 72.0]
					[MSFT, null]	[MSFT, 9.0]
					[YAH, 6.0]	[YAH, 7.0]

A.5.4. Output Rate Limiting - Last

The statement for this sample reads:

```
select istream symbol, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output last every 1 seconds
order by symbol
```

The output is as follows:

Input					Output	
					Insert Stream	Remove Stream
Time	Symbol	Volume	Price			
0.2						
	IBM	100	25.0	Event E1 arrives		
0.8						
	MSFT	5000	9.0	Event E2 arrives		
1.0						
1.2						
					[IBM, 25.0]	[IBM, null]
					[MSFT, 9.0]	[MSFT, null]
1.5						
	IBM	150	24.0	Event E3 arrives		
	YAH	10000	1.0	Event E4 arrives		
2.0						
2.1						
	IBM	155	26.0	Event E5 arrives		
2.2						
					[IBM, 75.0]	[IBM, 25.0]
					[YAH, 1.0]	[YAH, null]
2.5						
3.0						
3.2						
					(empty result)	(empty result)
3.5						
	YAH	11000	2.0	Event E6 arrives		
4.0						
4.2						
					[YAH, 3.0]	[YAH, 1.0]
4.3						

4.9	IBM	150	22.0	Event E7 arrives		
5.0	YAH	11500	3.0	Event E8 arrives		
5.2					[IBM, 97.0]	[IBM, 75.0]
					[YAH, 6.0]	[YAH, 3.0]
5.7				Event E1 leaves the time window		
5.9	YAH	10500	1.0	Event E9 arrives		
6.0						
6.2					[IBM, 72.0]	[IBM, 97.0]
					[YAH, 7.0]	[YAH, 6.0]
6.3				Event E2 leaves the time window		
7.0				Event E3 and E4 leave the time window		
7.2					[IBM, 48.0]	[IBM, 72.0]
					[MSFT, null]	[MSFT, 9.0]
					[YAH, 6.0]	[YAH, 7.0]

A.5.5. Output Rate Limiting - First

The statement for this sample reads:

```
select istream symbol, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output first every 1 seconds
```

The output is as follows:

Input					Output	
					Insert Stream	Remove Stream
Time	Symbol	Volume	Price			
0.2						
	IBM	100	25.0	Event E1 arrives	[IBM, 25.0]	[IBM, null]
0.8						
	MSFT	5000	9.0	Event E2 arrives	[MSFT, 9.0]	[MSFT, null]
1.0						
1.2						
1.5						
	IBM	150	24.0	Event E3 arrives	[IBM, 49.0]	[IBM, 25.0]
	YAH	10000	1.0	Event E4 arrives	[YAH, 1.0]	[YAH, null]
2.0						
2.1	IBM	155	26.0	Event E5 arrives		
2.2						
2.5						
3.0						
3.2						
3.5	YAH	11000	2.0	Event E6 arrives	[YAH, 3.0]	[YAH, 1.0]
4.0						
4.2						
4.3	IBM	150	22.0	Event E7 arrives	[IBM, 97.0]	[IBM, 75.0]
4.9	YAH	11500	3.0	Event E8 arrives	[YAH, 6.0]	[YAH, 3.0]

```

5.0
5.2
5.7                Event E1 leaves the time window
                        [IBM, 72.0]          [IBM, 97.0]
5.9      YAH   10500   1.0  Event E9 arrives
                        [YAH, 7.0]          [YAH, 6.0]
6.0
6.2
6.3                Event E2 leaves the time window
                        [MSFT, null]        [MSFT, 9.0]
7.0                Event E3 and E4 leave the time window
                        [IBM, 48.0]         [IBM, 72.0]
                        [YAH, 6.0]          [YAH, 7.0]
7.2

```

A.5.6. Output Rate Limiting - Snapshot

The statement for this sample reads:

```

select istream symbol, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output snapshot every 1 seconds
order by symbol

```

The output is as follows:

Input				Output	
Time	Symbol	Volume	Price	Insert Stream	Remove Stream
0.2					
	IBM	100	25.0	Event E1 arrives	
0.8					
	MSFT	5000	9.0	Event E2 arrives	
1.0					
1.2					[IBM, 25.0] [MSFT, 9.0]
1.5					
	IBM	150	24.0	Event E3 arrives	
	YAH	10000	1.0	Event E4 arrives	
2.0					
2.1					
	IBM	155	26.0	Event E5 arrives	
2.2					[IBM, 75.0] [MSFT, 9.0] [YAH, 1.0]
2.5					
3.0					
3.2					[IBM, 75.0] [MSFT, 9.0] [YAH, 1.0]
3.5					
	YAH	11000	2.0	Event E6 arrives	
4.0					
4.2					[IBM, 75.0] [MSFT, 9.0] [YAH, 3.0]
4.3					
	IBM	150	22.0	Event E7 arrives	
4.9					
	YAH	11500	3.0	Event E8 arrives	
5.0					

```

5.2                                     [IBM, 97.0]
                                     [MSFT, 9.0]
                                     [YAH, 6.0]
5.7                                     Event E1 leaves the time window
5.9
    YAH    10500    1.0  Event E9 arrives
6.0
6.2                                     [IBM, 72.0]
                                     [MSFT, 9.0]
                                     [YAH, 7.0]
6.3                                     Event E2 leaves the time window
7.0                                     Event E3 and E4 leave the time window
7.2                                     [IBM, 48.0]
                                     [YAH, 6.0]

```

A.6. Output for Aggregated and Grouped Queries

This chapter provides sample output for queries that have aggregation functions, and that have a `group by` clause, and in which some event properties are not under aggregation.

A.6.1. No Output Rate Limiting

The statement for this sample reads:

```
select irstream symbol, volume, sum(price) from MarketData.win:time(5.5 sec) group by symbol
```

The output is as follows:

Input				Output	
				Insert Stream	Remove Stream
Time	Symbol	Volume	Price		
0.2					
	IBM	100	25.0	Event E1 arrives	[IBM, 100, 25.0]
0.8					
	MSFT	5000	9.0	Event E2 arrives	[MSFT, 5000, 9.0]
1.0					
1.2					
1.5					
	IBM	150	24.0	Event E3 arrives	[IBM, 150, 49.0]
	YAH	10000	1.0	Event E4 arrives	[YAH, 10000, 1.0]
2.0					
2.1					
	IBM	155	26.0	Event E5 arrives	[IBM, 155, 75.0]
2.2					
2.5					
3.0					
3.2					
3.5					
	YAH	11000	2.0	Event E6 arrives	[YAH, 11000, 3.0]
4.0					
4.2					
4.3					
	IBM	150	22.0	Event E7 arrives	

```

                                [IBM, 150, 97.0]
4.9      YAH      11500      3.0      Event E8 arrives
                                [YAH, 11500, 6.0]
5.0
5.2
5.7                                Event E1 leaves the time window
                                [IBM, 100, 72.0]
5.9      YAH      10500      1.0      Event E9 arrives
                                [YAH, 10500, 7.0]
6.0
6.2
6.3                                Event E2 leaves the time window
                                [MSFT, 5000, null]
7.0                                Event E3 and E4 leave the time window
                                [IBM, 150, 48.0]
                                [YAH, 10000, 6.0]
7.2

```

A.6.2. Output Rate Limiting - Default

The default (no keyword) and the ALL keyword do not result in the same output. The default generates an output row per input event, while the ALL keyword generates a row for all groups based on the last new event for each group.

The statement for this sample reads:

```

select istream symbol, volume, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output every 1 seconds

```

The output is as follows:

Input				Output	
				Insert Stream	Remove Stream
Time	Symbol	Volume	Price		
0.2					
	IBM	100	25.0	Event E1 arrives	
0.8					
	MSFT	5000	9.0	Event E2 arrives	
1.0					
1.2					
					[IBM, 100, 25.0]
					[MSFT, 5000, 9.0]
1.5					
	IBM	150	24.0	Event E3 arrives	
	YAH	10000	1.0	Event E4 arrives	
2.0					
2.1					
	IBM	155	26.0	Event E5 arrives	
2.2					
					[IBM, 150, 49.0]
					[YAH, 10000, 1.0]
					[IBM, 155, 75.0]
2.5					
3.0					
3.2					
					(empty result)
3.5					(empty result)
	YAH	11000	2.0	Event E6 arrives	
4.0					
4.2					
					[YAH, 11000, 3.0]
4.3					

4.9	IBM	150	22.0	Event E7 arrives	
5.0	YAH	11500	3.0	Event E8 arrives	
5.2					[IBM, 150, 97.0] [YAH, 11500, 6.0]
5.7				Event E1 leaves the time window	
5.9	YAH	10500	1.0	Event E9 arrives	
6.0					[YAH, 10500, 7.0] [IBM, 100, 72.0]
6.2					
6.3				Event E2 leaves the time window	
7.0				Event E3 and E4 leave the time window	
7.2					[MSFT, 5000, null] [IBM, 150, 48.0] [YAH, 10000, 6.0]

A.6.3. Output Rate Limiting - All

The statement for this sample reads:

```
select istream symbol, volume, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output all every 1 seconds
order by symbol
```

The output is as follows:

Input				Output	
				Insert Stream	Remove Stream
Time	Symbol	Volume	Price		
0.2					
	IBM	100	25.0	Event E1 arrives	
0.8					
	MSFT	5000	9.0	Event E2 arrives	
1.0					
1.2					[IBM, 100, 25.0] [MSFT, 5000, 9.0]
1.5					
	IBM	150	24.0	Event E3 arrives	
	YAH	10000	1.0	Event E4 arrives	
2.0					
2.1					
	IBM	155	26.0	Event E5 arrives	
2.2					[IBM, 150, 49.0] [IBM, 155, 75.0] [MSFT, 5000, 9.0] [YAH, 10000, 1.0]
2.5					
3.0					
3.2					[IBM, 155, 75.0] [MSFT, 5000, 9.0] [YAH, 10000, 1.0]
3.5					
	YAH	11000	2.0	Event E6 arrives	
4.0					
4.2					[IBM, 155, 75.0] [MSFT, 5000, 9.0] [YAH, 11000, 3.0]

```

4.3      IBM      150      22.0  Event E7 arrives
4.9      YAH      11500     3.0  Event E8 arrives
5.0
5.2                                [IBM, 150, 97.0]
                                [MSFT, 5000, 9.0]
                                [YAH, 11500, 6.0]
5.7                                Event E1 leaves the time window
5.9      YAH      10500     1.0  Event E9 arrives
6.0
6.2                                [IBM, 150, 72.0]  [IBM, 100, 72.0]
                                [MSFT, 5000, 9.0]
                                [YAH, 10500, 7.0]
6.3                                Event E2 leaves the time window
7.0                                Event E3 and E4 leave the time window
7.2                                [IBM, 150, 48.0]  [IBM, 150, 48.0]
                                [MSFT, 5000, null] [MSFT, 5000, null]
                                [YAH, 10500, 6.0] [YAH, 10000, 6.0]

```

A.6.4. Output Rate Limiting - Last

The statement for this sample reads:

```

select istream symbol, volume, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output last every 1 seconds
order by symbol

```

The output is as follows:

Input				Output	
				Insert Stream	Remove Stream
Time	Symbol	Volume	Price		
0.2					
	IBM	100	25.0	Event E1 arrives	
0.8					
	MSFT	5000	9.0	Event E2 arrives	
1.0					
1.2					[IBM, 100, 25.0] [MSFT, 5000, 9.0]
1.5					
	IBM	150	24.0	Event E3 arrives	
	YAH	10000	1.0	Event E4 arrives	
2.0					
2.1					
	IBM	155	26.0	Event E5 arrives	
2.2					[IBM, 155, 75.0] [YAH, 10000, 1.0]
2.5					
3.0					
3.2					(empty result) (empty result)
3.5					
	YAH	11000	2.0	Event E6 arrives	
4.0					
4.2					[YAH, 11000, 3.0]
4.3					
	IBM	150	22.0	Event E7 arrives	

```

4.9      YAH      11500      3.0      Event E8 arrives
5.0
5.2                                     [IBM, 150, 97.0]
                                     [YAH, 11500, 6.0]
5.7                                     Event E1 leaves the time window
5.9      YAH      10500      1.0      Event E9 arrives
6.0
6.2                                     [YAH, 10500, 7.0] [IBM, 100, 72.0]
6.3                                     Event E2 leaves the time window
7.0                                     Event E3 and E4 leave the time window
7.2
                                     [IBM, 150, 48.0]
                                     [MSFT, 5000, null]
                                     [YAH, 10000, 6.0]

```

A.6.5. Output Rate Limiting - First

The statement for this sample reads:

```

select istream symbol, volume, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output first every 1 seconds

```

The output is as follows:

Input					Output	
					Insert Stream	Remove Stream
Time	Symbol	Volume	Price			
0.2						
	IBM	100	25.0	Event E1 arrives		
					[IBM, 100, 25.0]	
0.8						
	MSFT	5000	9.0	Event E2 arrives		
					[MSFT, 5000, 9.0]	
1.0						
1.2						
1.5						
	IBM	150	24.0	Event E3 arrives		
					[IBM, 150, 49.0]	
	YAH	10000	1.0	Event E4 arrives		
					[YAH, 10000, 1.0]	
2.0						
2.1						
	IBM	155	26.0	Event E5 arrives		
2.2						
2.5						
3.0						
3.2						
3.5						
	YAH	11000	2.0	Event E6 arrives		
					[YAH, 11000, 3.0]	
4.0						
4.2						
4.3						
	IBM	150	22.0	Event E7 arrives		
					[IBM, 150, 97.0]	
4.9						
	YAH	11500	3.0	Event E8 arrives		
					[YAH, 11500, 6.0]	
5.0						
5.2						
5.7				Event E1 leaves the time window		

```

                    [IBM, 100, 72.0]
5.9      YAH      10500      1.0      Event E9 arrives
                    [YAH, 10500, 7.0]
6.0
6.2
6.3                      Event E2 leaves the time window
                    [MSFT, 5000, null]
7.0                      Event E3 and E4 leave the time window
                    [IBM, 150, 48.0]
                    [YAH, 10000, 6.0]
7.2

```

A.6.6. Output Rate Limiting - Snapshot

The statement for this sample reads:

```

select istream symbol, volume, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output snapshot every 1 seconds

```

The output is as follows:

Input				Output	
Time	Symbol	Volume	Price	Insert Stream	Remove Stream
0.2					
	IBM	100	25.0	Event E1 arrives	
0.8					
	MSFT	5000	9.0	Event E2 arrives	
1.0					
1.2					
				[IBM, 100, 25.0]	
				[MSFT, 5000, 9.0]	
1.5					
	IBM	150	24.0	Event E3 arrives	
	YAH	10000	1.0	Event E4 arrives	
2.0					
2.1					
	IBM	155	26.0	Event E5 arrives	
2.2					
				[IBM, 100, 75.0]	
				[MSFT, 5000, 9.0]	
				[IBM, 150, 75.0]	
				[YAH, 10000, 1.0]	
				[IBM, 155, 75.0]	
2.5					
3.0					
3.2					
				[IBM, 100, 75.0]	
				[MSFT, 5000, 9.0]	
				[IBM, 150, 75.0]	
				[YAH, 10000, 1.0]	
				[IBM, 155, 75.0]	
3.5					
	YAH	11000	2.0	Event E6 arrives	
4.0					
4.2					
				[IBM, 100, 75.0]	
				[MSFT, 5000, 9.0]	
				[IBM, 150, 75.0]	
				[YAH, 10000, 3.0]	
				[IBM, 155, 75.0]	
				[YAH, 11000, 3.0]	
4.3					
	IBM	150	22.0	Event E7 arrives	


```
4.9      YAH      11500      3.0      Event E8 arrives
5.0
5.2                                     [IBM, 100, 97.0]
                                     [MSFT, 5000, 9.0]
                                     [IBM, 150, 97.0]
                                     [YAH, 10000, 6.0]
                                     [IBM, 155, 97.0]
                                     [YAH, 11000, 6.0]
                                     [IBM, 150, 97.0]
                                     [YAH, 11500, 6.0]
5.7                                     Event E1 leaves the time window
5.9      YAH      10500      1.0      Event E9 arrives
6.0
6.2                                     [MSFT, 5000, 9.0]
                                     [IBM, 150, 72.0]
                                     [YAH, 10000, 7.0]
                                     [IBM, 155, 72.0]
                                     [YAH, 11000, 7.0]
                                     [IBM, 150, 72.0]
                                     [YAH, 11500, 7.0]
                                     [YAH, 10500, 7.0]
6.3                                     Event E2 leaves the time window
7.0                                     Event E3 and E4 leave the time window
7.2                                     [IBM, 155, 48.0]
                                     [YAH, 11000, 6.0]
                                     [IBM, 150, 48.0]
                                     [YAH, 11500, 6.0]
                                     [YAH, 10500, 6.0]
```

Appendix B. Reserved Keywords

The words in the following table are explicitly reserved in EPL, however certain keywords are allowed as event property names in expressions and as column names in the rename syntax of the `select` clause.

Most of the words in the table are forbidden by standard SQL as well. A few are reserved because EPL needs them.

Names of built-in functions and certain auxiliary keywords are permitted as identifiers for use either as event property names in expressions and for the column rename syntax. The second column in the table below indicates which keywords are acceptable. For example, `count` is acceptable.

An example of permitted use is:

```
select last, count(*) as count from MyEvent
```

This example shows an incorrect use of a reserved keyword:

```
// incorrect
select insert from MyEvent
```

The table of explicitly reserved keywords and permitted keywords:

Table B.1. Reserved Keywords

Keyword	Property Name and Rename Syntax
after	-
all	-
and	-
as	-
at	yes
asc	-
avedev	yes
avg	yes
between	-
by	-
case	-
cast	yes
coalesce	yes
count	yes
create	-
current_timestamp	-

Keyword	Property Name and Rename Syntax
day	-
days	-
delete	-
define	yes
desc	-
distinct	-
else	-
end	-
escape	yes
events	yes
every	yes
exists	-
false	yes
first	yes
from	-
full	yes
group	-
having	-
hour	-
hours	-
in	-
inner	-
insert	-
instanceof	yes
into	-
irstream	-
is	-
istream	-
join	yes
last	yes
lastweekday	yes
left	yes
limit	-

Keyword	Property Name and Rename Syntax
like	-
max	yes
match_recognize	-
matches	-
median	yes
measures	yes
metadatasql	yes
min	yes
minute	yes
minutes	yes
msec	yes
millisecond	yes
milliseconds	yes
not	-
null	-
offset	-
on	-
or	-
order	-
outer	yes
output	-
partition	-
pattern	yes
prev	yes
prior	yes
regexp	-
retain-union	yes
retain-intersection	yes
right	yes
rstream	-
sec	-
second	-
seconds	-

Reserved Keywords

Keyword	Property Name and Rename Syntax
select	-
set	-
some	-
snapshot	yes
sql	yes
stddev	yes
sum	yes
then	-
true	-
unidirectional	yes
until	yes
update	-
variable	yes
weekday	yes
when	-
where	-
window	yes

Index

Symbols

-> pattern operator, 139

A

after, 62

aggregation functions

 custom plug-in, 335

 overview, 177

and pattern operator, 137

annotation, 38

 application-provided, 38

 builtin, 39

 interrogating, 284

arithmetic operators, 160

array definition operator, 161

B

between operator, 163

binary operators, 161

C

case control flow function, 169

cast function, 169

coalesce function, 170

concatenation operators, 160

configuration

 items to configure, 286

 logging, 316

 overview, 285

 programmatic, 285

 runtime, 257, 315

 via XML, 285

Configuration class, 285

constants, 9, 37

correlation view, 244

create index, 104

create schema, 107

create window, insert, 98

current_timestamp function, 170

D

data types, 36

data window views

 custom plug-in view, 331

 externally-timed window, 234

 grouped data window, 239

 keep-all window, 237

 last event window, 241

length batch window, 234

length window, 233

overview, 231

size window, 240

sorted window, 246

time batch window, 235

time length batch window, 236

time window, 234

time-accumulating window, 237

time-order window, 246

unique window, 238

decorated event, 69

deployment

 EPDeploymentAdmin interface, 320

 EPL module, 319

 J2EE, 322

derived-value views

 correlation, 244

 overview, 232

 regression, 243

 univariate statistics, 242

 weighted average, 245

differences, 284

dot operator, 162

duck typing, 162

dynamic event properties, 5

E

enumeration, 9

enumerator, 255

EPAdministrator interface, 249

EPL

 from clause, 48

 group by clause, 55

 having clause, 57

 inner join, 78

 insert into clause, 66

 join, 76

 join, unidirectional, 79

 joining non-relational data via method invocation, 86

 joining relational data via SQL, 81

 limit clause, 65

 named window, 90

 deleting from, 100

 inserting into, 93

 merge, 101

 populating from a named window, 98

 selecting from, 94

 triggered playback using On Insert, 97

 triggered select using On Select, 95

 updating, 98

- upsert, 101
- order by clause, 64
- outer join, 78
- outer join, unidirectional, 79
- output control and stabilizing, 59
- select clause, 43
- subqueries, 70
- variable, 110
- where clause, 54
- EPRuntime interface, 257
- EPServiceProviderManager class, 248
- EPStatement interface, 249
- EPStatementObjectModel interface, 275
- escape, 36
- event
 - additional representations, 20
 - bulk, 21
 - coarse, 21
 - dynamic properties, 5
 - insert into, 21
 - Map representation, 10
 - object, 6
 - properties, 4
 - underlying representation, 3
 - update, 20
 - version, 20
 - XML representation, 14
- event as a property, 69
- event object
 - custom, 342
- event type
 - declaring, 107
- EventBean interface, 23, 260
- EventType interface, 260
- every pattern operator, 128
- every-distinct pattern operator, 132
- exists function, 170
- expression declaration, 41
- external time, 267
- externally-timed window, 234

F

- first event, 242
- first length window, 238
- first time window, 238
- first unique window, 242
- followed-by pattern operator, 139
- from clause, 48
- functions
 - case control flow, 169
 - cast, 169
 - coalesce, 170

- current_timestamp, 170
- exists, 170
- instance-of, 170
- max, 171
- min, 171
- previous, 171
- previous count, 175
- previous tail, 173
- previous window, 174
- prior, 176
- type-of, 176
- user-defined, 167, 183

G

- group by clause, 55
- groupwin window, 239

H

- having clause, 57

I

- in set operator, 162
- inner join, 78
- insert into clause, 66
- insert stream, 23
- instance-of function, 170

J

- join, 76
 - from clause, 48
 - non-relational data via method invocation, 86
 - relational data via SQL, 81

K

- keep-all window, 237
- keywords, 35

L

- lambda
 - expression declaration, 41
- last event window, 241
- length batch window, 234
- length window, 233
- like operator, 164
- limit clause, 65
- limiting output row count, 65
- literals, 37
- logical and comparison operators, 160

M

- map event representation, 10
- match recognize

- comparison, 147
- overview, 147
- match_recognize
 - operator precedences, 150
- max function, 171
- merge, 101
- methods
 - date-time, 211
 - enumeration, 186
- min function, 171

N

- named window, 90
 - create index, 104
 - deleting from, 100
 - index, 104
 - inserting into, 93
 - merge, 101
 - populating from a named window, 98
 - selecting from, 94
 - triggered playback using On Insert, 97
 - triggered select using On Select, 95
 - updating, 98
 - upsert, 101
 - versioning events, 105
- not pattern operator, 138

O

- on-delete, 100
- on-insert, 97
- on-select, 95
- on-update, 98
- operators
 - arithmetic, 160
 - array definition, 161
 - between, 163
 - binary, 161
 - concatenation, 160
 - dot (period), 162
 - in, 162
 - like, 164
 - logical and comparison, 160
 - regexp, 164
- or pattern operator, 138
- order by clause, 64
- ordering output, 64
- outer join, 78
- output
 - suppressing output, 62
- output control and stabilizing clause, 59
- output ordering, 64
- output row count, 65

- output when, 61

P

- packaging
 - EPDeploymentAdmin interface, 320
 - EPL module, 319
 - J2EE, 322
- pattern
 - filter expressions, 127
 - operator precedences, 126
 - overview, 123
- pattern atom, 143
- pattern guard, 140
 - custom plug-in, 338
 - timer-within, 141
 - timer-withinmax, 141
 - while, 142
- pattern observer
 - custom plug-in, 340
 - timer-at, 144
 - timer-interval, 143
- pattern operator
 - and, 137
 - every, 128
 - every-distinct, 132
 - followed-by, 139
 - not, 138
 - or, 138
- plug-in
 - custom aggregation function, 335
 - custom event object, 342
 - custom pattern guard, 338
 - custom pattern observer, 340
 - custom view, 331
 - single-row function, 330
- plug-in loader, 283
- previous count function, 175
- previous function, 171
- previous tail function, 173
- previous window function, 174
- prior function, 176
- pull API, 255

R

- regexp operator, 164
- regression view, 243
- relational databases, 81
- remove stream, 24

S

- safe enumerator, 255
- select clause, 43

- single-row functions
 - custom plug-in, 330
- size window, 240
- sorted window, 246
- SQL, 81
- statement
 - receiving results, 250
 - subscriber object, 252
- static methods, 167
- subqueries, 70
- subscriber object, 252
 - multi-row, 254
 - row-by-row, 252

T

- threading, 263
- time
 - controlling, 267
 - resolution, 270
- time batch window, 28, 235
- time length batch window, 236
- time window, 27, 234
- time-accumulating window, 237
- time-order window, 246
- timer-at pattern observer, 144
- timer-interval pattern observer, 143
- timer-within pattern guard, 141
- timer-withinmax pattern guard, 141
- type-of function, 176

U

- UDF
 - user-defined function, 183
- unidirectional joins, 79
- unique window, 238
- univariate statistics view, 242
- UnmatchedListener interface, 258
- UpdateEventHandler delegate, 255
- upsert, 101
- user-defined function, 183
- user-defined single-row function, 167

V

- variable, 110
- variant stream, 68
- views
 - batch window processing, 29
 - correlation, 244
 - custom plug-in view, 331
 - externally-timed window, 234
 - first event, 242
 - first length window, 238

- first time window, 238
- first unique window, 242
- grouped data window, 239
- keep-all window, 237
- last event window, 241
- length batch window, 234
- length window, 233
- overview, 231
- regression, 243
- size window, 240
- sorted window, 246
- time batch window, 235
- time length batch window, 236
- time window, 234
- time-accumulating window, 237
- time-order window, 246
- unique window, 238
- univariate statistics, 242
- weighted average, 245

W

- weighted average view, 245
- where clause, 54
- while pattern guard, 142

X

- XML event representation, 14