Amazon Kinesis Analytics

SQL Reference

Amazon Kinesis Analytics: SQL Reference

Table of Contents

Amazon Kinesis Analytics SQL Reference

The Amazon Kinesis Analytics SQL Reference describes the SQL structured query language supported by Amazon Kinesis Analytics. The language is based on the SQL:2008 standard with some extensions to accommodate the concept of streams.

This guide covers the following:

- Basic Building Blocks (p. 6) – Data Types (p. 6), Streaming SQL Operators (p. 11), Functions (p. 29), Standard Functions (p. 30).
- Standard SQL Operators (p. 123) – CREATE statements (p. 123), SELECT statement (p. 134), MERGE statements (p. 129).
- Operators for transforming and filtering incoming data – WHERE clause (p. 149), JOIN clause (p. 142), GROUP BY clause (p. 148), WINDOW clause (Sliding Windows) (p. 152).
- Logical Operators (p. 25) – AS, AND, OR, etc.

Temporal Predicates

The following table shows a graphic representation of temporal predicates supported by standard SQL and extensions to the SQL standard supported by Amazon Kinesis Analytics. It shows the relationships that each predicate covers. Each relationship is represented as an upper interval and a lower interval with the combined meaning *upperInterval predicate lowerInterval evaluates to TRUE*. The first 7 predicates are standard SQL. The last 10 predicates, shown in bold text, are Amazon Kinesis Analytics extensions to the SQL standard.

Pre	Covered Relationships
CON	
OVE	
EQU	
PRE	
SUC	
IMM PRE	
IMM SUC	
LEA	
LAG	
STR CON	
STR OVE	
STR PRE	

Pre	Covered Relationships
STR SUC	
STR LEA	
STR LAG	
IMM_Y LEA	
IMM_Y LAG	

To enable concise expressions, Amazon Kinesis Analytics also supports the following extensions:

- Optional PERIOD keyword – The PERIOD keyword can be omitted.
- Compact chaining – If two of these predicates occur back to back, separated by an AND, the AND can be omitted provided that the right interval of the first predicate is identical to the left interval of the second predicate.
- TSDIFF – This function takes two TIMESTAMP arguments and returns their difference in milliseconds.

For example, you can write the following expression:

```
  PERIOD (s1,e1) PRECEDES PERIOD(s2,e2)
  AND PERIOD(s2, e2) PRECEDES PERIOD(s3,e3)
```

More concisely as follows:

```
(s1,e1) PRECEDES (s2,e2) PRECEDES PERIOD(s3,e3)
```

The following concise expression:

```
TSDIFF(s,e)
```

Means the following:

```
CAST((e - s) SECOND(10, 3) * 1000 AS BIGINT)
```

Finally, standard SQL allows the CONTAINS predicate to take a single TIMESTAMP as its right-hand argument. For example, the following expression:

```
PERIOD(s, e) CONTAINS t
```

Is equivalent to the following:

```
s <= t AND t < e
```

Syntax

Temporal predicates are integrated into a new BOOLEAN valued expression:

```
<period-expression> :=
  <left-period> <half-period-predicate> <right-period>

<half-period-predicate> :=
  <period-predicate> [ <left-period> <half-period-predicate> ]

<period-predicate> :=
   EQUALS
 | [ STRICTLY ] CONTAINS
 | [ STRICTLY ] OVERLAPS
 | [ STRICTLY | IMMEDIATELY ] PRECEDES
 | [ STRICTLY | IMMEDIATELY ] SUCCEEDS
 | [ STRICTLY | IMMEDIATELY ] LEADS
 | [ STRICTLY | IMMEDIATELY ] LAGS

<left-period> := <bounded-period>

<right-period> := <bounded-period> | <timestamp-expression>

<bounded-period> := [ PERIOD ] ( <start-time>, <end-time> )

<start-time> := <timestamp-expression>

<end-time> := <timestamp-expression>

<timestamp-expression> :=
  an expression which evaluates to a TIMESTAMP value

where <right-period> may evaluate to a <timestamp-expression> only if
the immediately preceding <period-predicate> is [ STRICTLY ] CONTAINS
```

This Boolean expression is supported by the following builtin function:

```
BIGINT tsdiff( startTime TIMESTAMP, endTime TIMESTAMP )
```

Returns the value of (endTime - startTime) in milliseconds.

Example

The following example code records an alarm if a window is open while the air conditioning is on:

```
create or replace pump alarmPump stopped as
  insert into alarmStream( houseID, roomID, alarmTime, alarmMessage )
    select stream w.houseID, w.roomID, current_timestamp,
                  'Window open while air conditioner is on.'
    from
        windowIsOpenEvents over (range interval '1' minute preceding) w
    join
```

```
        acIsOnEvents over (range interval '1' minute preceding) h
    on w.houseID = h.houseID
    where (h.startTime, h.endTime) overlaps (w.startTime, w.endTime);
```

Sample Use Case

The following query uses a temporal predicate to raise a fraud alarm when two people try to use the same credit card simultaneously at two different locations:

```
create pump creditCardFraudPump stopped as
 insert into alarmStream
  select stream
    current_timestamp, creditCardNumber, registerID1, registerID2
  from transactionsPerCreditCard
  where registerID1 <> registerID2
  and (startTime1, endTime1) overlaps (startTime2, endTime2)
;
```

Basic Building Blocks

The following topics discuss the basic building blocks in Amazon Kinesis Analytics that underlie its syntax and operations:

Topics

Data Types

The following table summarizes the data types supported by Amazon Kinesis Analytics.

Data Type	Description	Notes
BIGINT	64-bit signed integer	
BINARY	Binary (non character) data	Substring works on BINARY. Concatenation does not work on BINARY.
BOOLEAN	TRUE, FALSE, or NULL	Evaluates to TRUE, FALSE, and UNKNOWN.
CHAR (n)	A character string of fixed length n. Also specifiable as CHARACTER	n must be greater than 0 and less than 65535.
DATE	A date is a calendar day (year/month/day).	Precision is day. Range runs from the largest value, approximately +229 (in years) to the smallest value, -229.

Data Type	Description	Notes
DECIMAL DEC NUMERIC	A fixed point, with up to 19 significant digits.	Can be specified with DECIMAL, DEC, or NUMERIC.
DOUBLE DOUBLE PRECISION	A 64-bit floating point number	64-bit approx value; -1.79E+308 to 1.79E+308. Follows the ISO DOUBLE PRECISION data type, 53 bits are used for the number's mantissa in scientific notation, representing 15 digits of precision and 8 bytes of storage.
INTEGER INT		32-bit signed integer. Range is -2147483648 to 2147483647 [2**(31) to 2**(31)- 1]
INTERVAL <timeunit> [TO <timeunit>]	Day-time intervals supported, year-month intervals not supported	Allowed in an expression in date arithmetic, but cannot be used as a datatype for a column in a table or stream.
<timeUnit>	The units of a INTERVAL value	Supported units are YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND
SMALLINT	16-bit signed integer	Range is -32768 to 32767 [2**(15) to 2**(15)-1]
REAL	A 32-bit floating point number	Following the ISO REAL data type, 24 bits are used for the number's mantissa in scientific notation, representing 7 digits of precision and 4 bytes of storage. The minimum value is -3.40E +38; the maximum value is 3.40E+38.
TIME	A TIME is a time in a day (hour:minute:second).	Its precision is milliseconds; its range is 00:00:00.000 to 23:59:59.999. Since the system clock runs in UTC, the timezone used for values stored in a TIME or TIMESTAMP column is not considered. for values stored in a TIME or TIMESTAMP column.

Data Type	Description	Notes
TIMESTAMP	A TIMESTAMP is a combined DATE and TIME.	A TIMESTAMP value always has a precision of 1 millisecond. It has no particular timezone. Since the system clock runs in UTC, the timezone used for values stored in a TIME or TIMESTAMP column is not considered. Its range runs from the largest value, approximately +229 (in years) to the smallest value, -229. Each timestamp is stored as a signed 64-bit integer, with 0 representing the Unix epoch (Jan 1, 1970 00:00am). This means that the largest TIMESTAMP value represents approximately 300 million years after 1970, and the smallest value represents approximately 300 million years before 1970. Following the SQL standard, a TIMESTAMP value has an undefined timezone.
TINYINT	8-bit signed integer	Range is -128 to 127,
VARBINARY (n)	Also specifiable as BINARY VARYING	n must be greater than 0 and less than 65535.
VARCHAR (n)	Also specifiable as CHARACTER VARYING	n must be greater than 0 and less than 65535.

Notes

Regarding characters:

- Amazon Kinesis Analytics supports only Java single-byte CHARACTER SETs.
- Implicit type conversion is not supported. That is, characters are mutually assignable if and only if they are taken from the same character repertoire and are values of the data types CHARACTER or CHARACTER VARYING.

Regarding numbers:

- Numbers are mutually comparable and mutually assignable if they are values of the data types NUMERIC, DECIMAL, INTEGER, BIGINT, SMALLINT, TINYINT, REAL, and DOUBLE PRECISION.

The following sets of data types are synonyms:

- DEC and DECIMAL
- DOUBLE PRECISION and DOUBLE
- CHARACTER and CHAR
- CHAR VARYING or CHARACTER VARYING and VARCHAR
- BINARY VARYING and VARBINARY

- INT and INTEGER

- Binary values (data types BINARY and BINARY VARYING) are always mutually comparable and are mutually assignable.

Regarding dates, times, and timestamps:

- Implicit type conversion is not supported (that is, datetime values are mutually assignable only if the source and target of the assignment are both of type DATE, or both of type TIME, or both of type TIMESTAMP).

- The Amazon Kinesis Analytics timezone is always UTC. The time functions, including the Amazon Kinesis Analytics extension CURRENT_ROW_TIMESTAMP, return time in UTC.

Numeric Types and Precision

For DECIMAL we support a maximum of 18 digits for precision and scale.

Precision specifies the maximum number of decimal digits that can be stored in the column, both to the right and to the left of the decimal point. You can specify precisions ranging from 1 digit to 18 digits or use the default precision of 18 digits.

Scale specifies the maximum number of digits that can be stored to the right of the decimal point. Scale must be less than or equal to the precision. You can specify a scale ranging from 0 digits to 18 digits, or use the default scale of 0 digits.

Rule for Divide

Let p1, s1 be the precision and scale of the first operand, such as DECIMAL (10,1).

Let p2, s2 be the precision and scale of the second operand, such as DECIMAL (10,3).

Let p, s be the precision and scale of the result.

Let d be the number of whole digits in the result. Then, the result type is a decimal as shown following:

`d = p1 - s1 + s2`	D = 10 - 1 + 3 Number of whole digits in result = 6
s <= MAX (6, s1 + p2 +1)	S <= MAX (6, 1 + 10 + 1) Scale of result = 14
p = d + s	Precision of result = 18

Precision and scale are capped at their maximum values (18, where scale cannot be larger than precision).

Precedence is first giving at least the scale of the first argument (s >= s1) followed by enough whole digits to represent the result without overflow

Rule for Multiply

Let p1, s1 be the precision and scale of the first operand DECIMAL (10,1).

Let p2, s2 be the precision and scale of the second operand DECIMAL (10,3).

Let p, s be the precision and scale of the result.

Then, the result type is a decimal as shown following:

p = p1 + p2	p = 10 + 10 Precision of result = 18
s = s1 + s2	s = 1 + 3 Scale of result = 4

Rule for Sum or Subtraction

Type-inference strategy whereby the result type of a call is the decimal sum of two exact numeric operands where at least one of the operands is a decimal.

Let p1, s1 be the precision and scale of the first operand DECIMAL (10,1).

Let p2, s2 be the precision and scale of the second operand DECIMAL (10,3).

Let p, s be the precision and scale of the result, as shown following:

s = max(s1, s2)	s = max (1,3) Scale of result = 3
p = max(p1 - s1, p2 - s2) + s + 1	p = max(10-1,10-3) + 3 + 1 Precision of result = 11

s and p are capped at their maximum values

Identifiers

All identifiers may be up to 128 characters. Identifiers may be quoted (with case-sensitivity) by enclosing them in double-quote marks ("), or unquoted (with implicit uppercasing before both storage and lookup).

Unquoted identifiers must start with a letter or underscore, and be followed by letters, digits or underscores; letters are all converted to upper case.

Quoted identifiers can contain other punctuation too (in fact, any Unicode character except control characters: codes 0x0000 through 0x001F). You can include a double-quote in an identifier by escaping it with another double-quote.

In the following example, a stream is created with an unquoted identifier, which is converted to upper case before the stream definition is stored in the catalog. It can be referenced using its upper-case name, or by an unquoted identifier which is implicitly converted to upper case.

```
-- Create a stream. Stream name specified without quotes,
-- which defaults to uppercase.
CREATE OR REPLACE STREAM ExampleStream (col1 VARCHAR(4));

– example 1: OK, stream name interpreted as uppercase.
CREATE OR REPLACE PUMP "STREAM_PUMP" AS INSERT INTO ExampleStream
    SELECT * FROM SOURCE_SQL_STRAM_001;
```

```
– example 2: OK, stream name interpreted as uppercase.
CREATE OR REPLACE PUMP "STREAM_PUMP" AS INSERT INTO examplestream
    SELECT * FROM   customerdata;

– example 3: Ok.
CREATE OR REPLACE PUMP "STREAM_PUMP" AS INSERT INTO EXAMPLESTREAM
    SELECT * FROM   customerdata;

– example 2: Not found. Quoted names are case-sensitive.
CREATE OR REPLACE PUMP "STREAM_PUMP" AS INSERT INTO "examplestream"
    SELECT * FROM   customerdata;
```

When objects are created in Amazon Kinesis Analytics, their names are implicitly quoted, so it is easy to create identifiers that contain lowercase characters, spaces, dashes, or other punctuation. If you reference those objects in SQL statements, you will need to quote their names.

Reserved Words and Keywords

Certain identifiers, called keywords, have special meaning if they occur in a particular place in a streaming SQL statement. A subset of these key words are called reserved words and may not be used as the name of an object, unless they are quoted. For more information, see Reserved Words and Keywords (p. 163).

Streaming SQL Operators

Subquery Operators

Operators are used in queries and subqueries to combine or test data for various properties, attributes, or relationships.

The available operators are described in the topics that follow, grouped into the following categories:

IN Operator

As an operator in a condition test, IN tests a scalar or row value for membership in a list of values, a relational expression, or a subquery.

```
Examples:
1. --- IF column IN ('A','B','C')
2. --- IF (col1, col2) IN (
    select a, b from my_table
    )
```

Returns TRUE if the value being tested is found in the list, in the result of evaluating the relational expression, or in the rows returned by the subquery; returns FALSE otherwise.

Note
IN has a different meaning and use in CREATE FUNCTION (p. 125).

EXISTS Operator

Tests whether a relational expression returns any rows; returns TRUE if any row is returned, FALSE otherwise.

Scalar Operators

Operator Types

The two general classes of scalar operators are:

- unary: A unary operator operates on only one operand. A unary operator typically appears with its operand in this format:

```
operator operand
```

- binary: A binary operator operates on two operands. A binary operator appears with its operands in this format:

```
operand1 operator operand2
```

A few operators that use a different format are noted specifically in the operand descriptions below.

If an operator is given a null operand, the result is almost always null (see the topic on logical operators for exceptions).

Precedence

Streaming SQL follows the usual precedence of operators:

1. Evaluate bracketed sub-expressions.
2. Evaluate unary operators (e.g., + or -, logical NOT).
3. Evaluate multiplication and divide (* and /).
4. Evaluate addition and subtraction (+ and -) and logical combination (AND and OR).

If one of the operands is NULL, the result is also NULL If the operands are of different but comparable types, the result will be of the type with the greatest precision. If the operands are of the same type, the result will be of the same type as the operands. For instance 5/2 = 2, not 2.5, as 5 and 2 are both integers.

Arithmetic Operators

Operator	Unary/Binary	Description
+	U	Identity
-	U	Negation
+	B	Addition
-	B	Subtraction
*	B	Multiplication
/	B	Division

Each of these operators works according to normal arithmetic behavior, with the following caveats:

1. If one of the operands is NULL, the result is also NULL
2. If the operands are of different but comparable types, the result will be of the type with the greatest precision.
3. If the operands are of the same type, the result will be of the same type as the operands. For instance 5/2 = 2, not 2.5, as 5 and 2 are both integers.

Examples

Operation	Result
1 + 1	2
2.0 + 2.0	4.0
3.0 + 2	5.0
5 / 2	2
5.0 / 2	2.500000000000
5*2+2	12

String Operators

You can use string operators for streaming SQL, including concatenation and string pattern comparison, to combine and compare strings.

Operator	Unary/Binary	Description	Notes
||	B	Concatenation	Also applies to binary types
LIKE	B	String pattern comparison	<string> LIKE <like pattern> [ESCAPE <escape character>]
SIMILAR TO	B	String pattern comparison	<string> SIMILAR TO <similar to pattern> [ESCAPE <escape character>]

Concatenation

This operator is used to concatenate one or more strings as shown in the following table.

Operation	Result
'SQL'||'stream'	Amazon Kinesis Analytics
'SQL'||''||'stream'	Amazon Kinesis Analytics
'SQL'||'stream'||' Incorporated'	Amazon Kinesis Analytics Incorporated
<col1>||<col2>||<col3>||<col4>	<col1><col2><col3><col4>

LIKE patterns

LIKE compares a string to a string pattern. In the pattern, the characters _ (underscore) and % (percent) have special meaning.

Character in pattern	Effect
_	Matches any single character
%	Matches any substring, including the empty string
<any other character>	Matches only the exact same character

If either operand is NULL, the result of the LIKE operation is UNKNOWN.

To explicitly match a special character in the character string, you must specify an escape character using the ESCAPE clause. The escape character must then precede the special character in the pattern. The following table lists examples.

Operation	Result
'a' LIKE 'a'	TRUE
'a' LIKE 'A'	FALSE
'a' LIKE 'b'	FALSE
'ab' LIKE 'a_'	TRUE
'ab' LIKE 'a%'	TRUE
'ab' LIKE 'a_' ESCAPE '\'	FALSE
'ab' LIKE 'a\%' ESCAPE '\'	FALSE
'a_' LIKE 'a_' ESCAPE '\'	TRUE
'a%' LIKE 'a\%' ESCAPE '\'	TRUE
'a' LIKE 'a_'	FALSE
'a' LIKE 'a%'	TRUE
'abcd' LIKE 'a_'	FALSE
'abcd' LIKE 'a%'	TRUE

Operation	Result
'' LIKE ''	TRUE
'1a' LIKE '_a'	TRUE
'123aXYZ' LIKE '%a%'	TRUE
'123aXYZ' LIKE '_%_a%_'	TRUE

SIMILAR TO patterns

SIMILAR TO compares a string to a pattern. It is much like the LIKE operator, but more powerful, as the patterns are regular expressions.

In the following SIMILAR TO table, *seq* means any sequence of characters explicitly specified, such as '13aq'. Non-alphanumeric characters intended for matching must be preceded by an escape character explicitly declared in the SIMILAR TO statement, such as '13aq\!' SIMILAR TO '13aq\!24br\!% ESCAPE '\' (This statement is TRUE).

When a range is indicated, as when a dash is used in a pattern, the current collating sequence is used. Typical ranges are 0-9 and a-z. PostgreSQL provides a typical discussion of pattern-matching, including ranges.

When a line requires multiple comparisons, the innermost pattern that can be matched will be matched first, then the "next-innermost," etc.

Expressions and matching operations that are enclosed within parentheses are evaluated before surrounding operations are applied, again by innermost-first precedence.

Delimiter	Character in pattern	Effect	Rule ID
parentheses ( )	(seq)	Groups the *seq* (used for defining precedence of pattern expressions)	1
brackets [ ]	[seq]	Matches any single character in the seq	2
caret or circumflex	[^seq]	Matches any single character not in the seq	3
	[seq ^ seq]	Matches any single character in seq and not in seq	4
dash	<character1>-<character2>	Specifies a range of characters between character1 and character2 (using some known sequence like 1-9 or a-z)	5
bar	[seq seq]	Matches either seq or seq	6
asterisk	seq*	Matches zero or more repetitions of seq	7

Delimiter	Character in pattern	Effect	Rule ID
plus	seq+	Matches one or more repetitions of seq	8
braces	seq{<number>}	Matches exactly number repetitions of seq	9
	seq{<low number>,<high number>}	Matches low number or more repetitions of seq, to a maximum of high number	10
question-mark	seq?	Matches zero or one instances of seq	11
underscore	_	Matches any single character	12
percent	%	Matches any substring, including the empty string	13
character	<any other character>	Matches only the exact same character	14
NULL	NULL	If either operand is NULL, the result of the SIMILAR TO operation is UNKNOWN.	15
Non-alphanumeric	Special characters	To explicitly match a special character in the character string, that special character must be preceded by an escape character defined using an ESCAPE clause specified at the end of the pattern.	16

The following table lists examples.

Operation	Result	Rule
'a' SIMILAR TO 'a'	TRUE	14
'a' SIMILAR TO 'A'	FALSE	14
'a' SIMILAR TO 'b'	FALSE	14
'ab' SIMILAR TO 'a_'	TRUE	12
'ab' SIMILAR TO 'a%'	TRUE	13
'a' SIMILAR TO 'a_'	FALSE	12 & 14

Operation	Result	Rule
'a' SIMILAR TO 'a%'	TRUE	13
'abcd' SIMILAR TO 'a_'	FALSE	12
'abcd' SIMILAR TO 'a%'	TRUE	13
'' SIMILAR TO ''	TRUE	14
'1a' SIMILAR TO '_a'	TRUE	12
'123aXYZ' SIMILAR TO ''	TRUE	14
'123aXYZ' SIMILAR TO '_%_a %_'	TRUE	13 & 12
'xy' SIMILAR TO '(xy)'	TRUE	1
'abd' SIMILAR TO '[ab][bcde]d'	TRUE	2
'bdd' SIMILAR TO '[ab][bcde]d'	TRUE	2
'abd' SIMILAR TO '[ab]d'	FALSE	2
'cd' SIMILAR TO '[a-e]d'	TRUE	2
'cd' SIMILAR TO '[a-e^c]d'	FALSE	4
'cd' SIMILAR TO '[^(a-e)]d'	INVALID	
'yd' SIMILAR TO '[^(a-e)]d'	INVALID	
'amy' SIMILAR TO 'amyfred'	TRUE	6
'fred' SIMILAR TO 'amyfred'	TRUE	6
'mike' SIMILAR TO 'amyfred'	FALSE	6
'acd' SIMILAR TO 'ab*c+d'	TRUE	7 & 8
'accccd' SIMILAR TO 'ab*c+d'	TRUE	7 & 8
'abd' SIMILAR TO 'ab*c+d'	FALSE	7 & 8
'aabc' SIMILAR TO 'ab*c+d'	FALSE	
'abb' SIMILAR TO 'a(b{3})'	FALSE	9
'abbb' SIMILAR TO 'a(b{3})'	TRUE	9
'abbbbb' SIMILAR TO 'a(b{3})'	FALSE	9
'abbbbb' SIMILAR TO 'ab{3,6}'	TRUE	10
'abbbbbbbb' SIMILAR TO 'ab{3,6}'	FALSE	10
'' SIMILAR TO 'ab?'	FALSE	11
'' SIMILAR TO '(ab)?'	TRUE	11
'a' SIMILAR TO 'ab?'	TRUE	11

Operation	Result	Rule
'a' SIMILAR TO '(ab)?'	FALSE	11
'a' SIMILAR TO 'a(b?)'	TRUE	11
'ab' SIMILAR TO 'ab?'	TRUE	11
'ab' SIMILAR TO 'a(b?)'	TRUE	11
'abb' SIMILAR TO 'ab?'	FALSE	11
'ab' SIMILAR TO 'a_' ESCAPE '\'	FALSE	16
'ab' SIMILAR TO 'a\%' ESCAPE '\'	FALSE	16
'a_' SIMILAR TO 'a_' ESCAPE '\'	TRUE	16
'a%' SIMILAR TO 'a\%' ESCAPE '\'	TRUE	16
'a(b{3})' SIMILAR TO 'a(b{3})'	FALSE	16
'a(b{3})' SIMILAR TO 'a\(b\{3\}\)' ESCAPE '\'	TRUE	16

Date, Timestamp, and Interval Operators

The arithmetic operators +, -, *, and / are binary operators.

Operator	Description	Notes
+	Addition	interval + interval = interval interval + datetime = datetime datetime + interval = datetime
-	Subtraction	interval - interval = interval datetime - interval = datetime (<datetime> - <datetime>) Date, Timestamp, and Interval Operators (p. 18) <interval qualifier> = interval
*	Multiplication	interval * numeric = interval numeric * interval = interval
/	Division	interval / numeric = interval

Examples

Example	Operation	Result
1	INTERVAL '1' DAY + INTERVAL '3' DAY	INTERVAL '4' DAY
2	INTERVAL '1' DAY + INTERVAL '3 4' DAY TO HOUR	INTERVAL '+4 04' DAY TO HOUR
3	INTERVAL '1' DAY - INTERVAL '3 4' DAY TO HOUR	INTERVAL '-2 04' DAY TO HOUR
4	INTERVAL '1' YEAR + INTERVAL '3-4' YEAR TO MONTH	INTERVAL '+4-04' YEAR TO MONTH
5	2 * INTERVAL '3 4' DAY TO HOUR	INTERVAL '6 8' DAY TO HOUR
6	INTERVAL '3 4' DAY TO HOUR / 2	INTERVAL ' 1 14' DAY TO HOUR

In the example 3, '3 4 DAY means 3 days and 4 hours, so the result in that row means 24 hours minus 76 hours, resulting in minus 52 hours, which is a negative 2 days and 4 hours.

Example 4 uses TO MONTH rather than TO HOUR, so the INTERVAL specified as '3-4' means 3 years and 4 months, or 40 months.

In example 6, the "/2" applies to the INTERVAL '3 4', which is 76 hours, half of which is 38, or 1 day and 14 hours.

Further Examples of Interval Operations

Streaming SQL also supports subtracting two datetimes, giving an interval. You specify what kind of interval you want for the result, as shown following:

```
(<datetime> - <datetime>) <interval qualifier>
```

The following examples show operations that can be useful in Amazon Kinesis Analytics applications.

Example 1 – Time Difference (as minutes to the nearest second or as seconds)

```
 values cast ((time  '12:03:34' - time '11:57:23') minute to second as
 varchar(8));
+---------+
 EXPR$0
+---------+
 +6:11
+---------+
1 row selected
.............. 6 minutes, 11 seconds
or
 values cast ((time  '12:03:34' - time '11:57:23') second as varchar(8));
+---------+
 EXPR$0
+---------+
 +371
+---------+
1 row selected
```

Example 2 – Time Difference (as minutes only)

```
values cast ((time  '12:03:34' - time '11:57:23') minute as varchar(8));
+---------+
 EXPR$0
+---------+
 +6
+---------+
1 row selected
.............. 6 minutes; seconds ignored.
values cast ((time  '12:03:23' - time '11:57:23') minute as varchar(8));
+---------+
 EXPR$0
+---------+
 +6
+---------+
1 row selected
.............. 6 minutes
```

Example 3 – Time-to-Timestamp Difference (as days to the nearest second) Invalid

```
values cast ((time '12:03:34'-timestamp '2004-04-29 11:57:23') day to second
 as varchar(8));
Error: From line 1, column 14 to line 1, column 79: Parameters must be of the
 same type
```

Example 4 – Timestamp difference (as days to the nearest second)

```
values cast ((timestamp  '2004-05-01 12:03:34' - timestamp '2004-04-29
 11:57:23') day to
                second as varchar(8));
+-----------+
  EXPR$0
+-----------+
 +2 00:06
+-----------+
1 row selected
.............. 2 days, 6 minutes
.............. Although "second" was specified above, the varchar(8) happens
 to allow
only room enough to show only the minutes, not the seconds.
The example below expands to varchar(11), showing the full result:
values cast ((timestamp  '2004-05-01 12:03:34' - timestamp '2004-04-29
 11:57:23') day to
                second as varchar(11));
+--------------+
    EXPR$0
+--------------+
 +2 00:06:11
+--------------+
1 row selected
.............. 2 days, 6 minutes, 11 seconds
```

Example 5 – Timestamp Difference (as days to the nearest second)

```
values cast ((timestamp  '2004-05-01 1:03:34' - timestamp '2004-04-29
 11:57:23') day to
                second as varchar(11));
+--------------+
    EXPR$0
+--------------+
 +1 13:06:11
+--------------+
1 row selected
.............. 1 day, 13 hours, 6 minutes, 11 seconds
values cast ((timestamp  '2004-05-01 13:03:34' - timestamp '2004-04-29
 11:57:23') day to
                second as varchar(11));
+--------------+
    EXPR$0
+--------------+
 +2 01:06:11
+--------------+
1 row selected
.............. 2 days, 1 hour, 6 minutes, 11 seconds
```

Example 6 – Timestamp Difference (as days)

```
values cast ((timestamp  '2004-05-01 12:03:34' - timestamp '2004-04-29
 11:57:23') day
               as varchar(8));
+---------+
 EXPR$0
+---------+
 +2
+---------+
1 row selected
.............. 2 days
```

Example 7 – Time Difference (as days)

```
values cast ((date '2004-12-02 ' - date '2003-12-01 ') day  as varchar(8));
Error: Illegal DATE literal '2004-12-02 ': not in format 'yyyy-MM-dd'
.............. Both date literals end with a space;  disallowed.
values cast ((date '2004-12-02' - date '2003-12-01 ') day  as varchar(8));
Error: Illegal DATE literal '2003-12-01 ': not in format 'yyyy-MM-dd'
.............. Second date literal still ends with a space;  disallowed.
values cast ((date '2004-12-02' - date '2003-12-01') day  as varchar(8));
+---------+
 EXPR$0
+---------+
 +367
+---------+
1 row selected
.............. 367 days
```

Example 8 – Not Supported (Simple Difference of Dates)

If you don't specify "day" as the intended unit, as shown following, the subtraction is not supported.

```
    values cast ((date '2004-12-02' - date '2003-12-01') as varchar(8));
    Error: From line 1, column 15 to line 1, column 51:
           Cannot apply '-' to arguments of type '<DATE> - <DATE>'.
    Supported form(s): '<NUMERIC> - <NUMERIC>'
                       '<DATETIME_INTERVAL> - <DATETIME_INTERVAL>'
                       '<DATETIME> - <DATETIME_INTERVAL>'
```

Why Use "as varchar" in Conversion Examples?

The reason for using the "values cast (<expression> AS varchar(N))" syntax in the examples above is that while the SQLline client used above (with Amazon Kinesis Analytics running) does return an interval, JDBC does not support returning that result so as to display it. Therefore, that "values" syntax is used to see/show it.

If you close the Amazon Kinesis Analytics (with a !kill command) or if you don't start it before running SQLline, then you can run the sqllineEngine (rather than the sqllineClient) from the bin subdirectory of your Amazon Kinesis Analytics home, which can show your results without the Amazon Kinesis Analytics application or JDBC:

Rules for Specifying Intervals

A Day-Time Interval Literal is a string that denotes a single interval value: for example '10' SECONDS. Note it has two parts: the value (which must always be in single-quotes) and the qualifier (here, SECONDS), which give the units for the value.

The qualifier takes the following form:

```
DAY  HOUR  MINUTE  SECOND [TO HOUR  MINUTE  SECOND]
```

Note
YEAR TO MONTH intervals require a dash separating the values, whereas DAY TO HOUR intervals use a space to separate the values, as seen in the 2nd, 3rd, 5th, and 6th examples in that topic.

In addition, the leading term has to be of greater significance than the optional trailing term, so this means you can only specify:

```
DAY
HOUR
MINUTE
SECOND
DAY TO HOUR
DAY TO MINUTE
DAY TO SECOND
HOUR TO MINUTE
HOUR TO SECOND
MINUTE TO SECOND
```

The easiest way to understand these may be to translate X TO Y as "Xs to the nearest Y". Hence, DAY TO HOUR is "days to the nearest hour".

When DAY, HOUR, or MINUTE is the leading term, you can specify a precision, e.g., DAY(3) TO HOUR, indicating the number of digits the associated field in the value can have. The maximum precision is 10, and the default is 2. You can't specify precision for HOUR, OR MINUTE in the trailing term - they are always of precision 2. So for example, HOUR(3) TO MINUTE is legal, HOUR TO MINUTE(3) is not.

SECOND can also take a precision, but the way it is specified differs depending on whether it is the leading or trailing field.

- If SECOND is the leading field, you can specify the digits before and after the decimal point. For example, SECOND(3,3) would allow you to specify up to 999.999 seconds. The default is (2,3), which is actually a deviation from the SQL:2008 spec (it should be (2,6), but we only have millisecond precision).
- If SECOND is the trailing field, you can only specify precision for the fractional seconds, that is, the part shown after the seconds' decimal point below. For example, SECOND(3) would indicate milliseconds. The default is 3 digits after the decimal point, but as above this is a deviation from the standard of 6.

As for the value, it takes the general form of:

```
[+-]'[+-]DD HH:MM:SS.SSS'
```

Where DD are digits indicating days, HH hours, MM minutes, and SS.SSS is seconds (adjust the number of digits appropriately if precision is explicitly specified).

Not all values have to include all fields—you can trim from both front or back, but not from in the middle. So you could make it 'DD HH' or 'MM:SS.SSS', but not 'DD MM'.

However you write it, though, the value must match the qualifier, as shown following:

```
INTERVAL '25 3' DAY to HOUR ------> legal
INTERVAL '3:45:04.0' DAY TO HOUR --> illegal
```

As stated in the SQL spec, if the precision is not explicitly specified, it is implied to be 2. Thus:

- INTERVAL '120' MINUTE is an illegal interval. The legal form for the desired interval is INTERVAL '120' MINUTE(2)

  and

- INTERVAL '120' SECOND is not legal. The legal form for the desired interval is INTERVAL '120' SECOND(3).

```
  values INTERVAL '120' MINUTE(2);
  Error: From line 1, column 8 to line 1, column 31:
                      Interval field value 120 exceeds precision of
 MINUTE(2) field
  values INTERVAL '120' MINUTE(3);
  Conversion not supported
```

Also, if HOUR, MINUTE, or SECOND are not the leading field, they must fall in the following ranges (taken from Table 6 in topic 4.6.3 of the SQL:2008 foundation spec), as shown following:

```
 HOUR: 0-23
 MINUTE: 0-59
 SECOND: 0-59.999
```

Year-month intervals are similar, except that the qualifiers are as shown following:

```
 YEAR
 MONTH
 YEAR TO MONTH
```

Precision can be specified just as with DAY and HOUR, and the max of 10 and default of 2 is the same.

The value format for year-month is: 'YY-MM'. If MONTH is the trailing field, it must fall in the range 0-11.

```
<interval qualifier> := <start field> TO <end field>  <single datetime field>

<start field> := <non-second primary datetime field> [ <left paren> <interval
 leading field precision> <right paren> ]

<end field> := <non-second primary datetime field>  SECOND [ <left paren>
 <interval fractional seconds precision> <right paren> ]

<single datetime field> := <non-second primary datetime field> [ <left paren>
 <interval leading field precision> <right paren> ]
```

```
   SECOND [ <left paren> <interval leading field precision>
            [ <comma> <interval fractional seconds precision> ] <right
 paren> ]
<primary datetime field> := <non-second primary datetime field>       SECOND
<non-second primary datetime field> := YEAR  MONTH  DAY  HOUR      MINUTE
<interval fractional seconds precision> := <unsigned integer>
<interval leading field precision> := <unsigned integer>
```

Logical Operators

Logical operators let you establish conditions and test their results.

Operator	Unary/Binary	Description	Operands
NOT	U	Logical negation	Boolean
AND	B	Conjunction	Boolean
OR	B	Disjunction	Boolean
IS	B	Logical assertion	Boolean
IS NOT UNKNOWN	U	Negated unknown comparison: <expr> IS NOT UNKNOWN	Boolean
IS NULL	U	Null comparison: <expr> IS NULL	Any
IS NOT NULL	U	Negated null comparison: <expr> IS NOT NULL	Any
=	B	Equality	Any
!=	B	Inequality	Any
<>	B	Inequality	Any
>	B	Greater than	Ordered types (Numeric, String, Date, Time)
>=	B	Greater than or equal to (not less than)	Ordered types
<	B	Less than	Ordered types
<=	B	Less than or equal to (not more than)	Ordered types
BETWEEN	Ternary	Range comparison: col1 BETWEEN expr1 AND expr2	Ordered types
IS DISTINCT FROM	B	Distinction	Any

Operator	Unary/Binary	Description	Operands
IS NOT DISTINCT FROM	B	Negated distinction	Any

Three State Boolean Logic

SQL boolean values have three possible states rather than the usual two: TRUE, FALSE, and UNKNOWN, the last of which is equivalent to a boolean NULL. TRUE and FALSE operands generally function according to normal two-state boolean logic, but additional rules apply when pairing them with UNKNOWN operands, as the tables that follow will show.

> **Note**
> UNKOWN represents "maybe TRUE, maybe FALSE" or, to put it another way, "not definitely TRUE and not definitely FALSE." This understanding may help you clarify why some of the expressions in the tables evaluate as they do.

Negation (NOT)

Operation	Result
NOT TRUE	FALSE
NOT FALSE	TRUE
NOT UNKNOWN	UNKNOWN

Conjunction (AND)

Operation	Result
TRUE AND TRUE	TRUE
TRUE AND FALSE	FALSE
TRUE AND UNKNOWN	UNKNOWN
FALSE AND TRUE	FALSE
FALSE AND FALSE	FALSE
FALSE AND UNKNOWN	FALSE
UNKNOWN AND TRUE	UNKNOWN
UNKNOWN AND FALSE	FALSE
UNKNOWN AND UNKNOWN	UNKNOWN

Disjunction (OR)

Operation	Result
TRUE OR TRUE	TRUE
TRUE OR FALSE	TRUE
TRUE OR UNKNOWN	TRUE

Operation	Result
FALSE OR TRUE	TRUE
FALSE OR FALSE	FALSE
FALSE OR UNKNOWN	UNKNOWN
UNKNOWN OR TRUE	TRUE
UNKNOWN OR FALSE	UNKNOWN
UNKNOWN OR UNKNOWN	UNKNOWN

Assertion (IS)

Operation	Result
TRUE IS TRUE	TRUE
TRUE IS FALSE	FALSE
TRUE IS UNKNOWN	FALSE
FALSE IS TRUE	FALSE
FALSE IS FALSE	TRUE
FALSE IS UNKNOWN	FALSE
UNKNOWN IS TRUE	FALSE
UNKNOWN IS FALSE	FALSE
UNKNOWN IS UNKNOWN	TRUE

IS NOT UNKNOWN

Operation	Result
TRUE IS NOT UNKNOWN	TRUE
FALSE IS NOT UNKNOWN	TRUE
UNKNOWN IS NOT UNKNOWN	FALSE

IS NOT UNKNOWN is a special operator in and of itself. The expression "x IS NOT UNKNOWN" is equivalent to "(x IS TRUE) OR (x IS FALSE)", not "x IS (NOT UNKNOWN)". Thus, substituting in the table above:

x	Operation	Result		Result of substituting for x in "(x IS TRUE) OR (x IS FALSE)"
TRUE	TRUE IS NOT UNKNOWN	TRUE	becomes	"(TRUE IS TRUE) OR (TRUE IS

x	Operation	Result		Result of substituting for x in "(x IS TRUE) OR (x IS FALSE)"
				FALSE)" -- hence TRUE
FALSE	FALSE IS NOT UNKNOWN	TRUE	becomes	"(FALSE IS TRUE) OR (FALSE IS FALSE)" -- hence TRUE
UNKNOWN	UNKNOWN IS NOT UNKNOWN	FALSE	becomes	"(UNKNOWN IS TRUE) OR (UNKNOWN IS FALSE)" -- hence FALSE, since UNKNOWN is neither TRUE not FALSE

Since IS NOT UNKNOWN is a special operator, the operations above are not transitive around the word IS:

Operation	Result
NOT UNKNOWN IS TRUE	FALSE
NOT UNKNOWN IS FALSE	FALSE
NOT UNKNOWN IS UNKNOWN	TRUE

IS NULL and IS NOT NULL

Operation	Result
UNKNOWN IS NULL	TRUE
UNKNOWN IS NOT NULL	FALSE
NULL IS NULL	TRUE
NULL IS NOT NULL	FALSE

IS DISTINCT FROM and IS NOT DISTINCT FROM

Operation	Result
UNKNOWN IS DISTINCT FROM TRUE	TRUE
UNKNOWN IS DISTINCT FROM FALSE	TRUE
UNKNOWN IS DISTINCT FROM UNKNOWN	FALSE

Operation	Result
UNKNOWN IS NOT DISTINCT FROM TRUE	FALSE
UNKNOWN IS NOT DISTINCT FROM FALSE	FALSE
UNKNOWN IS NOT DISTINCT FROM UNKNOWN	TRUE

Informally, "x IS DISTINCT FROM y" is similar to "x <> y", except that it is true even when either x or y (but not both) is NULL. DISTINCT FROM is the opposite of identical, whose usual meaning is that a value (true, false, or unknown) is identical to itself, and distinct from every other value. The IS and IS NOT operators treat UNKOWN in a special way, because it represents "maybe TRUE, maybe FALSE".

Other Logical Operators

For all other operators, passing a NULL or UNKNOWN operand will cause the result to be UNKNOWN (which is the same as NULL).

Examples

Operation	Result
TRUE AND CAST(NULL AS BOOLEAN)	UNKNOWN
FALSE AND CAST(NULL AS BOOLEAN)	FALSE
1 > 2	FALSE
1 < 2	TRUE
'foo' = 'bar'	FALSE
'foo' <> 'bar'	TRUE
'foo' <= 'bar'	FALSE
'foo' <= 'bar'	TRUE
3 BETWEEN 1 AND 5	TRUE
1 BETWEEN 3 AND 5	FALSE
3 BETWEEN 3 AND 5	TRUE
5 BETWEEN 3 AND 5	TRUE
1 IS DISTINCT FROM 1.0	FALSE
CAST(NULL AS INTEGER) IS NOT DISTINCT FROM CAST (NULL AS INTEGER)	TRUE

Functions

The topics in this section describe functions supported by streaming SQL.

Topics

Standard Functions

The tables in this section describe the standard functions for Amazon Kinesis Analytics streaming SQL.

Topics

Datetime Conversion Functions

You specify date and time formats using patterned letters. Date and time pattern strings use unquoted letters from 'A' to 'Z' and from 'a' to 'z', with each letter representing a formatting element.

For more information, see Class SimpleDateFormat on the Oracle website.

Note
If you include other characters, they will be incorporated into the output string during formatting or compared to the input string during parsing.

The pattern letters in the following table are defined (all other characters from 'A' to 'Z' and from 'a' to 'z' are reserved).

Letter	Date or Time Component	Presentation	Examples
y	Year	Year	yyyy; yy 2018;18
Y	Week year	Year	YYYY; YY 2009; 09
M	Month in year	Month	MMM;MM;MM July; Jul; 07
w	Week in year	Number	ww; 27
W	Week in month	Number	W 2

Letter	Date or Time Component	Presentation	Examples
D	Day in year	Number	DDD 321
d	Day in month	Number	dd 10
F	Day of week in month	Number	F 2
E	Day name in week	Text	Tuesday; Tue
u	Day number of week (1 = Monday, ..., 7 = Sunday)	Number	1
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800
X	Time zone	ISO 8601 time zone	-08; -0800; -08:00

You determine the exact presentation by repeating pattern letters, along the lines of YYYY.

Text

If the number of repeated pattern letters is 4 or more, the full form is used; otherwise a short or abbreviated form is used if available. For parsing, both forms are accepted, independent of the number of pattern letters.

Number

For formatting, the number of pattern letters is the minimum number of digits, and shorter numbers are zero-padded to this amount. For parsing, the number of pattern letters is ignored unless it's needed to separate two adjacent fields.

Year

If the formatter's Calendar is the Gregorian calendar, the following rules are applied.

- For formatting, if the number of pattern letters is 2, the year is truncated to 2 digits; otherwise it is interpreted as a number.
- For parsing, if the number of pattern letters is more than 2, the year is interpreted literally, regardless of the number of digits. So using the pattern "MM/dd/yyyy", "01/11/12" parses to Jan 11, 12 A.D.

For parsing with the abbreviated year pattern ("y" or "yy"), SimpleDateFormat must interpret the abbreviated year relative to some century. It does this by adjusting dates to be within 80 years before and 20 years after the time the SimpleDateFormat instance is created. For example, using a pattern of "MM/dd/yy" and a SimpleDateFormat instance created on Jan 1, 2018, the string "01/11/12" would be interpreted as Jan 11, 2012 while the string "05/04/64" would be interpreted as May 4, 1964. During parsing, only strings consisting of exactly two digits, as defined by Character.isDigit(char), will be parsed into the default century. Any other numeric string, such as a one digit string, a three or more digit string, or a two digit string that isn't all digits (for example, "-1"), is interpreted literally. So "01/02/3" or "01/02/003" are parsed, using the same pattern, as Jan 2, 3 AD. Likewise, "01/02/-3" is parsed as Jan 2, 4 BC.

Otherwise, calendar system specific forms are applied. For both formatting and parsing, if the number of pattern letters is 4 or more, a calendar specific long form is used. Otherwise, a calendar specific short or abbreviated form is used.

Char To Timestamp(Sys)

The Char to Timestamp function is one of the most frequently-used system functions, because it lets you create a timestamp out of any correctly formatted input string. Using this function, you can specify which parts of the timestamp string you wish to use in subsequent processing, and create a TIMESTAMP value containing only those. To do so, you specify a template that identifies the parts of the timestamp you want. For example, to use only year and month, you would specify 'yyyy-MM'.

The input date-time string can contain any parts of a full timestamp ('yyyy-MM-dd hh:mm:ss'). If all these elements are present in your input string, and 'yyyy-MM-dd hh:mm:ss' is the template you supply, then the input-string elements are interpreted in that order as year, month, day, hour, minute, and seconds, such as in '2009-09-16 03:15:24'. The yyyy cannot be uppercase; the hh can be uppercase to mean using a 24-hour clock.

For the full range of valid specifiers, see Class SimpleDateFormat on the Oracle website.

CHAR_TO_TIMESTAMP uses the template you specify as a parameter in the function call. The template causes the TIMESTAMP result to use only the parts of the input-date-time value that you specified in the template. Those fields in the resulting TIMESTAMP contain the corresponding data taken from your input-date-time string. Fields not specified in your template will use default values (see below). The format of the template used by CHAR_TO_TIMESTAMP is defined by the Class SimpleDateFormat on the Oracle website. For more information, see Date and Time Patterns (p. 107).

The function-call syntax is as follows:

```
CHAR_TO_TIMESTAMP('<format_string>','<input_date_time_string>')
```

Where <format_ string> is the template you specify for the parts of <date_time_string> you want, and <input_date_time_string> is the original string that is being converted to a TIMESTAMP result.

Note that each string must be enclosed in single quotes and each element of the <input_date_time_string> must be in the range for its corresponding element in the template, otherwise no result is returned.

For example, the input-string-element whose position corresponds with MM must be an integer from 1 to 12, because anything else does not represent a valid month. Similarly, the input-string-element whose position corresponds with dd must be an integer from 1 to 31, because anything else does not represent a valid day. (However, if MM is 2, dd cannot be 30 or 31, because February never has such days.)

For hours, minutes, or seconds, the default starting value is zero, so when those specifiers are omitted from the template, zeroes are substituted. For months or days, the default starting value substituted for the omitted parts is 01.

For example, using '2009-09-16 03:15:24' as your input string, you can obtain a TIMESTAMP containing only the date, with zeros for the other fields such as hours, minutes, or seconds.

```
CHAR_TO_TIMESTAMP('yyyy-MM-dd','2009-09-16 03:15:24').
```

The result would is TIMESTAMP 2009-09-16 00:00:00.

If the call had kept hours and minutes in the template while omitting months, days, and seconds, as illustrated in the following call.

```
--- --- CHAR_TO_TIMESTAMP('yyyy-hh-mm','2009-09-16 03:15:24')
```

Then, the resulting TIMESTAMP would be 2009-01-01 03:15:00.

Template Strings to Create Specific Output Timestamps (p. 35) shows further illustrative examples of templates and input strings used to create the indicated output TIMESTAMPs.

Note
Input string MUST use the form 'yyyy-MM-dd hh:mm:ss' or a subset or reordering thereof. As a result, using an input string like 'Wednesday, 16 September 2009 03:15:24' will NOT work, meaning that no output will result.

About Delimiters and Values

Delimiters in the template must match those in the input string and values in the input string must be acceptable for the template specifiers to which they correspond.

As a general convention, a colon is used to separate hours from minutes, and minutes from seconds. Similarly, the general convention is to use a dash or slash to separate years from months and months from days.

For example, the following template has values that line up correctly with the input string.

```
values (CHAR_TO_TIMESTAMP('MM/dd/yy hh:mm:ss','09/16/11 03:15:24') );
'EXPR$0'
'2011-09-16 03:15:24'
1 row selected
```

If values in the input string are not acceptable for the template specifiers to which they correspond, the result fails, as in the following example.

```
values (CHAR_TO_TIMESTAMP('MM/dd/yy hh:mm:ss','2009/09/16 03:15:24') );
'EXPR$0'
No rows selected
```

This example returns no rows because 2009 is not an acceptable value for months, which is the first specifier (MM) in the template.

Omissions in the supplied string can cause the template value 'yyyy' to produce logical but unintended or unexpected results. The following examples each return an erroneous year, but one that derives directly from the first element in the supplied string.

```
 VALUES(CHAR_TO_TIMESTAMP('yyyy','09-16 03:15'));
'EXPR$0'
'0009-01-01 00:00:00'
1 row selected
VALUES(CHAR_TO_TIMESTAMP('yyyy','16 03:15'));
'EXPR$0'
'0016-01-01 00:00:00'
```

```
1 row selected
```

Examples Using Templates to Create TIMESTAMPS

The order of the template must match the input string. That means that you cannot specify "hh" after "yyyy" and expect the method to find the hour automatically. For example, the following template specifies years first, then hours, then minutes, and returns an erroneous result.

```
 values (CHAR_TO_TIMESTAMP('yyyy-hh-mm','2009-09-16 03:15:24'));
'EXPR$0'
'2009-01-01 09:16:00'
1 row selected
```

Since the specifiers for months and days are not present in the template, their values in the input string were ignored, with 01 substituted for both values in the output TIMESTAMP. The template specified hours and minutes as the second and third input values, so 09 became the hours and 16 became the minutes. No specifier was present for seconds, so 00 was used.

The years specifier can be alone or after a delimiter matching the input string shows the end of the years specifier, with one of the hours:minutes:seconds specifiers.

```
values (CHAR_TO_TIMESTAMP('yyyy','2009-09-16 03:15:24') );
'EXPR$0'
'2009-01-01 00:00:00'
1 row selected
```

In contrast, the template below fails because it has a space-as-delimiter before the "hh" rather than the dash delimiter used in the input string's date specification.

```
  values (CHAR_TO_TIMESTAMP('yyyy hh','2009-09-16 03:15:24') );
  'EXPR$0'
  No rows selected
```

The four templates below work because they use the same delimiter to separate the years specifier from the next specifier as is used in the input string's date specification (dash in the first case, space in the second, slash in the third, and dash in the fourth).

```
values (CHAR_TO_TIMESTAMP('yyyy-hh','2009-09-16 03:15:24') );
'EXPR$0'
'2009-01-01 09:00:00'
1 row selected
values (CHAR_TO_TIMESTAMP('yyyy hh','2009 09 16 03:15:24') );
'EXPR$0'
'2009-01-01 09:00:00'
1 row selected
values (CHAR_TO_TIMESTAMP('yyyy/hh','2009/09/16 03:15:24') );
'EXPR$0'
'2009-01-01 09:00:00'
1 row selected
values (CHAR_TO_TIMESTAMP('yyyy-mm','2009-09-16 03:15:24') );
'EXPR$0'
'2009-01-01 00:09:00'
1 row selected
```

However, if the template specifies months (MM), it cannot then specify hours, minutes, or seconds unless days are also specified.

Template Strings to Create Specific Output Timestamps

Template	Input String	Output TIMESTAMP	Notes
`'yyyy-MM-dd hh:mm:ss'`	`'2009-09-16 03:15:24'`	`'2009-09-16 03:15:24'`	
`'yyyy-mm'`	`'2011-02-08 07:23:19'`	`'2011-01-01 00:02:00'`	The template above specifies only year first and minutes second, so the second element in the input string ("02") is used as minutes. Default values are used for Month and Day ("01") and for hours and seconds ("00").
`'MMM dd, yyyy'`	`'March 7, 2010'`	`'2010-03-07 00:00:00'`	MMM in the template above matches "March"; the template's 'comma space' matches the input string. If the template lacks the comma, so must the input string, or there is no output; If the input string lacks the comma, so must the template.
`'MMM dd,'`	`'March 7, 2010'`	`'1970-03-07 00:00:00'`	Note that the template above doesn't use a year specifier, causing the output TIMESTAMP to use the earliest year in this epoch, 1970.
`'MMM dd,y'`	`'March 7, 2010'`	`'2010-03-07 00:00:00'`	Using the template above, if the input string were 'March 7, 10', the output TIMESTAMP would be '0010-03-07 00:00:00'.
`'M-d'`	`'2-8'`	`'1970-02-08 00:00:00'`	Absent a yyyy specifier in the template, as above, the earliest year in this epoch (1970) is used. An input string of '2–8#2011' would give the same result; using

Template	Input String	Output TIMESTAMP	Notes
			'2011–2#8' would give no result because 2011 is not a valid month.
`'MM-dd-yyyy'`	`'06-23-2011 10:11:12'`	`'2011-06-23 00:00:00'`	Dashes as delimiters (as above) are fine, if template and input both use them in the same positions. Since the template omits hours, minutes, and seconds, zeroes are used in the output TIMESTAMP.
`'dd-MM-yy hh:mm:ss'`	`'23-06-11 10:11:12'`	`'2011-06-23 10:11:12'`	You can have the specifiers in any order as long as that order matches the meaning of the input string you supply. The template and input string of the next example below have the same meaning (and the same output TIMESTAMP) as this example, but they specify months before days and seconds before hours.
`'MM-dd-yy ss:hh:mm'`	`'06-23-11 12:10:11'`	`'2011-06-23 10:11:12'`	In the template used above, the order of the month and day specifiers is reversed from the example just above, and the specifier for seconds is before hours instead of after minutes; but because the input string also puts months before days and seconds before hours, the meaning (and the output TIMESTAMP) is the same as the example ABOVE.

Template	Input String	Output TIMESTAMP	Notes
`'yy-dd-MM ss:hh:mm'`	`'06-23-11 12:10:11'`	`'2006-11-23 10:11:12'`	The template used above reverses (compared to the prior example above) the years and months specifiers, while the input string remains the same. In this case, the output TIMESTAMP uses the first element of the input string as the years, the second as the days, and the third as the months.
`'dd-MM-yy hh:mm'`	`'23-06-11 10:11:12'`	`'2011-06-23 10:11:00'`	With seconds omitted in the template, as above, the output TIMESTAMP uses 00 seconds. Any number of y specifiers produces the same result; but if the input string inadvertently uses a 1 instead of 11 for the year, as in '23-06-1 10:11:12', then the output TIMESTAMP becomes '0001-06-23 10:11:00'.

`'MM/dd/yy hh:mm:ss'`	`'12/19/11 10:11:12'`
	
	`'12/19/11 12:11:12'`
		`'2011-12-19 10:11:12'`
		
		`'12/19/11 00:11:12'`
			Slashes as delimiters are fine, if template and input both use them in the same positions, as above. Using specifier hh, input times of 12:11:10 and 00:11:10 have the same meaning as a time in the morning.

Template	Input String	Output TIMESTAMP	Notes

'MM/dd/yy HH:mm:ss'	'12/19/11 12:59:59'
	
	'12/19/11 21:08:07'
	
	'2011-12-19 00:11:12'
	
	'2011-12-19 12:11:12'
		'2011-12-19 12:59:59'
		
		'2011-12-19 21:08:07'
			The input-string values '2011-12-19 00:11:12' or '2011-12-19 12:11:12' would fail with this template because '2011' is not a month, as required/expected by the template-string 'MM/dd/yy HH:mm:ss'.
			
			However, changing the template gives useful output:
			
			`values(cast(CHAR_TO_TIMESTAMP('y MM/dd HH:mm:ss', '2011/12/19 00:11:12') as varchar(19)));`
			`'EXPR$0'`
			`'2011-12-19 00:11:12'`
			
			1 row selected
			
			'12/19/11 00:11:12' would fail with the above template ('y/MM/dd'), since 19 is not a valid month; supplying '12/11/19 00:11:12' works.
			
			'2011-12-19 12:11:12' would fail as input because dashes don't match the slashes in the template, '2011/12/19 12:11:12' works.
			
			Note that for times after 12 noon (that is, for afternoon and evening times), the hours specifier must be HH instead of hh, and the input string must specify the afternoon or evening hour in 24-

Template	Input String	Output TIMESTAMP	Notes
			hour clock time, hours running from 00 to 23. Using specifier HH, input times of 12:11:10 and 00:11:10 have different meanings, the first as a time in the afternoon and the second as a time in the morning. Using the specifier hh, the times from 12:00 through 11:59:59 are morning times: • Given the specifiers hh:mm:ss, the output TIMESTAMP will include `'00:09:08'` in the morning for both input string '12:09:08' and input string `'00:09:08'`; whereas, • Given the specifiers HH:mm:ss, the output TIMESTAMP for input string `'00:09:08'` in the morning will include `'00:09:08'` and the output TIMESTAMP for input string `'12:09:08'` in the afternoon will include `'12:09:08'`.

CHAR_TO_DATE

Converts a string to a date, according to the specified format string.

```
CHAR_TO_DATE(format,dateString);
```

CHAR_TO_TIME

Converts a string to a date, according to the specified format string

```
CHAR_TO_TIME(format,dateString);
```

DATE_TO_CHAR

The DATE_TO_CHAR converts a date to a string.

```
DATE_TO_CHAR(format,d);
```

Where d is a date that will be converted to a string.

TIME_TO_CHAR

Uses a format string to format a time. Returns the formatted time or portion of a time as a string.

```
TIME_TO_CHAR(format,time);
```

TIMESTAMP_TO_CHAR

Uses a format string to format a timestamp as char. Returns the timestamp as a string.

```
TIMESTAMP_TO_CHAR(format,ts);
```

Where ts is timestamp.

ANY

```
ANY ( <boolean_expression> )
```

ANY returns true if the supplied boolean_expression is true in any of the selected rows. Returns false if the supplied boolean_expression is true in none of the selected rows.

Example

The following SQL snippet returns 'true' if the price for any ticker in the stream of trades is below 1. Returns 'false' if every price in the stream is 1 or greater.

```
SELECT STREAM ANY (price < 1) FROM trades
 GROUP BY (FLOOR trades.rowtime to hour)
```

EVERY

```
EVERY ( <boolean_expression> )
```

EVERY returns true if the supplied boolean_expression is true in all of the selected rows. Returns false if the supplied boolean_expression is false in any of the selected rows.

Example

The following SQL snippet returns 'true' if the price for every ticker in the stream of trades is below 1. Returns 'false' if any price is 1 or greater.

```
SELECT STREAM EVERY (price < 1) FROM trades
 GROUP BY (FLOOR trades.rowtime to hour)
```

EXP_AVG

```
EXP_AVG ( expression, <time-interval> )
```

EXP_AVG returns an exponentially-weighted average (exponential moving average) of a stream of value expressions selected in a specified time window. EXP_AVG divides the specified window into intervals based on the value of <time-interval>. The values of the specified expression are weighted the most heavily for the most recent time-intervals and exponentially less heavily for earlier intervals.

Example

This example creates an exponentially-weighted average of the price of each stock ticker over a 30-second window such that the prices (for that ticker symbol) in the most recent 10-second subwindow carry double the weight of the prices in the middle 10-second subwindow and four times the weight of the prices in the oldest 10-second subwindow.

```
select stream t.rowtime, ticker, price,
exp_avg(price, INTERVAL '10' SECOND) over w as avgPrice
from t
window w as (partition by ticker range interval '30' second preceding);
```

In this example, 10 seconds is the half-life of the decay function, that is, the period over which the weights applied to the prices being averaged decrease by a factor of two. In other words, the older one will be given half as much weight as the newer one. It is specified as the time_interval in the call to EXP_AVG as interval '10' second .

FIRST_VALUE

```
FIRST_VALUE( <value-expression>) <null treatment> OVER <window-specification>
```

FIRST_VALUE returns the evaluation of the <value expression> from the first row that qualifies for the aggregate. FIRST_VALUE requires the OVER clause, and is considered an Analytic Functions (p. 66). FIRST_VALUE has a null treatment option defined in the following table.

Null treatment option	Effect
FIRST_VALUE(x) IGNORE NULLS OVER <window-specification>	Returns first non null value of x in <window-specification>
FIRST_VALUE(x) RESPECT NULLS OVER <window-specification>	Returns first value, including null of x in <window-specification>
FIRST_VALUE(x) OVER <window-specification>	Returns first value, including null of x in <window-specification>

FIXED_COLUMN_LOG_PARSE

Parses fixed-width fields and automatically converts them to the given SQL types.

```
FIXED_COLUMN_LOG_PARSE ( <string value expression>, <column description
 string expression> )
 <column description string expression> := '<column description> [,...]'
 <column description> :=
   <identifier> TYPE <data type> [ NOT NULL ]
```

```
  START <numeric value expression> [FOR <numeric constant expression>]
```

Starting position of column is 0. Column specifications for types DATE,TIME and TIMESTAMP support a format parameter allowing the user to specify exact time component layout. The parser uses the Java class java.lang.SimpleDateFormat to parse the strings for types DATE, TIME and TIMESTAMP. The Date and Time Patterns (p. 107) topic gives a full description and examples of timestamp format strings. The following is an example of a column definition with a format string:

```
"name" TYPE TIMESTAMP 'dd/MMM/yyyy:HH:mm:ss'
```

Related Topics

REGEX_LOG_PARSE (p. 112)

Group Rank

This function applies a RANK() function to logical groups of rows and optionally delivers the group in sorted order.

Applications of Group_Rank include the following:

- To sort results of a streaming GROUP BY.
- To determine a relationship within the results of a group.

Group Rank can do the following actions:

- Apply Rank to a specified input column.
- Supply either sorted or non-sorted output.
- Enable the user to specify a period of inactivity for data flush.

SQL Declarations

The functional attributes and DDL are described in the topics that follow.

Functional attributes for Group_Rank

This function acts as follows:

- Gathers rows until either a rowtime change is detected or a specified idle-time limit is exceeded.
- Accepts any streaming rowset.
- Uses any column with a basic SQL data type of INTEGER, CHAR, VARCHAR as the column by which to do the ranking
- Orders the output rows either in the order received or in ascending or descending order of values in the selected column.

DDL for Group_Rank

```
 group_rank(c cursor, rankByColumnName VARCHAR(128),
   rankOutColumnName VARCHAR(128), sortOrder VARCHAR(10), outputOrder
VARCHAR(10),
   maxIdle INTEGER, outputMax INTEGER)
 returns table(c.*, "groupRank" INTEGER)
```

The parameters to the function are listed in the following table.

c	CURSOR to Streaming Result Set
rankByColumnName	String naming the column to use for ranking the group
rankOutColumnName	String naming the column to use for returning the rank. This string must match the name of the groupRank column in RETURNS clause of the CREATE FUNCTION statement.
sortOrder	Controls ordering of rows for rank assignment. Valid values are: • 'asc' - Ascending based on rank. • 'desc' - Descending based on the rank.
outputOrder	Controls ordering of output. Valid values are: • 'asc' - Ascending based on rank. • 'desc' - Descending based on the rank.
maxIdle	Time limit in milliseconds for holding a group for ranking. When maxIdle expires the current group is released to the stream. A value of zero indicates no idle time out.
outputMax	Maximum number of rows the function will output in a given group. A value of 0 indicates no limit.

Example of Group_Rank Use

```
SELECT STREAM
       pageId, COUNT(*) AS hitCount
    FROM AccessStream AS S
    GROUP BY
       FLOOR(S.ROWTIME TO MINUTE), pageId;
    SELECT STREAM "pageId", "hitCount", "groupRank"
    FROM (TABLE(group_rank(CURSOR(SELECT STREAM "pageId", "hitCount"
       FROM pageCounts1Min),
         'hitCount', 'groupRank', 'desc', 'asc', 10, 5)));
```

Sample Input

```
pageId hitCount
condo  51
auto  25
books  200
CDs  202
```

```
DVDs  1000
Games  500
```

Sample output

```
pageId hitCount groupRank
DVDs  1000  1
Games  500  2
CDs  202  3
books  200  4
Condo  51  5
```

Operational Overview

Rows are buffered from the input cursor for each group (that is, rows with the same rowtimes). Ranking of the rows is done either after the arrival of a row with a different rowtime (or when the idle timeout occurs). Rows continue to be read while ranking is performed on the group of rows with the same rowtime.

The outputMax parameter specifies the maximum number of rows to be returned for each group after ranks are assigned.

By default, group_rank supports column pass through, as the example illustrates by using c.* as the standard shortcut directing pass through of all input columns in the order presented. You can, instead, name a subset using the notation "c.columName", allowing you to reorder the columns. However, using specific column names ties the UDX to a very specific input set while using the c.* notation allows the UDX to handle any input set

The rankOutColumnName parameter specifies the output column used to return ranks. This column name must match the column name specified in the RETURNS clause of the CREATE FUNCTION statement.

LAST_VALUE

```
LAST_VALUE ( <value-expression> )  OVER <window-specification>
```

LAST_VALUE returns the evaluation of the <value expression> from the last row that qualifies for the aggregate.

Null Treatment Option	Effect
LAST_VALUE(x) IGNORE NULLS OVER <window-specification>	Returns last non null value of x in <window-specification>
LAST_VALUE(x) RESPECT NULLS OVER <window-specification>	Returns last value, including null of x in <window-specification>
LAST_VALUE(x) OVER <window-specification>	Returns last value, including null of x in <window-specification>

Monotonic Function

```
MONOTONIC(<expression>)
```

Streaming GROUP BY requires that at least one of the grouped expressions be monotonic and non-constant. The only column known in advance to be monotonic is ROWTIME. For more information, see Monotonic Expressions and Operators (p. 120).

The MONOTONIC function allows you to declare that a given expression is monotonic, enabling a streaming GROUP BY to use that expression as a key.

The MONOTONIC function evaluates its argument and returns the result (as the same type as its argument).

By enclosing an expression in MONOTONIC, you are asserting that values of that expression are either non-increasing or non-decreasing and never change direction. For example, if you have a stream LINEITEMS consisting of the line items of orders, and you wrote MONOTONIC(orderId), you are asserting that line items are consecutive in the stream. It would be OK if there were line items for order 1000, followed by line items for order 1001, followed by line items for order 1005. It would be illegal if there were then a line item for order 1001 (that is, the line item sequence became 1000, 1001, 1005, 1001). Similarly, a line item sequence of 987, 974, 823 would be legal, but the following line item sequences would be illegal:

- 987, 974, 823, 973
- 987, 974, 823, 1056

An expression declared monotonic can decrease, or even have arbitrary order.

For example the strings in following sequence are neither ascending nor descending, since "F" alphabetically precedes the other first letters.

Note that the definition of MONOTONIC is precisely what is needed for GROUP BY to make progress.

If an expression declared monotonic is not monotonic (that is, if the assertion is not valid for the actual data) then Amazon Kinesis Analytics behavior is unspecified.

In other words, if you are certain that an expression is monotonic, you can use this MONOTONIC function to enable Amazon Kinesis Analytics to treat the expression as monotonic.

However, if you are mistaken and the values resulting from evaluating the expression change from ascending to descending or from descending to ascending, unexpected results may arise. Amazon Kinesis Analytics streaming SQL will take you at your word and operate on your assurance that the expression is monotonic. But if in fact it is not monotonic, the resulting Amazon Kinesis Analytics behavior cannot be determined in advance, and so results may not be as expected or desired.

NTH_VALUE

```
NTH_VALUE(x, n) [ <from first or last> ] [ <null treatment> ] over w
```

Where:

<null treatment> := RESPECT NULLS | IGNORE NULL

<from first or last> := FROM FIRST | FROM LAST

NTH_VALUE returns the nth value of x from the first or last value in the window. Default is first. If <null treatment> is set to IGNORE NULLS, then function will skip over nulls while counting.

If there aren't enough rows in the window to reach nth value, the function returns NULL.

SYS_LOG_PARSE

Parses the standard syslog format:

```
Mon DD HH:MM:SS server message
```

SYS_LOG_PARSE processes entries commonly found in UNIX/Linux system logs. System log entries start with a timestamp and are followed with a free form text field. SYS_LOG_PARSE output consists of two columns. The first column is named "COLUMN1" and is SQL data type TIMESTAMP. The second column is named "COLUMN2" and is SQL type VARCHAR().

Note
For more information about SYSLOG, see IETF RFC3164. For more information about date-time patterns and matching, see Date and Time Patterns (p. 107).

VARIABLE_COLUMN_LOG_PARSE

```
VARIABLE_COLUMN_LOG_PARSE(
 <character-expression>, <columns>, <delimiter-string>
 [ , <escape-string>, <quote-string> ] )
 <columns> := <number of columns> | <list of columns>
 <number of columns> := <numeric value expression>
 <list of columns> := '<column description>[, ...]'
 <column description> := <identifier> TYPE <data type> [ NOT NULL ]
 <delimiter string> := <character-expression>
 <escape-string> := <character-expression>
 <quote-string> := '<begin quote character> [ <end quote character> ]'
```

VARIABLE_COLUMN_LOG_PARSE splits an input string (its first argument, <character-expression>) into fields separated by a delimiter character or delimiter string. Thus it handles comma-separated values or tab-separated values. It can be combined with FIXED_COLUMN_LOG_PARSE (p. 41) to handle something like maillog, where some fields are fixed-length and others are variable-length.

Note
Parsing of binary files is not supported.

The arguments <escape-string> and <quote-string> are optional. Specifying an <escape-string> allows the value of a field to contain an embedded delimiter. As a simple example, if the <delimiter-string> specified a comma, and the <escape-string> specified a backslash, then an input of "a,b' would be split into two fields "a" and "b", but an input of "a\,b" would result in a single field "a,b".

Since Amazon Kinesis Analytics supports Expressions and Literals (p. 118), a tab can also be a delimiter, specified using a unicode escape, e.g., u&'\0009', which is a string consisting only of a tab character.

Specifying a <quote-string> is another way to hide an embedded delimiter. The <quote-string> should be a one or two character expression: the first is used as the <begin quote character> character; the second, if present, is used as the <end quote character> character. If only one character is supplied, it is used as both to begin and to end quoted strings. When the input includes a quoted string, that is, a string enclosed in the characters specified as <quote-string>, then that string appears in one field, even if it contains a delimiter.

Note that the <begin quote character> and <end quote character> are single characters and can be different. The <begin quote character> can be used to start and end the quoted string, or the <begin quote character> can start the quoted string and the <end quote character> used to end that quoted string.

When a list of columns <list of columns> is supplied as the second parameter <columns>, the column specifications (<column description>) for types DATE, TIME, and TIMESTAMP support a format parameter allowing the user to specify exact time component layout. The parser uses the Java class

java.lang.SimpleDateFormat to parse the strings for those types. Date and Time Patterns (p. 107) gives a full description of timestamp format strings, with examples. The following is an example of a column definition with a format string:

```
   "name" TYPE TIMESTAMP 'dd/MMM/yyyy:HH:mm:ss'
```

By default, the output columns are named COLUMN1, COLUMN2, COLUMN3, etc., each of SQL data type VARCHAR(1024).

W3C_LOG_PARSE

```
W3C_LOG_PARSE( <character-expression>, <format-string> )
<format-string> := '<predefined-format> | <custom-format>'
<predefined format> :=
        COMMON
      | COMMON WITH VHOST
      | NCSA EXTENDED
      | REFERER
      | AGENT
      | IIS
<custom-format> := [an Apache log format specifier]
```

W3C Predefined Formats

Specifying the following W3C-predefined-format names summarizes using the format specifiers indicated, as shown in the following statement:

```
select stream W3C_LOG_PARSE(message, 'COMMON') r  from w3ccommon t;
```

Format Name	W3C Name	Format Specifiers
COMMON	Common Log Format (CLF)	%h %l %u %t "%r" %>s %b
COMMON WITH VHOST	Common Log Format with Virtual Host	%v %h %l %u %t "%r" %>s %b
NCSA EXTENDED	NCSA extended/combined log format	%h %l %u %t "%r" %>s %b "%[Referer]i" "%[User-agent]i"
REFERER	Referer log format	%[Referer]i ---> %U
AGENT	Agent (Browser) log format	%[User-agent]i

W3C Format Specifiers

The format specifiers are listed below. W3C_LOG_PARSE automatically detects these specifiers and output records with one column for each specifier. The column's type is automatically chosen based on the possible outputs of the specifier. For example, %b represents the number of bytes sent in processing an HTTP request, so the column type is numeric. For %B, however, zero bytes is represented by a dash - forcing the column type to be text. Note A explains what the "..." and "<" or ">" markings shown in the specifier table mean.

The following table lists W3C format specifiers alphabetically by command.

Format Specifier	Explanation
%	The percent sign (Apache 2.0.44 and later)
%...a	Remote IP-address
%...A	Local IP-address
%...B	Size of response in bytes, excluding HTTP headers.
%...b	Size of response in bytes, excluding HTTP headers, in CLF format, which means that when no bytes are sent, uses a '-' rather than a 0.
%...[Customerdata]C	The contents of cookie Customerdata in the request sent to the server.
%...D	The time taken to serve the request, in microseconds.
%...[CUSTOMERDATA]e	The contents of the environment variable CUSTOMERDATA
%...f	Filename
%...h	Remote host
%...H	The request protocol
%...[Customerdata]i	The contents of Customerdata: header line(s) in the request sent to the server.
%...l	Remote logname (from identd, if supplied)
%...m	The request method
%...[Customerdata]n	The contents of note Customerdata from another module.
%...[Customerdata]o	The contents of Customerdata: header line(s) in the reply.
%...p	The canonical port of the server serving the request
%...P	The process ID of the child that serviced the request.
%...[format]P	The process ID or thread id of the child that serviced the request. Valid formats are pid and tid. (Apache 2.0.46 and later)
%...q	The query string (prepended with a ? if a query string exists, otherwise an empty string)
%...r	First line of request
%...s	Status. For requests that got internally redirected, this is the status of the *original* request --- %...>s for the last.

Format Specifier	Explanation
%...t	Time, in common log format time format (standard English format)
%...[format]t	The time, in the form given by format, which should be in strimmer(3) format. (potentially localized)
%...T	The time taken to serve the request, in seconds.
%...u	Remote user (from auth; may be bogus if return status (%s) is 401)
%...U	The URL path requested, not including any query string.
%...v	The canonical ServerName of the server serving the request.
%...V	The server name according to the UseCanonicalName setting.
%...X	Connection status when response is completed X = connection aborted before the response completed. + = connection may be kept alive after the response is sent. - = connection will be closed after the response is sent. (The %..X directive was %...c in late versions of Apache 1.3, but this conflicted with the historical ssl %...[var]c syntax.)
:%...I:	Bytes received, including request and headers, cannot be zero. You need to enable mod_logio to use this.
:%...O:	Bytes sent, including headers, cannot be zero. You need to enable mod_logio to use this.

Note
Some W3C format specifiers are shown as containing a "..." indication or a "<" or ">", which are optional controls on suppressing or redirecting the output of that specifier. The "..." can either be empty (as in the COMMON specification "\%h %u %r \%s %b") or it can indicate conditions for including the item. The conditions are a list of HTTP status codes, possibly preceded by "!", and if the specified condition is not met, then the column or field returned shows "-".
For example, as described in the Apache documentation, specifying "%400,501[User-agent]i" will log the User-agent only on 400 errors and 501 errors (Bad Request, Not Implemented). Similarly, "%!200,304,302[Referer]i" will log the Referer: on all requests that fail to return some sort of normal status.

The modifiers "<" and ">" can be used to choose whether the original or final (respectively) request should be consulted when a request has been internally redirected. By default, the % directives %s, %U, %T, %D, and %r look at the original request while all others look at the final request. So for example, %>s can be used to record the final status of the request and %<u can be used to record the original authenticated user on a request that is internally redirected to an unauthenticated resource.
For security reasons, starting with Apache 2.0.46, non-printable and other special characters are escaped mostly by using \xhh sequences, where hh stands for the hexadecimal representation of the raw byte. Exceptions from this rule are " and \ which are escaped by prepending a backslash, and all white space characters which are written in their C-style notation (\n, \t etc). In httpd 2.0 versions prior to 2.0.46, no escaping was performed on the strings from %...r, %...i and %...o, so great care was needed when dealing with raw log files, since clients could have inserted control characters into the log.
Also, in httpd 2.0, the B format strings represent simply the size in bytes of the HTTP response (which will differ, for instance, if the connection is aborted, or if SSL is used). For the actual number of bytes sent over the network to the client, use the %O format provided by mod_logio.

W3C Format Specifiers by Function or Category

The categories are bytes sent, connection status, content of environmental variable, filename, host, IP, notes, protocol, query string, replies, requests, and time. For the markings "..." or "<" or "<", see the previous note.

Function or Category	W3C Format Specifiers
Bytes sent, excluding HTTP headers	
with a "0" when no bytes are sent	%...B
with a "-" (CLF format) when no bytes are sent	%...b
Bytes received, including request and headers, cannot be zero Must enable mod_logio to use this.	:% ... I:
Bytes sent, including headers, cannot be zero Must enable mod_logio to use this.	:%... O:
Connection status when response is completed	
Connection aborted before the response completed	X
Connection may be kept alive after the response is sent	+
Connection will be closed after the response is sent	-
Note The %..X directive was %...c in late versions of Apache 1.3, but this conflicted with the historical ssl %...[var]c syntax.	
Environment variable CUSTOMERDATA	
contents	%...[CUSTOMERDATA]e
Filename	%...f

Function or Category	W3C Format Specifiers
Host (remote)	%...h
Protocol	%...H
IP addresses	
Remote	%...a
Local	%...A
Notes	
Contents of note Customerdata from another module	%...[Customerdata]n
Protocol (request)	%...H
Query string **Note** If query exists, prepended with a ? If not, the empty string.	%...q
Replies	
Contents of Customerdata (header lines in the reply)	%...[Customerdata]o

The W3C format specifiers for the response and time categories are listed following table.

Function or Category	W3C Format Specifiers
Requests	
Canonical port of the server serving the request	%...p
Contents of cookie Customerdata in the request sent to server	%... [Customerdata]C
Contents of BAR:header line(s)	%... [BAR]i
First line sent:	%...r
Microseconds taken to serve a request	%...D
Protocol	%...H
Process ID of the child that serviced the request	%...P
Process ID or thread id of the child that serviced the request. Valid formats are pid and tid. (Apache 2.0.46 and later)	%...[format]P
Remote logname (from identd, if supplied)	%...l
Remote user: (from auth; may be bogus if return status (%s) is 401)	%...u

Function or Category	W3C Format Specifiers
Server (canonical ServerName) serving the request	%...v
Server name by the UseCanonicalName setting	%...V
Request method	%...m
Return status	%s
Seconds taken to serve the request	%...T
Status of the *original* request that was internally redirected	%...s
Status of the last request	%...>s
URL path requested, not including any query string	%...U
Time	
Common log format time format (standard English format)	%...t
Time in strftime(3) format, potentially localized	%...[format]t
Seconds taken to serve the request	%...T

W3C Examples

W3C_LOG_PARSE supports access to logs generated by W3C-compliant applications like the Apache web server, producing output rows with one column for each specifier. The data types are derived from the log entry description specifiers listed in the Apache mod_log_config specification.

Example 1

The input in this example is taken from an Apache log file and is representative of the COMMON log format.

Input

```
(192.168.254.30 - John [24/May/2004:22:01:02 -0700]
                    "GET /icons/apache_pb.gif HTTP/1.1" 304 0),
(192.168.254.30 - Jane [24/May/2004:22:01:02 -0700]
                    "GET /icons/small/dir.gif HTTP/1.1" 304 0);
```

DDL

```
CREATE OR REPLACE PUMP weblog AS
        SELECT STREAM
            l.r.COLUMN1,
            l.r.COLUMN2,
            l.r.COLUMN3,
            l.r.COLUMN4,
            l.r.COLUMN5,
            l.r.COLUMN6,
```

```
        l.r.COLUMN7
    FROM (SELECT STREAM W3C_LOG_PARSE(message, 'COMMON')
          FROM "weblog_read) AS l(r);
```

Output

```
 192.168.254.30 -  John  [24/May/2004:22:01:02 -0700] GET /icons/
apache_pb.gif HTTP/1.1  304  0
 192.168.254.30 -  Jane  [24/May/2004:22:01:02 -0700] GET /icons/small/
dir.gif HTTP/1.1  304  0
```

The specification of COMMON in the FROM clause means the Common Log Format (CLF), which uses the specifiers %h %l %u %t "%r" %>s %b.

The W3C-predefined formats shows the COMMON and other predefined specifier sets.

The specification of COMMON in the FROM clause means the Common Log Format (CLF), which uses the specifiers %h %l %u %t "%r" %>s %b.

The table below, Specifiers used by the Common Log Format, describes the specifiers used by COMMON in the FROM clause.

Specifiers Used by the Common Log Format

Output Column	Format Specifier	Returns
COLUMN1	%h	The IP address of the remote host
COLUMN2	%l	The remote logname
COLUMN3	%u	The remote user
COLUMN4	%t	The time
COLUMN5	"%r"	The first line of the request
COLUMN6	%>s	The status: For internally redirected requests, the status of the *original* request --- %...>s for the last.
COLUMN7	%b	The number of bytes sent, excluding HTTP headers

Example 2

The DDL in this example shows how to rename output columns and filter out unneeded columns.

DDL

```
   CREATE OR REPLACE VIEW "Schema1".weblogreduced AS
```

```
        SELECT STREAM CAST(s.COLUMN3 AS VARCHAR(5)) AS LOG_USER,
        CAST(s.COLUMN1 AS VARCHAR(15)) AS ADDRESS,
        CAST(s.COLUMN4 AS VARCHAR(30)) as TIME_DATES
        FROM "Schema1".weblog s;
```

Output

```
    +----------+-----------------+--------------------------------+
    | LOG_USER |     ADDRESS     |           TIME_DATES           |
    |          |                 |                                |
    +----------+-----------------+--------------------------------+
    | Jane     | 192.168.254.30  | [24/May/2004:22:01:02 -0700]   |
    |          |                 |                                |
    | John     | 192.168.254.30  | [24/May/2004:22:01:02 -0700]   |
    +----------+-----------------+--------------------------------+
```

W3C Customized Formats

The same results would be created by naming the specifiers directly rather than using the "COMMON" name, as shown following:

```
    CREATE OR REPLACE FOREIGN STREAM schema1.weblog
        SERVER logfile_server
        OPTIONS (LOG_PATH '/path/to/logfile',
                 ENCODING 'UTF-8',
                 SLEEP_INTERVAL '10000',
                 MAX_UNCHANGED_STATS '10',
                 PARSER 'W3C',
                 PARSER_FORMAT '%h %l %u %t \"%r\" %>s %b');
    or
     CREATE FOREIGN STREAM "Schema1".weblog_read
     SERVER "logfile_server"
     OPTIONS (log_path '/path/to/logfile',
     encoding 'UTF-8',
     sleep_interval '10000',
     max_unchanged_stats '10');
     CREATE OR REPLACE VIEW "Schema1".weblog AS
        SELECT STREAM
            l.r.COLUMN1,
            l.r.COLUMN2,
            l.r.COLUMN3,
            l.r.COLUMN4,
            l.r.COLUMN5,
            l.r.COLUMN6
        FROM (SELECT STREAM W3C_LOG_PARSE(message, '%h %l %u %t \"%r\" %>s
%b')
                FROM "Schema1".weblog_read) AS l(r);
```

Note
If you change %t to [%t], the date column contains the following:

```
        24/May/2004:22:01:02 -0700
```

(instead of `[24/May/2004:22:01:02 -0700]`)

Aggregate Functions

Instead of returning a result calculated from a single row, an aggregate function returns a result calculated from aggregated data contained in a finite set of rows, or from information about a finite set of rows. An aggregate function may appear in any of the following:

- <selection list> portion of a SELECT clause (p. 137)
- ORDER BY clause (p. 158)
- HAVING clause (p. 147)

An aggregate function is different from Analytic Functions (p. 66), which are always evaluated relative to a window that must be specified, and so they can't appear in a HAVING clause. Other differences are described in the table later in this topic.

Aggregate functions operate slightly differently in aggregate queries on tables than when you use them in aggregate queries on streams, as follows. If an aggregate query on tables contains a GROUP BY clause, the aggregate function returns one result per group in the set of input rows. Lacking an explicit GROUP BY clause is equivalent to GROUP BY (), and returns only one result for the entire set of input rows.

On streams, an aggregate query must contain an explicit GROUP BY clause on a monotonic expression based on rowtime. Without one, the sole group is the whole stream, which never ends, preventing any result from being reported. Adding a GROUP BY clause based on a monotonic expression breaks the stream into finite sets of rows, contiguous in time, and each such set can then be aggregated and reported.

Whenever a row arrives that changes the value of the monotonic grouping expression, a new group is started and the previous group is considered complete. Then, the Amazon Kinesis Analytics application outputs the value of the aggregate functions. Note that the GROUP BY clause may also include other non-monotonic expressions, in which case more than one result per set of rows may be produced.

Performing an aggregate query on streams is often referred to as streaming aggregation, as distinct from the windowed aggregation discussed in Analytic Functions (p. 66) and Windowed Aggregation on Streams (p. 72). For more information about stream-to-stream joins, see JOIN clause (p. 142).

If an input row contains a `null` in a column used as an input to a data analysis function, the data analysis function ignores the row (except for COUNT).

Differences Between Aggregate and Analytic Functions

Function Type	Outputs	Rows or Windows Used	Notes
Aggregate Functions	One output row per group of input rows.	All output columns are calculated over the same window or same group of rows.	COUNT DISTINCT is not allowed in streaming aggregation. Statements of the following type are not allowed: SELECT COUNT(DISTINCT x) ... FROM ... GROUP BY ...
Analytic Functions (p. 66)	One output row for each input row.	Each output column may be calculated	COUNT DISTINCT can't be used as Analytic

Function Type	Outputs	Rows or Windows Used	Notes
		using a different window or partition.	Functions (p. 66) or in windowed aggregation.

Streaming Aggregation and Rowtime Bounds

Normally, an aggregate query generates a result when a row arrives that changes the value of the monotonic expression in the GROUP BY. For example, if the query is grouped by FLOOR(rowtime TO MINUTE), and the rowtime of the current row is 9:59.30, then a new row with a rowtime of 10:00.00 will trigger the result.

Alternately, a rowtime bound can be used to advance the monotonic expression and enable the query to return a result. For example, if the query is grouped by FLOOR(rowtime TO MINUTE), and the rowtime of the current row is 9:59.30, then an incoming rowtime bound of 10:00.00 will also trigger the result.

Aggregate Function List

Amazon Kinesis Analytics supports the following aggregate functions:

- AVG (p. 59)
- COUNT (p. 60)
- MAX (p. 62)
- MIN (p. 62)
- SUM (p. 63)
- STDDEV_POP (p. 64)
- STDDEV_SAMP (p. 64)
- VAR_POP (p. 64)
- VAR_SAMP (p. 65)

For more information, see Statistical Variance and Deviation Functions (p. 63).

The following SQL uses the AVG aggregate function as part of a query to find the average age of all employees:

```
SELECT
    AVG(AGE) AS AVERAGE_AGE
FROM SALES.EMPS;
```

Result:

AVERAGE_AGE
38

To find the average age of employees in each department, we can add an explicit GROUP BY clause to the query:

```
SELECT
```

```
    DEPTNO,
    AVG(AGE) AS AVERAGE_AGE
FROM SALES.EMPS
GROUP BY DEPTNO;
```

Returns:

DEPTNO	AVERAGE_AGE
10	30
20	25
30	40
40	57

Examples of Aggregate Queries on Streams (Streaming Aggregation)

For this example, assume that the data in the following table is flowing through the stream called WEATHERSTREAM.

ROWTIME	CITY	TEMP
2018-11-01 01:00:00.0	Denver	29
2018-11-01 01:00:00.0	Anchorage	2
2018-11-01 06:00:00.0	Miami	65
2018-11-01 07:00:00.0	Denver	32
2018-11-01 09:00:00.0	Anchorage	9
2018-11-01 13:00:00.0	Denver	50
2018-11-01 17:00:00.0	Anchorage	10
2018-11-01 18:00:00.0	Miami	71
2018-11-01 19:00:00.0	Denver	43
2018-11-02 01:00:00.0	Anchorage	4
2018-11-02 01:00:00.0	Denver	39
2018-11-02 07:00:00.0	Denver	46
2018-11-02 09:00:00.0	Anchorage	3
2018-11-02 13:00:00.0	Denver	56
2018-11-02 17:00:00.0	Anchorage	2
2018-11-02 19:00:00.0	Denver	50
2018-11-03 01:00:00.0	Denver	36
2018-11-03 01:00:00.0	Anchorage	1

If you want to find the minimum and maximum temperature recorded anywhere each day (globally regardless of city), the minimum and maximum temperature can be calculated using the aggregate functions MIN and MAX respectively. To indicate that we want this information on a per-day basis (and to provide a monotonic expression as the argument of the GROUP BY clause), we use the FLOOR function to round each row's rowtime down to the nearest day:

```
SELECT STREAM
    FLOOR(WEATHERSTREAM.ROWTIME to DAY) AS FLOOR_DAY,
    MIN(TEMP) AS MIN_TEMP,
    MAX(TEMP) AS MAX_TEMP
FROM WEATHERSTREAM
GROUP BY FLOOR(WEATHERSTREAM.ROWTIME TO DAY);
```

The result of the aggregate query is shown in the following table.

FLOOR_DAY	MIN_TEMP	MAX_TEMP
2018-11-01 00:00:00.0	2	71
2018-11-02 00:00:00.0	2	56

There is no row for 2018-11-03, even though the example data does include temperature measurements on that day. This is because the rows for 2018-11-03 cannot be aggregated until all rows for that day are known to have arrived, and that will only happen when either a row with a rowtime of 2018-11-04 00:00:00.0 (or later) or a rowtime bound of 2018-11-04 00:00:00.0 (or later) arrives. If and when either did arrive, the next result would be as described in the following table.

FLOOR_DAY	MIN_TEMP	MAX_TEMP
2018-11-03 00:00:00.0	1	36

Let's say that instead of finding the global minimum and maximum temperatures each day, we want to find the minimum, maximum, and average temperature for each city each day. To do this, we use the SUM and COUNT aggregate functions to compute the average, and add CITY to the GROUP BY clause, as shown following:

```
SELECT STREAM
    FLOOR(WEATHERSTREAM.ROWTIME TO DAY) AS FLOOR_DAY,
    CITY,
    MIN(TEMP) AS MIN_TEMP,
    MAX(TEMP) AS MAX_TEMP,
    SUM(TEMP)/COUNT(TEMP) AS AVG_TEMP
FROM WEATHERSTREAM
GROUP BY FLOOR(WEATHERSTREAM.ROWTIME TO DAY), CITY;
```

The result of the aggregate query is shown in the following table.

FLOOR_DAY	CITY	MIN_TEMP	MAX_TEMP	AVG_TEMP
2018-11-01 00:00:00.0	Anchorage	2	10	7
2018-11-01 00:00:00.0	Denver	29	50	38

FLOOR_DAY	CITY	MIN_TEMP	MAX_TEMP	AVG_TEMP
2018-11-01 00:00:00.0	Miami	65	71	68
2018-11-02 00:00:00.0	Anchorage	2	4	3
2018-11-02 00:00:00.0	Denver	39	56	47

In this case, the arrival of rows for a new day's temperature measurements triggers the aggregation of the previous day's data, grouped by CITY, which then results in one row being produced per city included in the day's measurements.

Here again, a rowtime bound 2018-11-04 00:00:00.0 could be used to prompt a result for 2018-11-03 prior to any actual measurements for 2018-11-04 coming in is shown in the following table.

FLOOR_DAY	CITY	MIN_TEMP	MAX_TEMP	AVG_TEMP
2018-11-03 00:00:00.0	Anchorage	1	1	1
2018-11-03 00:00:00.0	Denver	36	36	36

AVG

```
AVG ( [DISTINCT  ALL] <number-expression> ) [ OVER <window-specification> ]
```

AVG returns the average of all the value expressions evaluated for each row in the aggregation. When used without the OVER clause, AVG is considered an Aggregate Functions (p. 55). When used with the OVER clause, it is an Analytic Functions (p. 66). (For exponential averaging, see EXP_AVG (p. 41).)

If DISTINCT is specified, only rows that match the <value expression> and have unique values qualify. If ALL is specified, all rows qualify. If neither DISTINCT nor ALL is specified, the behavior defaults to ALL.

When used as an Analytic Functions (p. 66), AVG will return null if the window being evaluated contains no rows, or if all rows contain null values. This will also be the result in the case of a PARTITION BY for which the partition within the window matching the input row contains no rows or all rows are null.

Otherwise AVG ignores null values. AVG of 1, 2, 3 is 2. AVG of 1,null, 2, null, 3, null is also 2 - the null values aren't counted as part of the total or in the count of rows. So AVG(x) is the same as SUM(x) / COUNT(x).

Example

This example shows the difference between AVG(ALL pct_free), which is calculated as (71 * 10 + 1 * 0)/72 = 9.86, and AVG(DISTINCT pct_free), which is calculated as (10 + 0)/2 = 5.

```
 Select Stream pct_free, count(*) from test1 group by pct_free;
+-----------+---------+
```

```
 PCT_FREE   EXPR$1
+-----------+---------+
 10.0       71
 0.0        1
+-----------+---------+
2 rows selected (0.672 seconds)
 Select Stream avg(all pct_free) as avg_all from test1;
+-------------------+
      AVG_ALL
+-------------------+
 9.86111111111111
+-------------------+
1 row selected (0.438 seconds)
 Select Stream avg(distinct pct_free) as avg_distinct from test1;
+---------------+
 AVG_DISTINCT
+---------------+
 5.0
+---------------+
1 row selected (0.516 seconds)
```

Limitations

AVG is only supported on numeric types.

> **Note**
> Amazon Kinesis Analytics does not support AVG applied to interval types. This is a departure from the SQL:2008 standard.

COUNT

First form is as follows:

```
COUNT( [DISTINCT | ALL] <value-expression> ) [ OVER <window-specification> ]
```

Regarding DISTINCT, see Limitations, later in this topic.

Second form is as follows:

```
COUNT ( * ) [OVER <window-specification> ]
```

The COUNT function returns the number of qualifying rows in the aggregation. When used without the OVER clause, COUNT is considered an Aggregate Functions (p. 55), as shown following:

```
COUNT( <value-expression> )
COUNT( ALL <value-expression> )
COUNT( DISTINCT <value-expression> )
```

When used with the OVER clause, it is an Analytic Functions (p. 66), as shown following:

```
COUNT( <value-expression> ) OVER ( <window-specification> )
COUNT( ALL <value-expression> ) OVER ( <window-specification> )
```

In all the above forms of the COUNT function, only rows where the <value_expression> is not NULL are counted. If ALL is specified, all such rows are counted. Since ALL is assumed by default, the first two above are equivalent.

If DISTINCT is specified, only distinct values of <value expression> are counted.

When used as an Analytic Functions (p. 66), COUNT will return zero if the window being evaluated contains no rows. In the case of a PARTITION BY, COUNT will return zero if the partition within the window matching the input row contains no rows.

In the second form, COUNT(*), all rows qualify.

Limitations

Amazon Kinesis Analytics does not support the FILTER clause of the COUNT function, nor the use of COUNT DISTINCT in either Aggregate Functions (p. 55), Analytic Functions (p. 66), or as an analytic function. These are departures from the SQL:2008 standard.

COUNT_DISTINCT_ITEMS_TUMBLING Function

Returns a count of the number of distinct items in the specified in-application stream column over a tumbling window. The resulting count is approximate, the function uses the HyperLogLog algorithm. For more information, see HyperLogLog on the Wikipedia website. Note that when there are less than or equal to 10,000 items in the window, the function returns an exact count.

Note
Getting an exact count of the number of distinct items can be inefficient and costly. Therefore, this function approximates the count. For example, if there are 100,000 distinct items, the algorithm may return 99,700. If cost and efficiency is not a consideration, you can write your own SELECT statement to get the exact count. For example:

```
CREATE OR REPLACE STREAM output_stream (ticker_symbol VARCHAR(4),
 unique_count BIGINT);

CREATE OR REPLACE PUMP stream_pump AS
INSERT INTO output_stream
SELECT STREAM ticker_symbol, COUNT(distinct_stream.price) AS
 unique_count
FROM (
    SELECT STREAM DISTINCT rowtime as window_time, ticker_symbol,
                  sector, CHANGE, price,
                  FLOOR((source_sql_stream_001.rowtime - TIMESTAMP
 '1970-01-01 00:00:00') SECOND / 5 TO SECOND)
    FROM source_sql_stream_001) as distinct_stream
GROUP BY ticker_symbol,
         FLOOR((distinct_stream.window_time - TIMESTAMP '1970-01-01
 00:00:00') SECOND / 5 TO SECOND);
```

This SELECT statement returns an exact count of the number of distinct rows for each ticker symbol in a five second tumbling window. The SELECT statement uses all of the columns (except ROWTIME) in determining the uniqueness.

The function operates on a tumbling window. You specify the size of the tumbling window as a parameter.

For an example, see Count Distinct Values.

Syntax

```
COUNT_DISTINCT_ITEMS_TUMBLING (
```

```
    in-application-stream Pointer,
    'columnName',
    windowSize
)
```

Parameters

The following sections describe the parameters.

in-application-streamPointer

Using this parameter, you provide a pointer to an in-application stream. You can set a pointer using the `CURSOR` function. For example, the following statement sets a pointer to `InputStream`.

```
CURSOR(SELECT STREAM * FROM InputStream)
```

columnName

Column name in your in-application stream that you want the function to use to count distinct values. Note the following about the column name:

- Must appear in single quotation marks ('). For example, `'column1'`.

windowSize

Size of the tumbling window in seconds. The size should be at least 1 second and should not exceed 1 hour = 3600 seconds.

MAX

```
MAX ( [DISTINCT | ALL] <value-expression> ) [ OVER <window-specification> ]
```

MAX returns the maximum value of all the value expressions evaluated for each row in the aggregation. When used without the OVER clause, MAX is considered an Aggregate Functions (p. 55). When used with the OVER clause, it is an Analytic Functions (p. 66).

For string values, MAX is determined by which string is last in the collating sequence.

When used as an Analytic Functions (p. 66), MAX will return null if the window being evaluated (or in the case of a PARTITION BY, the partition within the window matching the input row) contains no rows.

MIN

```
MIN ( [DISTINCT | ALL] <value-expression> ) [ OVER <window-specification> ]
```

MIN returns the minimum value of all the value expressions evaluated for each row in the aggregation. When used without the OVER clause, MIN is considered an Aggregate Functions (p. 55). When used with the OVER clause, it is an Analytic Functions (p. 66).

For string values, MIN is determined by which string is first in the collating sequence.

When used as an Analytic Functions (p. 66), MIN will return null if the window being evaluated (or in the case of a PARTITION BY, the partition within the window matching the input row) contains no rows.

SUM

```
SUM ( [DISTINCT | ALL] <number-expression> ) [ OVER <window-specification> ]
```

SUM returns the sum of all the value expressions evaluated for each row in the aggregation. When used without the OVER clause, SUM is considered an Aggregate Functions (p. 55). When used with the OVER clause, it is an Analytic Functions (p. 66).

If DISTINCT is specified, only distinct row values qualify. If ALL is specified, all rows qualify. If neither DISTINCT nor ALL is specified, the behavior defaults to ALL.

When used as an Analytic Functions (p. 66), SUM will return null if the window being evaluated (or in the case of a PARTITION BY, the partition within the window matching the input row) contains no rows.

Examples

The following example shows the difference between SUM(ALL value) and SUM(DISTINCT value); note how SUM(ALL value) returns 710 (71 rows with a value of 10 plus 1 row with a value of 0) whereas SUM(DISTINCT value) returns just 10 (10 + 0) as there are only two distinct values for PCT_FREE.

```
 Select Stream pct_free, count(*) from test1 group by pct_free;
+-----------+---------+
| PCT_FREE  | EXPR$1  |
+-----------+---------+
| 10.0      | 71      |
| 0.0       | 1       |
+-----------+---------+
2 rows selected (0.453 seconds)
 Select Stream sum(all pct_free) as sum_all from test1;
+----------+
| SUM_ALL  |
+----------+
| 710.0    |
+----------+
1 row selected (0.391 seconds)
Select Stream sum(distinct pct_free) as sum_distinct from test1;
+---------------+
| SUM_DISTINCT  |
+---------------+
| 10.0          |
+---------------+
1 row selected (0.422 seconds)
```

SUM is only supported on numeric types.

Limitation

Amazon Kinesis Analytics streaming SQL does not support SUM applied to interval types. This is a departure from the SQL:2008 standard.

Statistical Variance and Deviation Functions

Each of these functions takes a set of numbers, ignores nulls, and can be used as either an Aggregate Functions (p. 55) or an Analytic Functions (p. 66).

The relationships among these functions are described in the following table.

Function purpose	Function name	Formula	Comments
Population variance	VAR_POP (p. 64)(expr)	(SUM(expr*expr) - SUM(expr)*SUM(expr) / COUNT(expr)) / COUNT(expr)	Applied to an empty set, it returns null.
Population standard deviation	STDDEV_POP (p. 64)	Square root of the population variance (VAR_POP).	When VAR_POP returns null, STDDEV_POP returns null.
Sample variance	VAR_SAMP (p. 65)	(SUM(expr*expr) - SUM(expr)*SUM(expr) / COUNT(expr)) / (COUNT(expr)#1)	Applied to an empty set, it returns null. Applied to an input set of one element, VAR_SAMP returns null.
Sample standard deviation	STDDEV_SAMP (p. 64) (expr)	Square root of the sample variance (VAR_SAMP).	Applied to only 1 row of input data, STDDEV_SAMP returns null.

STDDEV_POP

Returns the square root of the VAR_POP (p. 64) population variance for <number expression>, evaluated for each row remaining in the group.

```
STDDEV_POP ( [DISTINCT | ALL] number-expression )
```

Where ALL includes (and DISTINCT excludes) duplicate values in the input set. ALL is the default. When the input set has no non-null data, STDDEV_POP returns NULL.

See also STDDEV_SAMP (p. 64) sample standard deviation, VAR_SAMP (p. 65), and VAR_POP (p. 64).

STDDEV_SAMP

Returns the statistical standard deviation of all values in number-expression, evaluated for each row remaining in the group and defined as the square root of the VAR_SAMP (p. 65).

```
STDDEV_SAMP ( [DISTINCT | ALL] number-expression )
```

Where ALL includes (and DISTINCT excludes) duplicate values in the input set. ALL is the default. When the input set has no non-null data, STDDEV_SAMP returns NULL.

STD_DEV is an alias of STDDEV_SAMP.

See also STDDEV_POP (p. 64)population standard deviation, VAR_SAMP (p. 65), and VAR_POP (p. 64).

VAR_POP

Returns the population variance of a non-null set of numbers (nulls being ignored)

```
VAR_POP ( [DISTINCT | ALL] number-expression )
```

Where ALL includes (and DISTINCT excludes) duplicate values in the input set. ALL is the default. When the input set has no non-null data, VAR_POP returns NULL.

VAR_POP uses the following calculation:

- (SUM(expr*expr) - SUM(expr)*SUM(expr) / COUNT(expr)) / COUNT(expr)

In other words, for a given set of non-null values, using S1 as the sum of the values and S2 as the sum of the squares of the values, VAR_POP returns the result (S2-S1*S1/N)/N.

You can use VAR_POP as either an aggregate and analytic function. Applied to an empty set, it returns null.

See also VAR_SAMP (p. 65) sample variance, STDDEV_POP (p. 64), and STDDEV_SAMP (p. 64).

VAR_SAMP

Returns the sample variance of a non-null set of numbers (nulls being ignored),

```
VAR_SAMP ( [DISTINCT | ALL] number-expression )
```

Where ALL includes (and DISTINCT excludes) duplicate values in the input set. ALL is the default. When the input set has no non-null data, VAR_SAMP returns NULL.

VAR_SAMP uses the following calculation:

- (SUM(expr*expr) - SUM(expr)*SUM(expr) / COUNT(expr)) / (COUNT(expr)-1)

In other words, for a given set of non-null values, using S1 as the sum of the values and S2 as the sum of the squares of the values, VAR_POP returns the result (S2-S1*S1/N)/(N-1).

You can use VAR_SAMP as either an aggregate and analytic function. Applied to an empty set, it returns null. Given an input set of one element, VAR_SAMP returns null.

See also VAR_POP (p. 64) population variance, STDDEV_POP (p. 64), and STDDEV_SAMP (p. 64).

TOP_K_ITEMS_TUMBLING Function

Returns the most frequently occurring values in the specified in-application stream column over a tumbling window. This can be used to find trending (most popular) values in a specified column.

For example, the Getting Started exercise uses a demo stream that provides continuous stock price updates (ticker_symbol, price, change, and other columns). Suppose you want to find the three most frequently traded stocks in each 1-minute tumbling window. You can use this function to find those ticker symbols.

> **Note**
> Counting each incoming record on your streaming source is not efficient, therefore the function approximates the most frequently occurring values. For example, when seeking the three most traded stocks, the function may return three of the five most traded stocks.

The function operates on a tumbling window. You specify the window size as a parameter.

For a sample application with step-by-step instructions, see Most Frequently Occurring Values.

Syntax

```
TOP_K_ITEMS_TUMBLING (
      in-application-streamPointer,
      'columnName',
      K,
      windowSize,
   )
```

Parameters

The following sections describe the parameters.

in-application-streamPointer

Pointer to an in-application stream. You can set a pointer using the CURSOR function. For example, the following statement sets a pointer to InputStream.

```
CURSOR(SELECT STREAM * FROM InputStream)
```

columnName

Column name in your in-application stream that you want to use to compute the topK values. Note the following about the column name:

- Must appear in single quotation marks ('). For example, `'column1'`.

K

Using this parameter, you specify how many of the most frequently occurring values from a specific column you want returned. The value K should be at least 1 and cannot exceed 100,000.

windowSize

Size of the tumbling window in seconds. The size should be at least 1 second and should not exceed 1 hour = 3600 seconds.

Analytic Functions

An analytic function is one that returns a result calculated from data in (or about) a finite set of rows identified by a SELECT clause (p. 137) or in the ORDER BY clause (p. 158).

The list of analytic functions that Amazon Kinesis Analytics supports appears below. The SELECT topic explains the order-by clause, showing the order-by chart, as well as the windowing clause (and window-specification chart). To see where an order-by clause is used in Select statements, see the Select chart in the SELECT topic of this guide.

1. Analytic functions must specify a window. Since there are a few restrictions on window specifications, and a few differences between specifying windows for windowed aggregation and windowed join, please see Allowed and Disallowed Window Specifications (p. 155) for explanations.
2. Analytic functions may only appear in the <selection list> portion of a SELECT clause or in the ORDER BY clause.
3. The list of analytic functions that Amazon Kinesis Analytics supports appears below. The SELECT topic explains the order-by clause, showing the order-by chart, as well as the windowing clause (and

window-specification chart). To see where an order-by clause is used in Select statements, see the Select chart in the SELECT topic of this guide.

Other differences are described in the table later in this topic.

Performing queries using analytic functions is commonly referred to as windowed aggregation (discussed below), as distinct from Aggregate Functions (p. 55).

Because of the presence of the window specification, queries that use analytic functions produce results in a different manner than do aggregate queries. For each row in the input set, the window specification identifies a different set of rows on which the analytic function operates. If the window specification also includes a PARTITION BY clause, then the only rows in the window that will be considered in producing a result will be those that share the same partition as the input row.

If an input row contains a null in a column used as an input to an analytic function, the analytic function ignores the row, except for COUNT, which does count rows with null values. In cases where the window (or in the case of a PARTITION BY, a partition within the window) contains no rows, an analytic function will return null. The exception to this is COUNT, which returns zero.

Differences Between Aggregate and Analytic Functions

Function Type	Outputs	Rows or Windows Used	Notes
Aggregate Functions (p. 55)	One output row per group of input rows.	All output columns are calculated over the same window or same group of rows.	COUNT DISTINCT is not allowed in Aggregate Functions (p. 55). Statements of the following type are not allowed: SELECT COUNT(DISTINCT x) ... FROM ... GROUP BY ...
Analytic Functions	One output row for each input row.	Each output column may be calculated using a different window or partition.	COUNT DISTINCT can't be used as analytic functions or in windowed aggregation.

Related Topics

- Windowed Aggregation on Streams (p. 72)
- SELECT statement (p. 134)
- SELECT clause (p. 137)

Functions that Implement Unsupervised Machine Learning Algorithms

Amazon Kinesis Analytics provides the following unsupervised functions that you might find useful in streaming analytics:

Topics

RANDOM_CUT_FOREST Function

Detects anomalies in your data stream. A record is an anomaly if it is distant from other records.

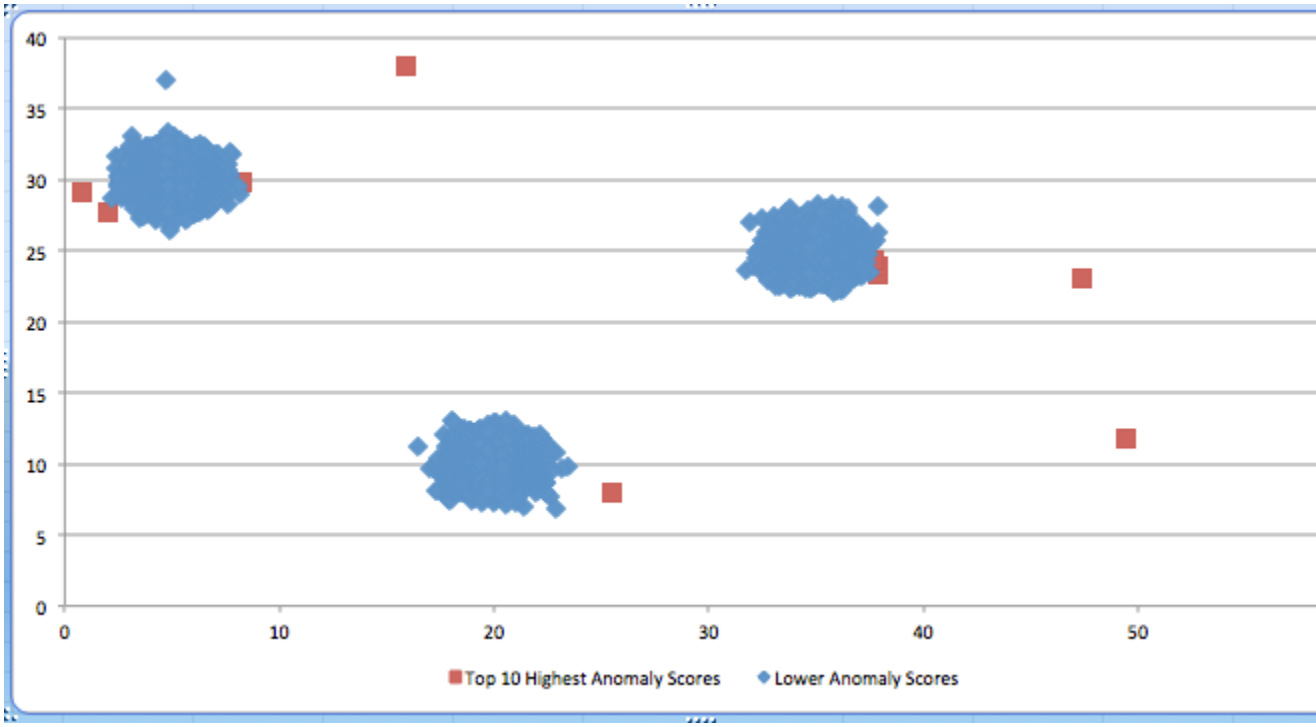
> **Note**
> The `RANDOM_CUT_FOREST` function's ability to detect anomalies is application-dependent. To cast your business problem so that it can be solved with this function requires domain expertise. For example, determining which combination of columns in your input stream to pass to the function and potentially normalize the data. For more information, see inputStream (p. 69).

A stream record can have non-numeric columns, but the function uses only numeric columns to assign an anomaly score. A record can have one or more numeric columns. The algorithm uses all of the numeric data in computing an anomaly score. If a record has ***n*** numeric columns, the underlying algorithm assumes each record is a point in ***n***-dimensional space. A point in ***n***-dimensional space that is distant from other points receives a higher anomaly score.

The algorithm accepts the `DOUBLE`, `INTEGER`, `FLOAT`, `TINYINT`, `SMALLINT`, `REAL`, and `BIGINT` data types.

> **Note**
> DECIMAL is not a supported type. Use DOUBLE instead.

The following is an example of anomaly detection. The diagram shows three clusters and a few anomalies randomly interjected. The red squares show the records that received the highest anomaly score according to the `RANDOM_CUT_FOREST` function. The blue diamonds represent the remaining records. Note how the highest scoring records tend to be outside the clusters.



For a sample application with step-by-step instructions, see Detect Anomalies.

Syntax

```
RANDOM_CUT_FOREST (inputStream,
                   numberOfTrees,
                   subSampleSize,
                   timeDecay,
                   shingleSize)
```

Parameters

The following sections describe the parameters.

inputStream

Pointer to your input stream. You set a pointer using the `CURSOR` function. For example, the following statements sets a pointer to `InputStream`.

```
CURSOR(SELECT STREAM * FROM InputStream)
CURSOR(SELECT STREAM IntegerColumnX, IntegerColumnY FROM InputStream)
-- Perhaps normalize the column X value.
CURSOR(SELECT STREAM IntegerColumnX / 100, IntegerColumnY FROM InputStream)
-- Combine columns before passing to the function.
CURSOR(SELECT STREAM IntegerColumnX - IntegerColumnY FROM InputStream)
```

The `CURSOR` function is the only required parameter for the `RANDOM_CUT_FOREST` function. The function assumes the following default values for the other parameters:

numberOfTrees = 100

subSampleSize = 256

timeDecay = 100,000

shingleSize = 1

When using this function, your input stream can have up to 30 numeric columns.

numberOfTrees

Using this parameter, you specify the number of random cut trees in the forest.

> **Note**
> By default, the algorithm constructs a number of trees, each constructed using a given number of sample records (see `subSampleSize` later in this list) from the input stream. The algorithm uses each tree to assign an anomaly score. The average of all these scores is the final anomaly score.

The default value for `numberOfTrees` is 100. You can set this value between 1 and 1,000 (inclusive). By increasing the number of trees in the forest, you can get a better estimate of the anomaly score, but this also increases the running time.

subSampleSize

Using this parameter, you can specify the size of the random sample that you want the algorithm to use when constructing each tree. Each tree in the forest is constructed with a (different) random sample of records. The algorithm uses each tree to assign an anomaly score.

The default value for `subSampleSize` is 256. You can set this value between 10 and 1,000 (inclusive).

Note that the `subSampleSize` must be less than the `timeDecay` parameter (which is set to 100,000 by default). Increasing the sample size provides each tree a larger view of the data, but also increases the running time.

timeDecay

The `timeDecay` parameter allows you to specify how much of the recent past to consider when computing an anomaly score. This is because data streams naturally evolve over time. For example, an eCommerce website's revenue may continuously increase, or global temperatures may rise over time. In such situations, we want an anomaly to be flagged relative to recent data, as opposed to data from the distant past.

The default value is 100,000 records (or 100,000 shingles if shingling is used, as described in the following section). You can set this value between 1 and the maximum integer (that is, 2147483647). The algorithm exponentially decays the importance of older data.

If you choose the `timeDecay` default of 100,000, the anomaly detection algorithm does the following:

- Uses only the most recent 100,000 records in the calculations (and ignores older records).
- Within the most recent 100,000 records, assigns exponentially more weight to recent records and less to older records in anomaly detection calculations.

The `timeDecay` parameter determines the maximum quantity of recent records kept in the working set of the anomaly detection algorithm. Smaller `timeDecay` values are desirable if the data is changing rapidly. The best `timeDecay` value is application-dependent.

shingleSize

The explanation given here is for a one-dimensional stream (that is, a stream with one numeric column), but it can also be used for multi-dimensional streams.

A shingle is a consecutive sequence of the most recent records. For example, a `shingleSize` of 10 at time *t* corresponds to a vector of the last 10 records received up to and including time *t*. The algorithm treats this sequence as a vector over the last `shingleSize` number of records.

If data is arriving uniformly in time, a shingle of size 10 at time *t* corresponds to the data received at time *t-9, t-8,…,t*. At time *t+1*, the shingle slides over one unit and consists of data from time *t-8,t-7, …, t, t+1*. These shingled records gathered over time correspond to a collection of 10-dimensional vectors over which the anomaly detection algorithm runs.

The intuition is that a shingle captures the shape of the recent past. Your data may have a typical shape. For example, if your data is collected hourly, a shingle of size 24 may capture the daily rhythm of your data.

The default `shingleSize` is one record (because shingle size is data dependent). You can set this value between 1 and 30 (inclusive).

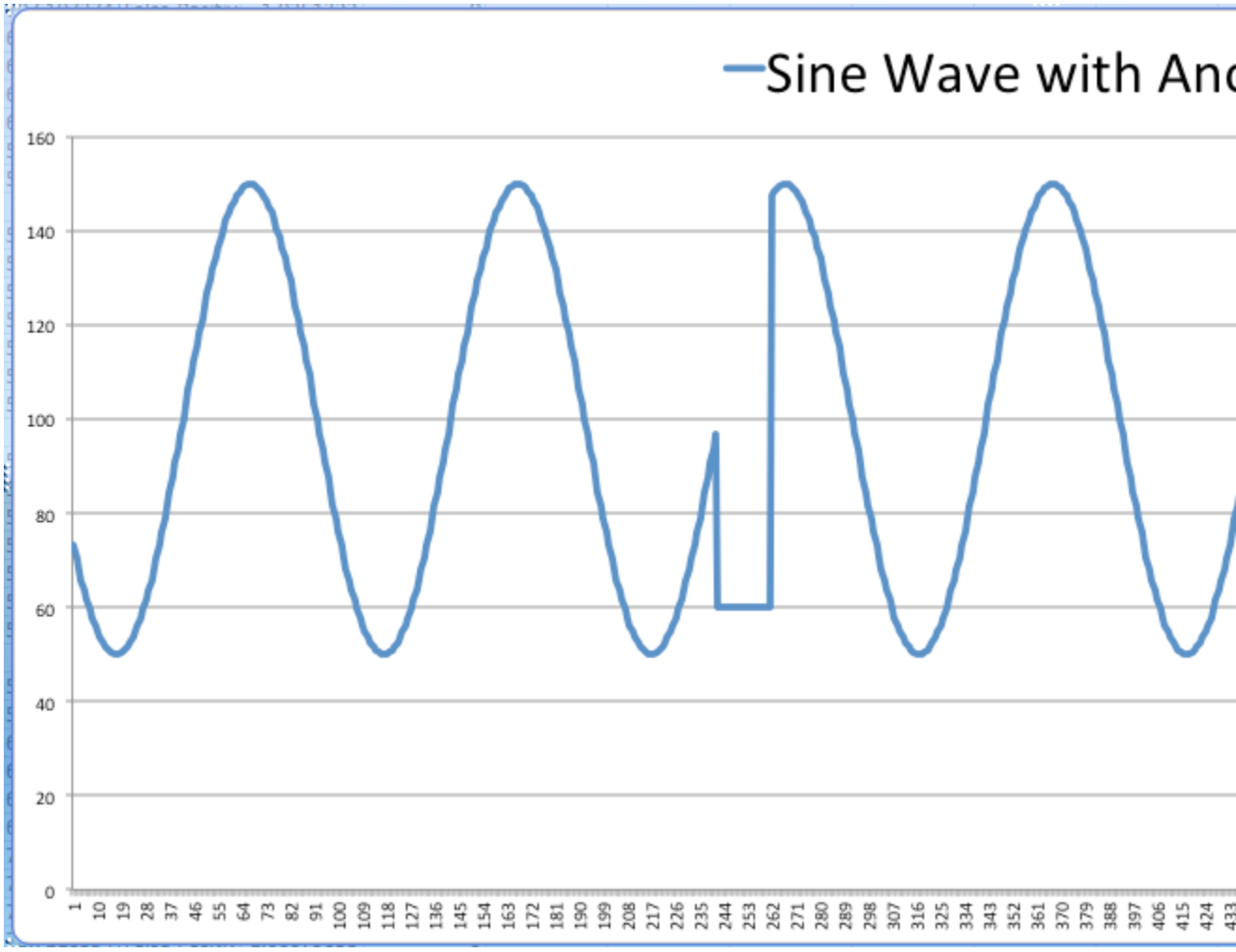
Note the following about setting the `shingleSize`:

- If you set the `shingleSize` too small, the algorithm will be more susceptible to minor fluctuations in the data, leading to high-anomaly scores for records that are not anomalous.
- If you set the `shingleSize` too large, it may take more time to detect anomalous records because there are more records in the shingle that are not anomalous. It may also take more time to determine that the anomaly has ended.

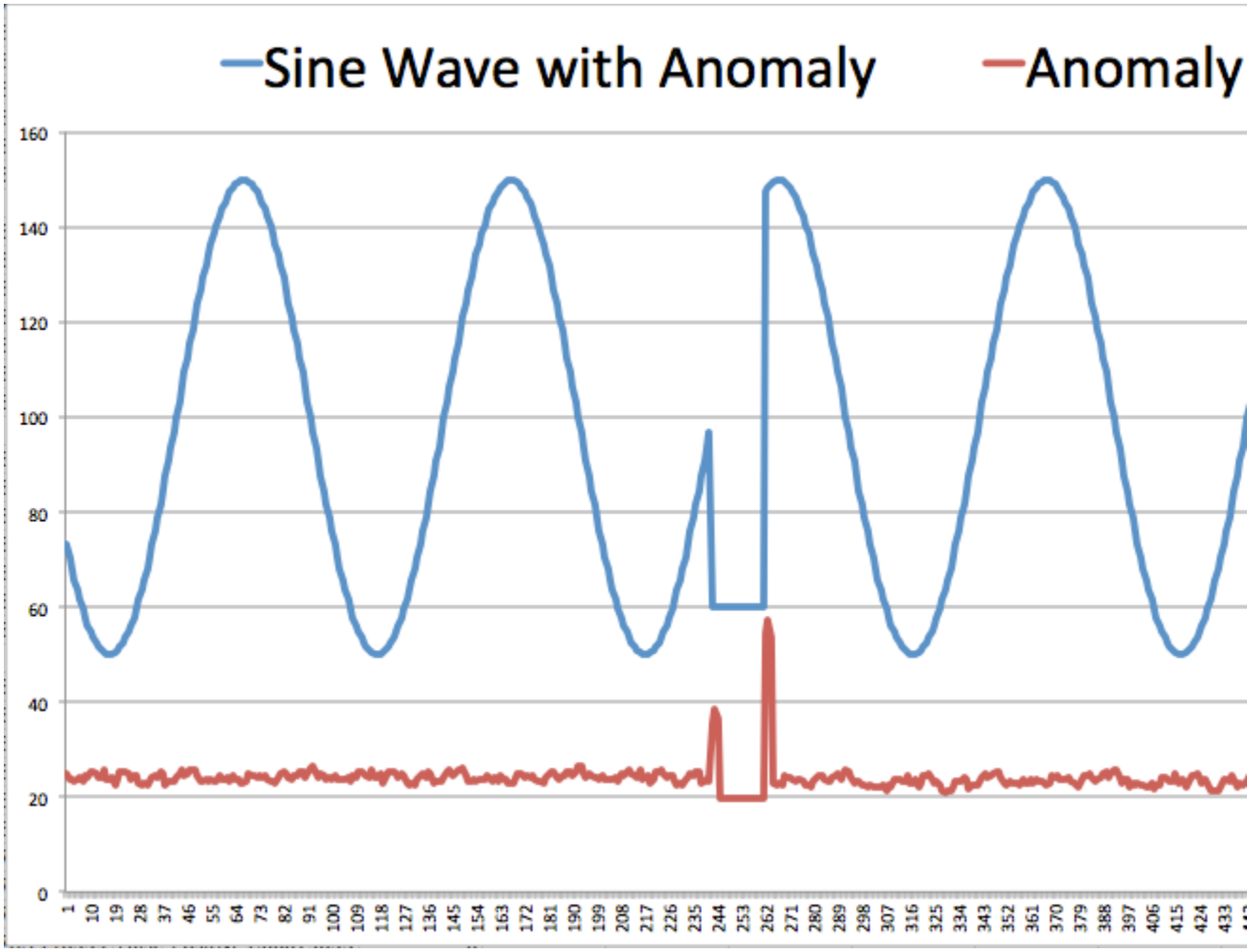
- Identifying the right shingle size is application-dependent. Experiment with different shingle sizes to ascertain the effect.

The following example illustrates how you can catch anomalies when you monitor the records with the highest anomaly score. In this particular example, the two highest anomaly scores also signal the beginning and end of an artificially injected anomaly.

Consider this stylized one-dimensional stream represented as a sine wave, intended to capture a circadian rhythm. This curve illustrates the typical number of orders that an eCommerce site receives per hour, the number of users logged into a server, the number of ad clicks received per hour, etc. A severe dip of 20 consecutive records is artificially injected in the middle of the plot.



We ran the `RANDOM_CUT_FOREST` function with a shingle size of four records. The result is shown below. The red line shows the anomaly score. Note that the beginning and the end of the anomaly received high scores.



When you use this function, we recommend that you investigate the highest scoring points as potential anomalies.

For more information, see the Robust Random Cut Forest Based Anomaly Detection On Streams white paper at the Journal of Machine Learning Research website.

Windowed Aggregation on Streams

To illustrate how windowed aggregation on works on streams, assume the data in the following table is flowing through a stream called WEATHERSTREAM.

ROWTIME	CITY	TEMP
2018-11-01 01:00:00.0	Denver	29
2018-11-01 01:00:00.0	Anchorage	2
2018-11-01 06:00:00.0	Miami	65
2018-11-01 07:00:00.0	Denver	32
2018-11-01 09:00:00.0	Anchorage	9

ROWTIME	CITY	TEMP
2018-11-01 13:00:00.0	Denver	50
2018-11-01 17:00:00.0	Anchorage	10
2018-11-01 18:00:00.0	Miami	71
2018-11-01 19:00:00.0	Denver	43
2018-11-02 01:00:00.0	Anchorage	4
2018-11-02 01:00:00.0	Denver	39
2018-11-02 07:00:00.0	Denver	46
2018-11-02 09:00:00.0	Anchorage	3
2018-11-02 13:00:00.0	Denver	56
2018-11-02 17:00:00.0	Anchorage	2
2018-11-02 19:00:00.0	Denver	50
2018-11-03 01:00:00.0	Denver	36
2018-11-03 01:00:00.0	Anchorage	1

Let's say we want to find the minimum and maximum temperature recorded in the 24-hour period prior to any given reading, globally, regardless of city. To do this, we define a window of RANGE INTERVAL '1' DAY PRECEDING, and use it in the OVER clause for the MIN and MAX analytic functions:

```
SELECT STREAM
       ROWTIME,
       MIN(TEMP) OVER W1 AS WMIN_TEMP,
       MAX(TEMP) OVER W1 AS WMAX_TEMP
FROM WEATHERSTREAM
WINDOW W1 AS (
   RANGE INTERVAL '1' DAY PRECEDING
);
```

Results

ROWTIME	WMIN_TEMP	WMAX_TEMP
2018-11-01 01:00:00.0	29	29
2018-11-01 01:00:00.0	2	29
2018-11-01 06:00:00.0	2	65
2018-11-01 07:00:00.0	2	65
2018-11-01 09:00:00.0	2	65
2018-11-01 13:00:00.0	2	65
2018-11-01 17:00:00.0	2	65
2018-11-01 18:00:00.0	2	71

ROWTIME	WMIN_TEMP	WMAX_TEMP
2018-11-01 19:00:00.0	2	71
2018-11-02 01:00:00.0	2	71
2018-11-02 01:00:00.0	2	71
2018-11-02 07:00:00.0	4	71
2018-11-02 09:00:00.0	3	71
2018-11-02 13:00:00.0	3	71
2018-11-02 17:00:00.0	2	71
2018-11-02 19:00:00.0	2	56
2018-11-03 01:00:00.0	2	56
2018-11-03 01:00:00.0	1	56

Now, let's assume we want to find the minimum, maximum, and average temperature recorded in the 24 hour period prior to any given reading, broken down by city. To do this, we add a PARTITION BY clause on CITY to the window specification, and add the AVG analytic function over the same window to the selection list:

```
SELECT STREAM
       ROWTIME,
       CITY,
       MIN(TEMP) over W1 AS WMIN_TEMP,
       MAX(TEMP) over W1 AS WMAX_TEMP,
       AVG(TEMP) over W1 AS WAVG_TEMP
FROM AGGTEST.WEATHERSTREAM
WINDOW W1 AS (
       PARTITION BY CITY
       RANGE INTERVAL '1' DAY PRECEDING
);
```

Results

ROWTIME	CITY	WMIN_TEMP	WMAX_TEMP	WAVG_TEMP
2018-11-01 01:00:00.0	Denver	29	29	29
2018-11-01 01:00:00.0	Anchorage	2	2	2
2018-11-01 06:00:00.0	Miami	65	65	65
2018-11-01 07:00:00.0	Denver	29	32	30
2018-11-01 09:00:00.0	Anchorage	2	9	5

ROWTIME	CITY	WMIN_TEMP	WMAX_TEMP	WAVG_TEMP
2018-11-01 13:00:00.0	Denver	29	50	37
2018-11-01 17:00:00.0	Anchorage	2	10	7
2018-11-01 18:00:00.0	Miami	65	71	68
2018-11-01 19:00:00.0	Denver	29	50	38
2018-11-02 01:00:00.0	Anchorage	2	10	6
2018-11-02 01:00:00.0	Denver	29	50	38
2018-11-02 07:00:00.0	Denver	32	50	42
2018-11-02 09:00:00.0	Anchorage	3	10	6
2018-11-02 13:00:00.0	Denver	39	56	46
2018-11-02 17:00:00.0	Anchorage	2	10	4
2018-11-02 19:00:00.0	Denver	39	56	46
2018-11-03 01:00:00.0	Denver	36	56	45
2018-11-03 01:00:00.0	Anchorage	1	4	2

Examples of Rowtime Bounds and Windowed Aggregation

This is an example of a windowed aggregate query:

```
SELECT STREAM ROWTIME, ticker, amount, SUM(amount)
   OVER (
       PARTITION BY ticker
       RANGE INTERVAL '1' HOUR PRECEDING)
AS hourlyVolume
FROM Trades
```

Because this is a query on a stream, rows pop out of this query as soon as they go in. For example, given the inputs:

```
Trades: IBM 10 10 10:00:00
Trades: ORCL 20 10:10:00
Trades.bound: 10:15:00
Trades: ORCL 15 10:25:00
```

```
Trades: IBM 30 11:05:00
Trades.bound: 11:10:00
```

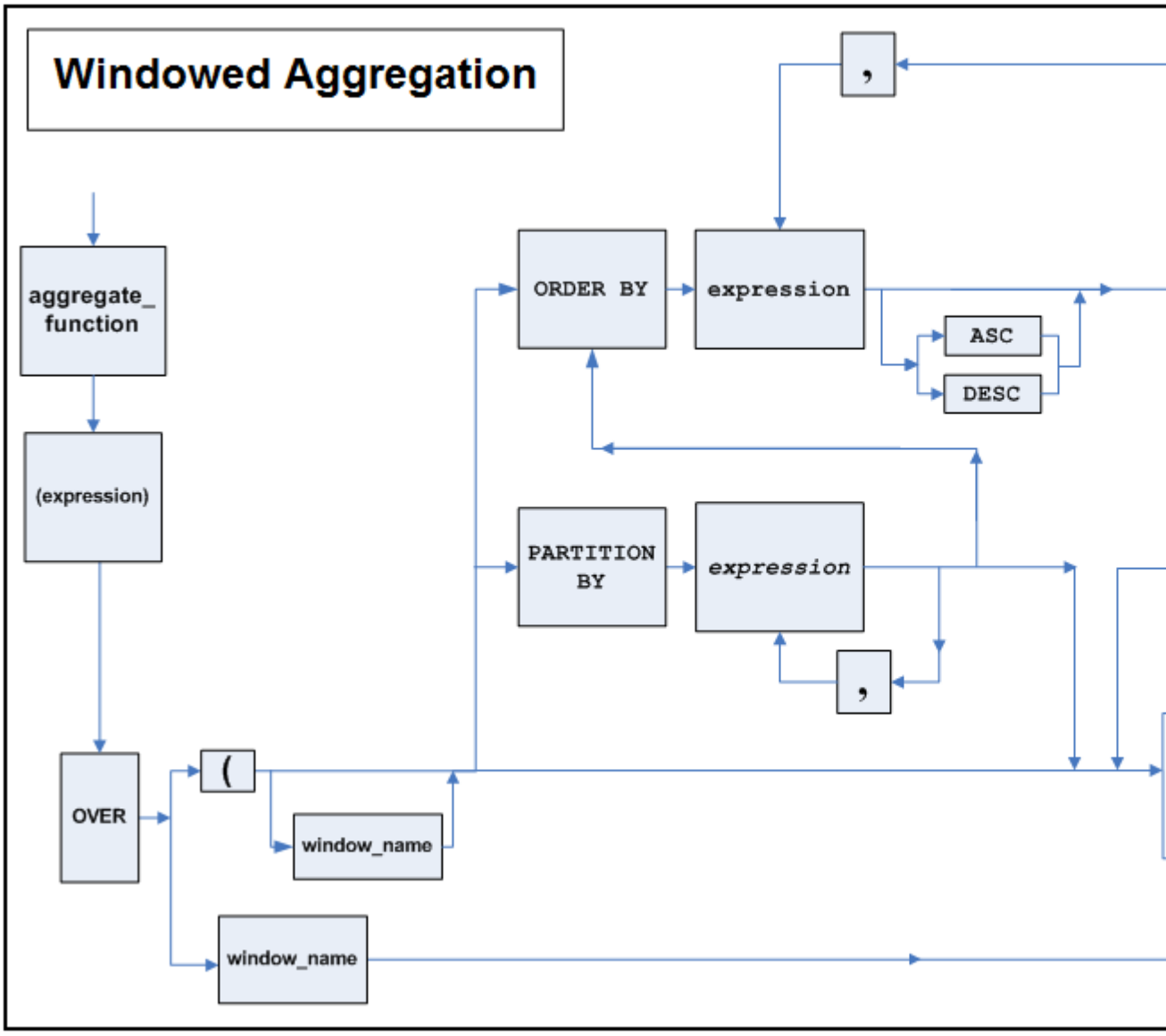
In this example, the output is as follows:

```
Trades: IBM 10 10 10:00:00
Trades: ORCL 20 20 10:10:00
Trades.bound: 10:15:00
Trades: ORCL 15 35 10:25:00
Trades: IBM 30 30 11:05:00
Trades.bound: 11:10:00
```

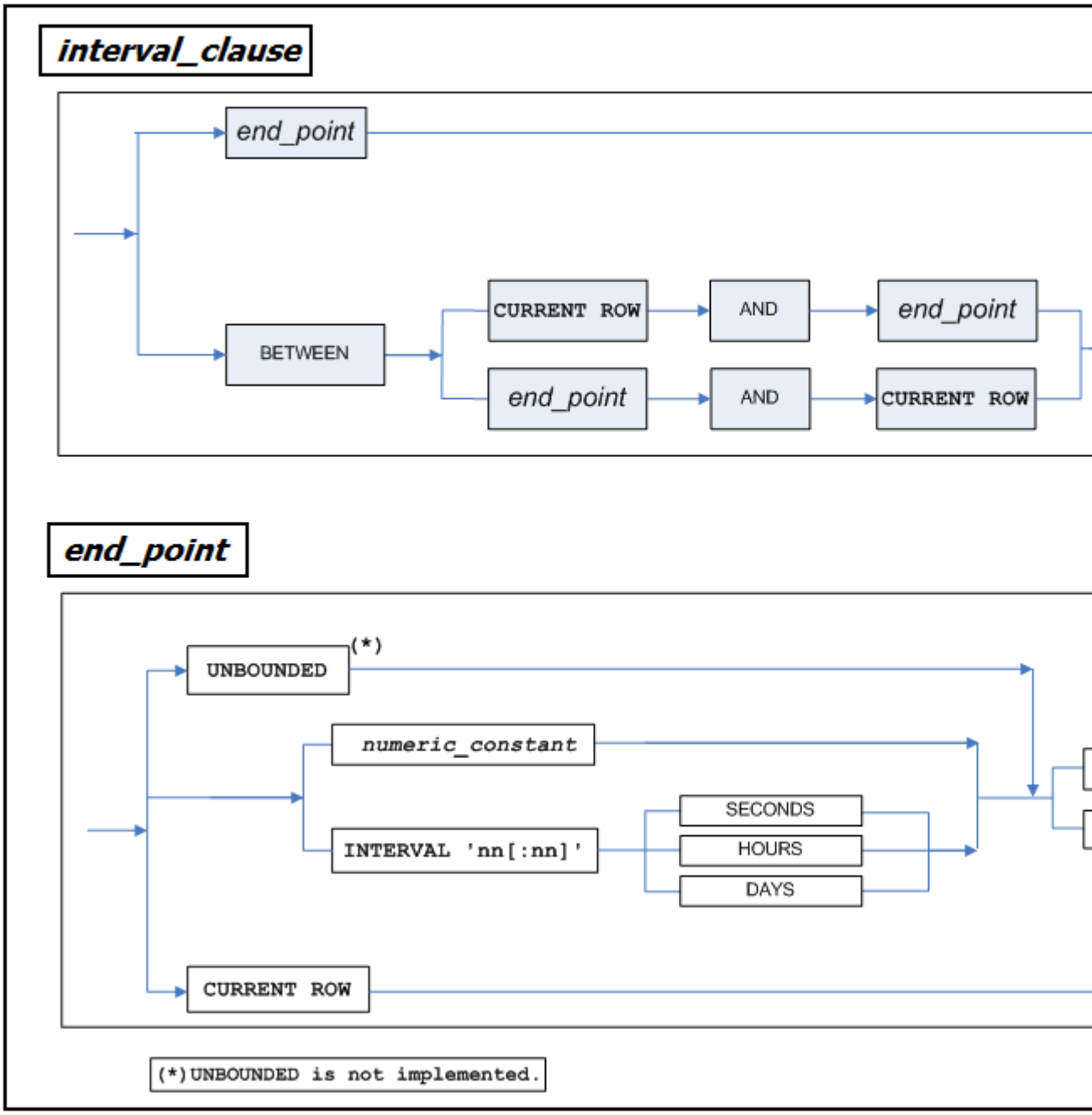
The rows still hang around behind the scenes for an hour, and thus the second ORCL row output has a total of 35; but the original IBM trade falls outside the "hour preceding" window, and so is excluded from the IBM sum.

Syntax Chart for Windowed Aggregation

(To see where windowed-aggregation fits into a SELECT statement, see the topic SELECT statement (p. 134) in this guide.)



Interval Clause



Example

Some business problems seem to need totals over the whole history of a stream, but this is usually not practical to compute. However, such business problems are often solvable by looking at the last day, the last hour, or the last N records. Sets of such records are called windowed aggregates.

They are easy to compute in a stream database, and can be expressed in ANSI (SQL:2008) standard SQL as follows:

```
SELECT STREAM ticker,
     avg(price  OVER  lastHour AS avgPrice,
```

```
    max(price) OVER  lastHour AS maxPrice
FROM Bids
WINDOW lastHour AS  (
    PARTITION BY ticker
    RANGE INTERVAL '1' HOUR PRECEDING)
```

Note
The Interval_clause must be of one of the following appropriate types:

- Integer literal with ROWS
- Numeric value for RANGE over a numeric column
- INTERVAL for a RANGE over a date/time/timestamp

Scalar Functions

Scalar functions are single-row functions (that is, they produce a single result for each input row to a query). The following scalar functions are available in streaming SQL:

Topics

ABS

Returns the absolute value of the input argument. Returns `null` if the input argument is null.

```
ABS ( <numeric-expression>  <interval-expression>
    )
```

Examples

Function	Result
ABS(2.0)	2.0
ABS(-1.0)	1.0
ABS(0)	0
ABS(-3 * 3)	9
ABS(INTERVAL '-3 4:20' DAY TO MINUTE)	INTERVAL '3 4:20' DAY TO MINUTE

If you use `cast as VARCHAR` in SQLline to show the output, the value is returned as `+3 04:20`.

```
values(cast(ABS(INTERVAL '-3 4:20' DAY TO MINUTE) AS VARCHAR(8)));
 +-----------+
   EXPR$0
 +-----------+
  +3 04:20
 +-----------+
 1 row selected
```

CAST

CAST lets you convert one value expression or data type to another value expression or data type.

```
CAST ( <cast-operand> AS <cast-target> )
 <cast-operand> := <value-expression>
 <cast-target>  := <data-type>
```

Valid Conversions

Using CAST with source operands of the types listed in the first column below can create cast target types as listed in the second column, without restriction. Other target types are not supported.

Source Operand Types	Target Operand Types
Any numeric type (NUMERIC, DECIMAL, SMALLINT, INTEGER, BIGINT, REAL, DOUBLE)	VARCHAR, CHAR, or any numeric type (See Note A.)
VARCHAR, CHAR	All of the above, plus, DATE, TIME, TIMESTAMP, DAY-TIME INTERVAL, BOOLEAN
DATE	DATE, VARCHAR, CHAR, TIMESTAMP
TIME	TIME, VARCHAR, CHAR, TIMESTAMP
TIMESTAMP	TIME, VARCHAR, CHAR, TIMESTAMP, DATE
DAY-TIME INTERVAL	DAY-TIME INTERVAL, BIGINT, DECIMAL, CHAR, VARCHAR
BOOLEAN	VARCHAR, CHAR, BOOLEAN
BINARY, VARBINARY	BINARY, VARBINARY

Examples

2.1 DATE to CHAR/VARCHAR

```
+-------------+
|   EXPR$0    |
+-------------+
| 2008-08-23  |
+-------------+
1 row selected
```

(Note that if an inadequate output specification is supplied, no rows are selected:

```
values(cast(date'2008-08-23' as varchar(9)));
'EXPR$0'
No rows selected
```

(Because the date literal requires 10 characters)

In the next case, the date is blank-padded on the right (because of the semantics of the CHAR datatype):

```
+---------------------------+
|           EXPR$0          |
+---------------------------+
| 2008-08-23                |
+---------------------------+
1 row selected
```

REAL to INTEGER

The real (NUMERIC or DECIMAL) is rounded by the cast:

```
+---------+
| EXPR$0  |
+---------+
| -2      |
+---------+
1 row selected
```

STRING to TIMESTAMP

There are two ways to convert a string to a timestamp. The first uses CAST, as shown in the next topic. The other uses Char To Timestamp(Sys) (p. 32).

Using CAST to Convert a String to a Timestamp

The example below illustrates this method for conversion:

```
'EXPR$0'
'2007-02-19 21:23:45'
1 row selected
```

If the input string lacks any one of the six fields (year, month, day, hours, minutes, seconds), or uses any delimiters different from those shown above, CAST will not return a value. (Fractional seconds are disallowed.)

If the input string is thus not in the appropriate format to be CAST, then to convert the string to a timestamp, you must use the CHAR_TO_TIMESTAMP method.

Using CHAR_TO_TIMESTAMP to convert a String to a Timestamp

When the input string is not in the appropriate format to be CAST, you can use the CHAR_TO_TIMESTAMP method. It has the additional advantage that you can specify which parts of the timestamp string you wish to use in subsequent processing, and create a TIMESTAMP value containing only those. To do so, you specify a template that identifies which parts you want, such as 'yyyy-MM' to use only the year and month parts.

The input-date-time string-to-be-converted can contain all or any parts of a full timestamp, that is, values for any or all of the standard elements ('yyyy-MM-dd hh:mm:ss'). If all these elements are present in your input string, and 'yyyy-MM-dd hh:mm:ss' is the template you supply, then the input-string elements are interpreted in that order as year, month, day, hour, minute, and seconds, such as in '2009-09-16 03:15:24'. The yyyy cannot be uppercase; the hh can be uppercase to mean using a 24-hour clock. For many examples of valid specifiers, see the table and examples later in this topic. For the full range of valid specifiers, see Class SimpleDateFormat on the Oracle website.

CHAR_TO_TIMESTAMP uses the template you specify as a parameter in the function call. The template causes the TIMESTAMP result to use only the parts of the input-date-time value that you specified in the template. Those fields in the resulting TIMESTAMP will then contain the corresponding data taken from your input-date-time string; fields not specified in your template will use default values (see below). The format of the template used by CHAR_TO_TIMESTAMP is defined by Class SimpleDateFormat, at which link all the specifiers are listed, some with examples. For more information, see Date and Time Patterns (p. 107).

The function-call syntax is as follows:

```
CHAR_TO_TIMESTAMP('<format_string>','<input_date_time_string>')
```

Where <format_ string> is the template you specify for the parts of <date_time_string> you want, and <input_date_time_string> is the original string that is being converted to a TIMESTAMP result.

Each string must be enclosed in single quotes, and each element of the <input_date_time_string> must be in the range for its corresponding element in the template. Otherwise, no result is returned.

Example 1

- The input-string-element whose position corresponds with MM must be an integer from 1 to 12, because anything else does not represent a valid month.
- The input-string-element whose position corresponds with dd must be an integer from 1 to 31, because anything else does not represent a valid day.
- However, if MM is 2, dd cannot be 30 or 31, because February never has such days.

However, for months or days, the default starting value substituted for the omitted parts is 01.

For example, using '2009-09-16 03:15:24' as your input string, you can obtain a TIMESTAMP containing only the date, with zeros for the other fields such as hours, minutes, or seconds, by specifying

```
CHAR_TO_TIMESTAMP('yyyy-MM-dd','2009-09-16 03:15:24').
```

The result would be the TIMESTAMP 2009-09-16 00:00:00.

Example 2

- If the call had kept hours and minutes in the template while omitting months, days, and seconds, as illustrated in the following call --- --- CHAR_TO_TIMESTAMP('yyyy-hh-mm','2009-09-16 03:15:24') --- --- then the resulting TIMESTAMP would be 2009-01-01 03:15:00.

Template	Input String	Output TIMESTAMP	Notes
`'yyyy-MM-dd hh:mm:ss'`	`'2009-09-16 03:15:24'`	`'2009-09-16 03:15:24'`	Input string MUST use the form 'yyyy-MM-dd hh:mm:ss' or a subset or reordering thereof; using an input string like 'Wednesday, 16 September 2009 03:15:24' will NOT work, meaning that no output will result.
`'yyyy-mm'`	`'2012-02-08 07:23:19'`	`'2012-01-01 00:02:00'`	The template above specifies only year first and minutes second, so the second element in the input string ("02") is used as minutes. Default values are used for Month and Day ("01") and for hours and seconds ("00").
`'yyyy-ss-mm'`	`'2012-02-08 07:23:19'`	`'2012-01-01 00:08:02'`	The template above specifies only year, seconds, and minutes, in that order, so the second element in the input string ("02") is used as seconds and the third as minutes ("08"). Default values are used for Month and Day ("01") and for hours ("00").
`'MMM dd, yyyy'`	`'March 7, 2010'`	`'2010-03-07 00:00:00'`	MMM in the template above matches "March"; the template's 'comma space' matches the input string. --- --- If the template lacks the comma, so must the input string, or there is no output;

Template	Input String	Output TIMESTAMP	Notes
			--- --- If the input string lacks the comma, so must the template.
`'MMM dd,'`	`'March 7, 2010'`	`'1970-03-07 00:00:00'`	Note that the template above doesn't use a year specifier, causing the output TIMESTAMP to use the earliest year in this epoch, 1970.
`'MMM dd,y'`	`'March 7, 2010'`	`'2010-03-07 00:00:00'`	Using the template above, if the input string were 'March 7, 10', the output TIMESTAMP would be '0010-03-07 00:00:00'.
`'M-d'`	`'2-8'`	`'1970-02-08 00:00:00'`	Absent a yyyy specifier in the template, as above, the earliest year in this epoch (1970) is used. An input string of '2-8-2012' would give the same result; using '2012-2-8' would give no result because 2012 is not a valid month.
`'MM-dd-yyyy'`	`'06-23-2012 10:11:12'`	`'2012-06-23 00:00:00'`	Dashes as delimiters (as above) are fine, if template and input both use them in the same positions. Since the template omits hours, minutes, and seconds, zeroes are used in the output TIMESTAMP.

Template	Input String	Output TIMESTAMP	Notes
`'dd-MM-yy hh:mm:ss'`	`'23-06-11 10:11:12'`	`'2011-06-23 10:11:12'`	You can have the specifiers in any order as long as that order matches the meaning of the input string you supply, as above. The template and input string of the next example below have the same meaning (and the same output TIMESTAMP) as this example, but they specify months before days and seconds before hours.
`'MM-dd-yy ss:hh:mm'`	`'06-23-11 12:10:11'`	`'2011-06-23 10:11:12'`	In the template used above, the order of the month and day specifiers is reversed from the example just above, and the specifier for seconds is before hours instead of after minutes; but because the input string also puts months before days and seconds before hours, the meaning (and the output TIMESTAMP) is the same as the example ABOVE.
`'yy-dd-MM ss:hh:mm'`	`'06-23-11 12:10:11'`	`'2006-11-23 10:11:12'`	The template used above reverses (compared to the prior example above) the years and months specifiers, while the input string remains the same. In this case, the output TIMESTAMP uses the first element of the input string as the years, the second as the days, and the third as the months.

Template	Input String	Output TIMESTAMP	Notes
`'dd-MM-yy hh:mm'`	`'23-06-11 10:11:12'`	`'2011-06-23 10:11:00'`	With seconds omitted in the template, as above, the output TIMESTAMP uses 00 seconds. Any number of y specifiers produces the same result; but if the input string inadvertently uses a 1 instead of 11 for the year, as in '23-06-1 10:11:12', then the output TIMESTAMP becomes '0001-06-23 10:11:00'.

`'MM/dd/yy hh:mm:ss'`	`'12/19/11 10:11:12'`
	
	`'12/19/11 12:11:10'`
		`'2011-12-19 10:11:12'`
		
		`'2011-12-19 00:11:10'`
			Slashes as delimiters are fine, if template and input both use them in the same positions, as above; otherwise, no output.
			
			Using specifier hh, input times of 12:11:10 and 00:11:10 have the same meaning as a time in the morning.

Template	Input String	Output TIMESTAMP	Notes

`'MM/dd/yy HH:mm:ss'`	`'12/19/11 12:59:59'`
	`'12/19/11 21:08:07'`
		`'2011-12-19 12:59:59'`
		`'2011-12-19 21:08:07'`
			The input-string values '2011-12-19 00:11:12' or '2011-12-19 12:11:12' would fail with this template because '2011' is not a month, as required/expected by the template-string 'MM/dd/yy HH:mm:ss'.
			
			However, changing the template gives useful output:
			
			`values(cast(CHAR_TO_TIMESTAMP('y MM/dd HH:mm:ss', '2011/12/19 00:11:12') as varchar(19)));`
			`'EXPR$0'`
			`'2011-12-19 00:11:12'`
			`1 row selected`
			
			`'12/19/11 00:11:12'` would fail with the above template (`'y/MM/dd'`), since 19 is not a valid month; supplying `'12/11/19 00:11:12'` works. `'2011-12-19 12:11:12'` would fail as input because dashes don't match the slashes in the template ; `'2011/12/19 12:11:12'` works.
			
			Note that for times after 12 noon, that is, for afternoon and evening times, the hours specifier must be HH instead of hh, and the input string must specify the afternoon or evening hour in 24-hour clock time, hours running from 00 to 23.

Template	Input String	Output TIMESTAMP	Notes
			--- --- Using specifier HH, input times of 12:11:10 and 00:11:10 have different meanings, the first as a time in the afternoon and the second as a time in the morning. --- --- Using the specifier hh, the times from 12:00 through 11:59:59 are morning times: --- --- Given the specifiers hh:mm:ss, the output TIMESTAMP will include '00:09:08' in the morning for both input string '12:09:08' and input string '00:09:08'; --- --- whereas --- --- Given the specifiers HH:mm:ss, the output TIMESTAMP for input string '00:09:08' in the morning will include '00:09:08' --- --- and the output TIMESTAMP for input string '12:09:08' in the afternoon will include '12:09:08'.

The examples below illustrate using various templates with CHAR_TO_TIMESTAMP, including some common misunderstandings.

```
 values (CHAR_TO_TIMESTAMP('yyyy-hh-mm','2009-09-16 03:15:24'));
'EXPR$0'
'2009-01-01 09:16:00'
1 row selected
```

Note that the fields in the input string above were used in the order given by the specifiers in the template, as defined by the dashes-as-delimiters in both template and input string: years first, then hours, then minutes. Since the specifiers for months and days are not present in the template, their values in the input string were ignored, with 01 substituted for both values in the output TIMESTAMP. The template specified hours and minutes as the second and third input values, so 09 became the hours and 16 became the minutes. No specifier was present for seconds, so 00 was used.

The years specifier can be alone or, after a delimiter matching the input string shows the end of the years specifier, with one of the hours:minutes:seconds specifiers:

```
values (CHAR_TO_TIMESTAMP('yyyy','2009-09-16 03:15:24') );
'EXPR$0'
'2009-01-01 00:00:00'
1 row selected
values (CHAR_TO_TIMESTAMP('yyyy hh','2009-09-16 03:15:24') );
'EXPR$0'
No rows selected
```

The template above fails because it has a space-as-delimiter before the "hh" rather than the dash delimiter used in the input string's date specification;

whereas the four templates below work because they use the same delimiter to separate the years specifier from the next specifier as is used in the input string's date specification (dash in the first case, space in the second, slash in the third, and dash in the fourth).

```
values (CHAR_TO_TIMESTAMP('yyyy-hh','2009-09-16 03:15:24') );
'EXPR$0'
'2009-01-01 09:00:00'
1 row selected
values (CHAR_TO_TIMESTAMP('yyyy hh','2009 09 16 03:15:24') );
'EXPR$0'
'2009-01-01 09:00:00'
1 row selected
values (CHAR_TO_TIMESTAMP('yyyy/hh','2009/09/16 03:15:24') );
'EXPR$0'
'2009-01-01 09:00:00'
1 row selected
values (CHAR_TO_TIMESTAMP('yyyy-mm','2009-09-16 03:15:24') );
'EXPR$0'
'2009-01-01 00:09:00'
1 row selected
```

However, if the template specifies months (MM), it cannot then specify hours, minutes, or seconds unless days are also specified:

Template specifying years and months only, thus omitting days/hours/minutes/seconds from the resulting TIMESTAMP:

```
values (CHAR_TO_TIMESTAMP('yyyy-MM','2009-09-16 03:15:24') );
'EXPR$0'
'2009-09-01 00:00:00'
1 row selected
```

The next two templates fail, lacking a 'days' specifier:

```
values (CHAR_TO_TIMESTAMP('yyyy-MM hh','2009-09-16 03:15:24') );
'EXPR$0'
No rows selected
values (CHAR_TO_TIMESTAMP('yyyy-MM hh:','2009-09-16 03:15:24') );
'EXPR$0'
No rows selected
```

The next three succeed, using a 'days' specifier:

```
 values (CHAR_TO_TIMESTAMP('yyyy-MM-dd hh','2009-09-16 03:15:24') );
'EXPR$0'
'2009-09-16 03:00:00'
1 row selected
```

The template above, 'yyyy-MM-dd hh', specifies only hours (hh) without minutes or seconds. Since hh is the 4th token/element of the template, its value is to be taken from the 4th token/element of the input string '2009-09-16 03:15:24' ; and that 4th element is 03, then used as the value output for hours. Since neither mm or ss is specified, the default or initial values defined as the starting point for mm and ss are used, which are zeroes.

```
values (CHAR_TO_TIMESTAMP('yyyy-MM-dd ss','2009-09-16 03:15:24') );
'EXPR$0'
'2009-09-16 00:00:03'
1 row selected
```

The template above, 'yyyy-MM-dd ss', specifies that the 4th token/element of the input string is to be used as seconds (ss). The 4th element of the input string '2009-09-16 03:15:24' is 03, which becomes the value output for seconds as specified in the template; and since neither hh nor mm is specified in the template, their default or initial values are used, which are zeroes.

```
values (CHAR_TO_TIMESTAMP('yyyy-MM-dd mm','2009-09-16 03:15:24') );
'EXPR$0'
'2009-09-16 00:03:00'
1 row selected
```

The template above, 'yyyy-MM-dd mm', specifies that the 4th token/element of the input string is to be used as minutes (mm). The 4th element of the input string '2009-09-16 03:15:24' is 03, which becomes the value output for minutes as specified in the template; and since neither hh nor ss is specified in the template, their default or initial values are used, which are zeroes.

Further failures, lacking a 'days' specifier:

```
values (CHAR_TO_TIMESTAMP('yyyy-MM- mm','2009-09-16 03:15:24') );
'EXPR$0'
No rows selected
values (CHAR_TO_TIMESTAMP('yyyy-MM   mm','2009-09-16 03:15:24') );
'EXPR$0'
No rows selected
values (CHAR_TO_TIMESTAMP('yyyy-MM   hh','2009-09-16 03:15:24') );
'EXPR$0'
No rows selected
```

About Delimiters and Values

Delimiters in the template must match those in the input string; values in the input string must be acceptable for the template specifiers to which they correspond.

As a general convention, a colon is used to separate hours from minutes, and minutes from seconds. Similarly, the general convention is to use a dash or slash to separate years from months and months from days. Any parallel usage seems to work, and the examples that follow illustrate this.

```
values (CHAR_TO_TIMESTAMP('MM/dd/yy hh:mm:ss','2009/09/16 03:15:24') );
'EXPR$0'
```

```
No rows selected
```

The example above fails because 2009 is not an acceptable value for months, which is the first specifier (MM) in the template.

```
values (CHAR_TO_TIMESTAMP('MM/dd/yy hh:mm:ss','09/16/11 03:15:24') );
'EXPR$0'
'2011-09-16 03:15:24'
1 row selected
```

The example above succeeds because the delimiters are parallel (slashes to slashes, colons to colons) and each value is acceptable for the corresponding specifier.

```
values (CHAR_TO_TIMESTAMP('MM/dd/yy hh/mm/ss','09/16/11 03/15/24') );
'EXPR$0'
'2011-09-16 03:15:24'
1 row selected
```

The example above succeeds because the delimiters are parallel (all slashes) and each value is acceptable for the corresponding specifier.

```
values (CHAR_TO_TIMESTAMP('MM/dd/yy hh-mm-ss','09/16/11 03-15-24') );
'EXPR$0'
'2011-09-16 03:15:24'
1 row selected
values (CHAR_TO_TIMESTAMP('yyyy|MM|dd hh|mm|ss','2009|09|16 03|15|24') );
'EXPR$0'
'2009-09-16 03:15:24'
1 row selected
values (CHAR_TO_TIMESTAMP('yyyy@MM@dd hh@mm@ss','2009@09@16 03@15@24') );
'EXPR$0'
'2009-09-16 03:15:24'
1 row selected
```

The examples above succeed because the delimiters are parallel and the values are acceptable per specifier.

In the following examples, note that omissions in the supplied string can cause the template value 'yyyy' to produce logical but unintended or unexpected results. The value given as the year in the resulting TIMESTAMP value derives directly from the first element in the supplied string.

```
 VALUES(CHAR_TO_TIMESTAMP('yyyy','09-16 03:15'));
'EXPR$0'
'0009-01-01 00:00:00'
1 row selected
VALUES(CHAR_TO_TIMESTAMP('yyyy','16 03:15'));
'EXPR$0'
'0016-01-01 00:00:00'
1 row selected
```

TIMESTAMP to STRING

```
 values(cast( TIMESTAMP '2007-02-19 21:25:35' AS VARCHAR(25)));
'EXPR$0'
```

```
'2007-02-19 21:25:35'
1 row selected
```

Note that CAST requires a TIMESTAMP-literal to have literally the full format of 'yyyy-mm-dd hh:mm:ss'. If any part of that full format is missing, the literal is rejected as illegal, as seen below:

```
 values( TIMESTAMP '2007-02-19 21:25');
Error: Illegal TIMESTAMP literal '2007-02-19 21:25':
                                  not in format 'yyyy-MM-dd
 HH:mm:ss' (state=,code=0)
 values( TIMESTAMP '2007-02-19 21:25:00');
'EXPR$0'
'2007-02-19 21:25:00'
1 row selected
```

Also, if an inadequate output specification is supplied, no rows are selected:

```
values(cast( TIMESTAMP '2007-02-19 21:25:35' AS VARCHAR(18)));
'EXPR$0'
No rows selected
(Because the timestamp literal requires 19 characters)
```

These restrictions apply similarly to CASTing to TIME or DATE types.

STRING to TIME

```
 values(cast(' 21:23:45.0' AS TIME));
'EXPR$0'
'21:23:45'
1 row selected
```

For more information, see Note A.

STRING to DATE

```
 values(cast('2007-02-19' AS DATE));
'EXPR$0'
'2007-02-19'
1 row selected
```

Note A
Note that CAST for strings requires that the string operand for casting to TIME or DATE have the exact form required to represent a TIME or DATE, respectively.

As shown below, the cast fails if:

- the string operand includes data extraneous to the targeted type, or
- the INTERVAL operand ('day hours:minutes:seconds.milliseconds') does not include necessary data, or
- the specified output field is too small to hold the conversion results.

```
 values(cast('2007-02-19 21:23:45.0' AS TIME));
```

```
'EXPR$0'
No rows selected
```

Fails because it includes date information not allowed as a TIME.

```
 values(cast('2007-02-19 21:23:45.0' AS DATE));
'EXPR$0'
No rows selected
```

Fails because it includes time information not allowed as a DATE.

```
 values(cast('2007-02-19 21' AS DATE));
'EXPR$0'
No rows selected
```

Fails because it includes time information not allowed as a DATE.

```
 values(cast('2009-02-28' AS DATE));
'EXPR$0'
'2009-02-28'
1 row selected
```

Succeeds because it includes a correct representation of date string.

```
 values(CAST (cast('2007-02-19 21:23:45.0' AS TIMESTAMP) AS DATE));
'EXPR$0'
'2007-02-19'
1 row selected
```

Succeeds because it correctly converts string to TIMESTAMP before casting to DATE.

```
 values(cast('21:23' AS TIME));
'EXPR$0'
No rows selected
```

Fails because it lacks time information (seconds) required for a TIME.

(Specifying fractional seconds is allowed but not required.)

```
 values(cast('21:23:34:11' AS TIME));
'EXPR$0'
No rows selected
```

Fails because it includes incorrect representation of fractional seconds.

```
 values(cast('21:23:34.11' AS TIME));
'EXPR$0'
'21:23:34'
1 row selected
```

Succeeds because it includes correct representation of fractional seconds.

```
 values(cast('21:23:34' AS TIME));
'EXPR$0'
'21:23:34'
1 row selected
```

This example succeeds because it includes correct representation of seconds without fractions of a second.

INTERVAL to exact numerics

CAST for intervals requires that the INTERVAL operand have only one field in it, such as MINUTE, HOUR, SECOND.

If the INTERVAL operand has more than one field, such as MINUTE TO SECOND, the cast fails, as shown below:

```
 values ( cast (INTERVAL '120' MINUTE(3) as decimal(4,2)));
+---------+
| EXPR$0  |
+---------+
+---------+
No rows selected

 values ( cast (INTERVAL '120' MINUTE(3) as decimal(4)));
+---------+
| EXPR$0  |
+---------+
| 120     |
+---------+
1 row selected

 values ( cast (INTERVAL '120' MINUTE(3) as decimal(3)));
+---------+
| EXPR$0  |
+---------+
| 120     |
+---------+
1 row selected

 values ( cast (INTERVAL '120' MINUTE(3) as decimal(2)));
+---------+
| EXPR$0  |
+---------+
+---------+
No rows selected

 values cast(interval '1.1' second(1,1) as decimal(2,1));
+---------+
| EXPR$0  |
+---------+
| 1.1     |
+---------+
1 row selected

 values cast(interval '1.1' second(1,1) as decimal(1,1));
+---------+
| EXPR$0  |
+---------+
```

```
+---------+
No rows selected
```

For year, decimal fractions are disallowed as input and as output.

```
values cast(interval '1.1' year (1,1) as decimal(2,1));
Error: org.eigenbase.sql.parser.SqlParseException: Encountered "," at line 1,
 column 35.
Was expecting:
    ")" ... (state=,code=0)
values cast(interval '1.1' year (1) as decimal(2,1));
Error: From line 1, column 13 to line 1, column 35:
              Illegal interval literal format '1.1' for INTERVAL YEAR(1)
 (state=,code=0)
values cast(interval '1.' year (1) as decimal(2,1));
Error: From line 1, column 13 to line 1, column 34:
              Illegal interval literal format '1.' for INTERVAL YEAR(1)
 (state=,code=0)
values cast(interval '1' year (1) as decimal(2,1));
+---------+
| EXPR$0  |
+---------+
| 1.0     |
+---------+
1 row selected
```

For additional examples, see SQL Operators: Further examples.

Limitations

Amazon Kinesis Analytics does not support directly casting numeric values to interval values. This is a departure from the SQL:2008 standard. The recommended way to convert a numeric to an interval is to multiply the numeric value against a specific interval value. For example, to convert the integer time_in_millis to a day-time interval:

```
time_in_millis * INTERVAL '0 00:00:00.001' DAY TO SECOND
```

For example:

```
 values cast( 5000 * (INTERVAL '0 00:00:00.001' DAY TO SECOND) as
 varchar(11));
'EXPR$0'
'5000'
1 row selected
```

CEIL / CEILING

```
 CEIL | CEILING ( <number-expression> )
 CEIL | CEILING ( <datetime-expression> TO <time-unit> )
 CEIL | CEILING ( <number-expression> )
 CEIL | CEILING ( <datetime-expression> TO <[[time-unit> )
```

When called with a numeric argument, CEILING returns the smallest integer equal to or larger than the input argument.

When called with a date, time, or timestamp expression, CEILING returns the smallest value equal to or larger than the input, subject to the precision specified by the <time unit>.

Returns null if any input argument is null.

Examples

Function	Result
CEIL(2.0)	2
CEIL(-1.0)	-1
CEIL(5.2)	6
CEILING(-3.3)	-3
CEILING(-3 * 3.1)	-9
CEILING(TIMESTAMP '2004-09-30 13:48:23' TO HOUR)	TIMESTAMP '2004-09-30 14:00:00'
CEILING(TIMESTAMP '2004-09-30 13:48:23' TO MINUTE)	TIMESTAMP '2004-09-30 13:49:00'
CEILING(TIMESTAMP '2004-09-30 13:48:23' TO DAY)	TIMESTAMP '2004-10-01 00:00:00.0'
CEILING(TIMESTAMP '2004-09-30 13:48:23' TO YEAR)	TIMESTAMP '2005-01-01 00:00:00.0'

Notes

- CEIL and CEILING are synonyms for this function provided by the SQL:2008 standard.
- CEIL(<datetime value expression> TO <time unit>) is an Amazon Kinesis Analytics extension.
- For more information, see FLOOR (p. 99).

CHAR_LENGTH / CHARACTER_LENGTH

```
CHAR_LENGTH | CHARACTER_LENGTH ( <character-expression> )
```

Returns the length in characters of the string passed as the input argument. Returns null if input argument is null.

Examples

`CHAR_LENGTH('one')`	`3`
`CHAR_LENGTH('')`	`0`
`CHARACTER_LENGTH('fred')`	`4`

`CHARACTER_LENGTH( cast (null as varchar(16) )`	`null`
`CHARACTER_LENGTH( cast ('fred' as char(16) )`	`16`

Limitations

Amazon Kinesis Analytics streaming SQL does not support the optional USING CHARACTERS | OCTETS clause. This is a departure from the SQL:2008 standard.

COALESCE

```
COALESCE (
      <value-expression>
      {,<value-expression>}... )
```

The COALESCE function takes a list of expressions (all of which must be of the same type) and returns the first non-null argument from the list. If all of the expressions are null, COALESCE returns null.

Examples

Expression	Result
COALESCE('chair')	chair
COALESCE('chair', null, 'sofa')	chair
COALESCE(null, null, 'sofa')	sofa
COALESCE(null, 2, 5)	2

EXP

```
EXP ( <number-expression> )
```

Returns the value of e (approximately 2.7182818284590455) raised to the power of the input argument. Returns null if the input argument is null.

Examples

Function	Result
EXP(1)	2.7182818284590455
EXP(0)	1.0
EXP(-1)	0.36787944117144233
EXP(10)	22026.465794806718

Function	Result
EXP(2.5)	12.182493960703473

EXTRACT

```
EXTRACT(YEAR|MONTH|DAY|HOUR|MINUTE|SECOND FROM <datetime expression>|
<interval expression>)
```

The EXTRACT function extracts one field from a DATE, TIME, TIMESTAMP or INTERVAL expression. Returns BIGINT for all fields other than SECOND. For SECOND it returns DECIMAL(5,3) and includes milliseconds.

Syntax

Examples

Function	Result
`EXTRACT(DAY FROM INTERVAL '2 3:4:5.678' DAY TO SECOND)`	2
`EXTRACT(HOUR FROM INTERVAL '2 3:4:5.678' DAY TO SECOND)`	3
`EXTRACT(MINUTE FROM INTERVAL '2 3:4:5.678' DAY TO SECOND)`	4
`EXTRACT(SECOND FROM INTERVAL '2 3:4:5.678' DAY TO SECOND)`	5.678
`EXTRACT(MINUTE FROM CURRENT_ROW_TIMESTAMP) where CURRENT_ROW_TIMESTAMP is 2016-09-23 04:29:26.234`	29
`EXTRACT (HOUR FROM CURRENT_ROW_TIMESTAMP)` where CURRENT_ROW_TIMESTAMP is 2016-09-23 04:29:26.234	4

Use in Function

EXTRACT can be used for conditioning data, as in the following function which returns a 30 minute floor when CURRENT_ROW_TIMESTAMP (p. 110) is input for p_time.

```
CREATE or replace FUNCTION FLOOR30MIN( p_time TIMESTAMP )
```

```
RETURNS  TIMESTAMP
CONTAINS SQL
RETURNS NULL ON NULL INPUT
RETURN  floor(p_time to HOUR) + (( EXTRACT (  MINUTE FROM p_time  ) / 30)*
 INTERVAL '30' MINUTE ) ;
```

You would implement this function using code along the following lines:

```
SELECT stream FLOOR30MIN( CURRENT_ROW_TIMESTAMP ) as ROWTIME , * from
 "MyStream" ) over (range current row ) as r
```

Note
The code above assumes that you have previously created a stream called "MyStream."

FLOOR

```
FLOOR ( <time-unit> )
```

When called with a numeric argument, FLOOR returns the largest integer equal to or smaller than the input argument.

When called with a date, time, or timestamp expression, FLOOR returns the largest value equal to or smaller than the input, subject to the precision specified by <time unit>.

Returns null if any input argument is null.

Examples

Function	Result
FLOOR(2.0)	2
FLOOR(-1.0)	-1
FLOOR(5.2)	5
FLOOR(-3.3)	-4
FLOOR(-3 * 3.1)	-10
FLOOR(TIMESTAMP '2004-09-30 13:48:23' TO HOUR)	TIMESTAMP '2004-09-30 13:00:00'
FLOOR(TIMESTAMP '2004-09-30 13:48:23' TO MINUTE)	TIMESTAMP '2004-09-30 13:48:00'
FLOOR(TIMESTAMP '2004-09-30 13:48:23' TO DAY)	TIMESTAMP '2004-09-30 00:00:00.0'
FLOOR(TIMESTAMP '2004-09-30 13:48:23' TO YEAR)	TIMESTAMP '2004-01-01 00:00:00.0'

Note

- FLOOR (<datetime expression> TO <timeunit>) is a Amazon Kinesis Analytics extension.

- For more information, see CEIL / CEILING (p. 95).

INITCAP

```
INITCAP ( <character-expression> )
```

Returns a converted version of the input string such that the first character of each space-delimited word is upper-cased, and all other characters are lower-cased.

Examples

Function	Result
INITCAP('each first letter is cAPITALIZED')	Each first letter is capitalized
INITCAP('')	<empty string>
INITCAP(cast(null as varchar(3)))	<null>

Note
The INITCAP function is not part of the SQL:2008 standard. It is an Amazon Kinesis Analytics extension.

LN

```
LN ( <number-expression> )
```

Returns the natural log (that is, the log with respect to base e) of the input argument. If the argument is negative or 0, an exception is raised. Returns null if the input argument is null.

For more information, see LOG10 (p. 100) and EXP (p. 97).

Examples

Function	Result
LN(1)	0.0
LN(10)	2.302585092994046
LN(2.5)	0.9162907318741551

LOG10

```
LOG10 ( <number-expression> )
```

Returns the base 10 logarithm of the input argument. If the argument is negative or 0, an exception is raised. Returns null if the input argument is null.

Examples

Function	Result
LOG10(1)	0.0
LOG10(100)	2.0
log10(cast('23' as decimal))	1.3617278360175928

Note
LOG10 is not a SQL:2008 standard function; it is a Amazon Kinesis Analytics extension to the standard.

LOWER

```
LOWER ( <character-expression> )
```

Converts a string to all lower-case characters. Returns null if input argument is null, and the empty string if the input argument is an empty string.

Examples

Function	Result
LOWER('abcDEFghi123')	abcdefghi123

MOD

```
MOD ( <dividend>, <divisor> )
 <dividend> := <integer-expression>
 <divisor>  := <integer-expression>
```

Returns the remainder when the first argument (the dividend is divided by the second numeric argument (the divisor). If the divisor is zero, a divide by zero error is raised.

Examples

Function	Result
MOD(4,2)	0
MOD(5,3)	2
MOD(-4,3)	-1
MOD(5,12)	5

Limitations

The Amazon Kinesis Analytics MOD function only supports arguments of scale 0 (integers). This is a departure from the SQL:2008 standard, which supports any numeric argument. Other numeric arguments can be CAST to an integer, of course.

NULLIF

```
NULLIF ( <value-expression>, <value-expression> )
```

Returns null if the two input arguments are equal, otherwise returns the first value. Both arguments must be of comparable type, or an exception is raised.

Examples

Function	Result
NULLIF(4,2)	4
NULLIF(4,4)	<null>
NULLIF('amy','fred')	amy
NULLIF('amy', cast(null as varchar(3)))	amy
NULLIF(cast(null as varchar(3)),'fred')	<null>

OVERLAY

```
OVERLAY ( <original-string>
          PLACING <replacement-string>
          FROM <start-position>
          [ FOR <string-length> ]
        )
<original-string> := <character-expression>
<replacement-string> := <character-expression>
<start-position> := <integer-expression>
<string-length> := <integer-expression>
```

The OVERLAY function is used to replace a portion of the first string argument (the original string) with the second string argument (the replacement string).

The start position indicates the character position in the original string where the replacement string should be overlaid. The optional string length parameter determines how many characters of the original string to replace (if not specified, it defaults to the length of the replacement string). If there are more characters in the replacement string than are left in the original string, the remaining characters are simply appended.

If the start position is greater than the length of the original string, the replacement string is simply appended. If the start position is less than 1, then (1 - start position) characters of the replacement string is prepended to the result, and the rest overlaid on the original (see examples below).

If the string length is less than zero, an exception is raised.

If any of the input arguments are null, the result is null.

Examples

Function	Result
OVERLAY ('12345' PLACING 'foo' FROM 1)	foo45

Function	Result
OVERLAY ('12345' PLACING 'foo' FROM 0)	foo345
OVERLAY ('12345' PLACING 'foo' FROM -2)	foo12345
OVERLAY ('12345' PLACING 'foo' FROM 4)	123foo
OVERLAY ('12345' PLACING 'foo' FROM 17)	12345foo
OVERLAY ('12345' PLACING 'foo' FROM 2 FOR 0)	1foo2345
OVERLAY ('12345' PLACING 'foo' FROM 2 FOR 2)	1foo45
OVERLAY ('12345' PLACING 'foo' FROM 2 FOR 9)	1foo

Limitations

Amazon Kinesis Analytics does not support the optional USING CHARACTERS | OCTETS clause defined in SQL:2008; USING CHARACTERS is simply assumed. Strict SQL:2008 also requires that a start position less than 1 return a null result, rather than the behavior described above. These are departures from the standard.

POSITION

```
POSITION ( <search-string> IN <source-string> )
search-string := <character-expression>
source-string := <character-expression>
```

The POSITION function searches for the first input argument (the search string) within the second input argument (the source string).

If the search string is found within the source string, POSITION returns the character position of the first instance of the search string (subsequent instances are ignored). If the search string is the empty string, POSITION returns 1.

If the search string is not found, POSITION returns 0.

If either the search string or the source string is null, POSITION returns null.

Examples

Function	Result
POSITION ('findme' IN '1234findmeXXX')	5
POSITION ('findme' IN '1234not-hereXXX')	0
POSITION ('1' IN '1234567')	1
POSITION ('7' IN '1234567')	7
POSITION ('' IN '1234567')	1

Limitations

Amazon Kinesis Analytics streaming SQL does not support the optional USING CHARACTERS | OCTETS clause defined in SQL:2008; USING CHARACTERS is simply assumed. This is a departure from the standard.

POWER

```
POWER ( <base>, <exponent> )
<base> := <number-expression>
<exponent> := <number-expression>
```

Returns the value of the first argument (the base) raised to the power of the second argument (the exponent). Returns null if either the base or the exponent is null, and raises an exception if the base is zero and the exponent is negative, or if the base is negative and the exponent is not a whole number.

Examples

Function	Result
POWER(3,2)	9
POWER(-2,3)	-8
POWER(4,-2)	1/16 ..or.. 0.0625
POWER(10.1,2.5)	324.19285157140644

SUBSTRING

```
SUBSTRING ( <source-string> FROM <start-position> [ FOR <string-length> ] )
SUBSTRING ( <source-string>, <start-position> [ , <string-length> ] )
<source-string> := <character-expression>
<start-position> := <integer-expression>
<string-length> := <integer-expression>
```

SUBSTRING extracts a portion of the source-string specified in the first argument, starting at start-position.

If string-length is specified, only string-length characters are returned (if there aren't that many characters left in the string, only the characters that are left are returned). If string-length is not specified, it defaults to the remaining length of the input string.

If the start position is less than 1, then it is interpreted as if the start position is 1 and the string length is reduced by (1 - start position). See examples below. If the start position is greater than the number of characters in the string, or the length parameter is 0, the result is an empty string.

Examples

Function	Result
SUBSTRING('123456789' FROM 3 FOR 4)	3456
SUBSTRING('123456789', 3, 4)	3456
SUBSTRING('123456789' FROM -1 FOR 4)	12

Function	Result
SUBSTRING('123456789' FROM 8 FOR 4)	89
SUBSTRING('123456789' FROM 17 FOR 4)	<empty string>
SUBSTRING('123456789' FROM 6 FOR 0)	<empty string>

Limitations

- Amazon Kinesis Analytics streaming SQL does not support the regular expression form of SUBSTRING defined in SQL:2008 (that is, 'SUBSTRING ... SIMILAR ... ESCAPE').
- Additionally, Amazon Kinesis Analytics streaming SQL does not support the optional 'USING CHARACTERS | OCTETS' clause defined in SQL:2008; USING CHARACTERS is simply assumed.
- Finally, the second form of the SUBSTRING function listed above (using commas rather than FROM...FOR) is not part of the SQL:2008 standard; it is a Amazon Kinesis Analytics streaming SQL extension.

TRIM

```
TRIM ( [ [ <trim-specification> ] [ <trim-character> ] FROM ] <trim-source> )
 <trim-specification> := LEADING | TRAILING | BOTH
 <trim-character> := <character-expression>
 <trim-source> := <character-expression>
```

TRIM removes instances of the specified trim-character from the beginning and/or end of the trim-source string as dictated by the trim-specification (that is, LEADING, TRAILING, or BOTH). If LEADING is specified, only repetitions of the trim character at the beginning of the source string are removed. If TRAILING is specified, only repetitions of the trim character at the end of the source string are removed. If BOTH is specified, or the trim specifier is left out entirely, then repetitions are removed from both the beginning and end of the source string.

If the trim-character is not explicitly specified, it defaults to the space character (' '). Only one trim character is allowed; specifying an empty string or a string longer than one character results in an exception.

If either input is null, null is returned.

Examples

Function	Result
`TRIM(' Trim front and back ')`	`'Trim front and back'`
`TRIM (BOTH FROM ' Trim front and back ')`	`'Trim front and back'`
`TRIM (BOTH ' ' FROM ' Trim front and back ')`	`'Trim front and back'`

Function	Result
`TRIM (LEADING 'x' FROM 'xxxTrim frontxxx')`	`'Trim frontxxx'`
`TRIM (TRAILING 'x' FROM 'xxxTrimxBackxxx')`	`'xxxTrimxBack'`
`TRIM (BOTH 'y' FROM 'xxxNo y to trimxxx')`	`'xxxNo y to trimxxx'`

UPPER

```
< UPPER ( <character-expression> )
```

Converts a string to all upper-case characters. Returns null if the input argument is null, and the empty string if the input argument is an empty string.

Examples

Function	Result
UPPER('abcDEFghi123')	ABCDEFGHI123

Date and Time Functions

The following built-in functions relate to dates and time.

Topics

- Date and Time Patterns (p. 107)
- CURRENT_DATE (p. 110)
- CURRENT_ROW_TIMESTAMP (p. 110)
- CURRENT_TIME (p. 110)
- CURRENT_TIMESTAMP (p. 111)
- LOCALTIME (p. 111)
- LOCALTIMESTAMP (p. 111)

Of these, the SQL extension CURRENT_ROW_TIMESTAMP is the most useful for a streaming context, because it gives you information about the times of streaming data as it emerges, not just when the query is run. This is a key difference between a streaming query and a traditional RDMS query: streaming queries remain "open," producing more data, so the timestamp for when the query was run does not offer good information.

LOCALTIMESTAMP, LOCALTIME, CURRENT_DATE, and CURRENT_TIMESTAMP all produce results which are set to values at the time the query first executes. Only CURRENT_ROW_TIMESTAMP generates a row with a unique timestamp (date and time) for each row.

A query run with LOCALTIMESTAMP (or CURRENT_TIMESTAMP or CURRENT_TIME) as one of the columns puts into all output rows the time the query is first run. If that column instead contains CURRENT_ROW_TIMESTAMP, each output row gets a newly-calculated value of TIME representing when that row was output.

Time Zones

Amazon Kinesis Analytics runs in UTC. As a result, all time functions return time in UTC.

Date and Time Patterns

Date and time formats are specified by date and time pattern strings. In these pattern strings, unquoted letters from A to Z and from a to z represent components of a data or time value. If a letter or text string is enclosed within a pair of single quotes, that letter or text is not interpreted but rather used as is, as are all other characters in the pattern string. During printing, that letter or text is copied as is to the output string; during parsing, they are matched against the input string. """ represents a single quote.

The following pattern letters are defined for the indicated Date or Time Component. All other characters from 'A' to 'Z' and from 'a' to 'z' are reserved. For an alphabetic ordering of the pattern letters, see Date and Time Pattern Letters in Alphabetic Order (p. 109).

Date or Time Component	Pattern Letter	Presentation as text or number	Examples
Era designator	G	Text (p. 108)	AD
Year	y	Year	1996; 96
Month in year	M	Month	July; Jul; 07
Week in year	w	Number	27
Week in month	W	Number	2
Day in year	D	Number	189
Day in month	d	Number	10
Day of week in month	F	Number	2
Day in week	E	Text (p. 108)	EE=Tu; EEE=Tue; EEEE=Tuesday
Am/pm marker	a	Text (p. 108)	PM
Hour in day (0-23)	H	Number	0
Hour in day (1-24)	k	Number	24
Hour in am/pm (0-11)	K	Number	0
Hour in am/pm (1-12)	h	Number	12
Minute in hour	m	Number	30
Second in minute	s	Number	55
Millisecond	S	Number	978
Time zone	z	General	Pacific Standard Time; PST; GMT-08:00
Time zone	Z	RFC	-0800

Pattern letters are usually repeated, as their number determines the exact presentation:

Text

For formatting, if the number of pattern letters is 4 or more, the full form is used; otherwise a short or abbreviated form is used if available. For parsing, both forms are accepted, independent of the number of pattern letters.

Number

For formatting, the number of pattern letters is the minimum number of digits, and shorter numbers are zero-padded to this amount. For parsing, the number of pattern letters is ignored unless it's needed to separate two adjacent fields.

Year

Time zones are interpreted as text if they have names. For time zones representing a GMT offset value, the following syntax is used:

```
GMTOffsetTimeZone:
GMT Sign Hours : Minutes
Sign: one of
+ -
Hours:
Digit
Digit Digit
Minutes:
Digit Digit
Digit: one of
0 1 2 3 4 5 6 7 8 9
```

Hours must be between 0 and 23, and Minutes must be between 00 and 59. The format is locale independent and digits must be taken from the Basic Latin block of the Unicode standard.

For parsing, RFC 822 time zones are also accepted.

RFC 822 time zone

For formatting, the RFC 822 4-digit time zone format is used:

```
RFC822TimeZone:
Sign TwoDigitHours Minutes
TwoDigitHours:
Digit Digit
```

TwoDigitHours must be between 00 and 23. Other definitions are as for general time zones.

For parsing, general time zones are also accepted.

SimpleDateFormat also supports "localized date and time pattern" strings. In these strings, the pattern letters described above may be replaced with other, locale dependent, pattern letters. SimpleDateFormat does not deal with the localization of text other than the pattern letters; that's up to the client of the class.

Examples

The following examples show how date and time patterns are interpreted in the U.S. locale. The given date and time are 2001-07-04 12:08:56 local time in the U.S. Pacific time zone.

Date and Time Pattern	Result
"yyyy.MM.dd G 'at' HH:mm:ss z"	2001.07.04 AD at 12:08:56 PDT
"EEE, MMM d, "yy"	Wed, Jul 4, '01
"h:mm a"	12:08 PM
"hh 'o''clock' a, zzzz"	12 o'clock PM, Pacific Daylight Time
"K:mm a, z"	0:08 PM, PDT
"yyyyy.MMMMM.dd GGG hh:mm aaa"	02001.July.04 AD 12:08 PM
"EEE, d MMM yyyy HH:mm:ss Z"	Wed, 4 Jul 2001 12:08:56 -0700
"yyMMddHHmmssZ"	010704120856-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSZ"	2001-07-04T12:08:56.235-0700

Date and Time Pattern Letters in Alphabetic Order

The same pattern letters shown at first, above, in Date or Time Component order are shown below in alphabetic order for easy reference.

Pattern Letter	Date or Time Component	Presentation as text or number	Examples
a	Am/pm marker	Text	PM
D	Day in year	Number	189
d	Day in month	Number	10
E	Day in week	Text	EE=Tu; EEE=Tue; EEEE=Tuesday
F	Day of week in month	Number	2
G	Era designator	Text	AD
H	Hour in day (0-23)	Number	0
h	Hour in am/pm (1-12)	Number	12
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
M	Month in year	Month	July; Jul; 07
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
w	Week in year	Number	27
W	Week in month	Number	2
y	Year	Year	1996; 96

Pattern Letter	Date or Time Component	Presentation as text or number	Examples
z	Time zone	General	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC	-0800

CURRENT_DATE

Returns the current Amazon Kinesis Analytics system date when the query executes as YYYY-MM-DD when the query executes.

For more information, see CURRENT_TIME (p. 110), CURRENT_TIMESTAMP (p. 111), LOCALTIMESTAMP (p. 111), LOCALTIME (p. 111), and CURRENT_ROW_TIMESTAMP (p. 110).

Example

```
+---------------+
| CURRENT_DATE  |
+---------------+
| 2008-08-27    |
+---------------+
```

CURRENT_ROW_TIMESTAMP

CURRENT_ROW_TIMESTAMP is an Amazon Kinesis Analytics extension to the SQL:2008 specification. This function returns the current timestamp as defined by the environment on which the Amazon Kinesis Analytics application is running. CURRENT_ROW_TIMESTAMP is always returned as UTC, not the local timezone.

CURRENT_ROW_TIMESTAMP is similar to LOCALTIMESTAMP (p. 111), but returns a new timestamp for each row in a stream.

A query run with LOCALTIMESTAMP (or CURRENT_TIMESTAMP or CURRENT_TIME) as one of the columns puts into all output rows the time the query is first run.

If that column instead contains CURRENT_ROW_TIMESTAMP, each output row gets a newly-calculated value of TIME representing when that row was output.

> **Note**
> CURRENT_ROW_TIMESTAMP is not defined in the SQL:2008 specification; it is an Amazon Kinesis Analytics extension.

For more information, see CURRENT_TIME (p. 110), CURRENT_DATE (p. 110), CURRENT_TIMESTAMP (p. 111), LOCALTIMESTAMP (p. 111), LOCALTIME (p. 111), and CURRENT_ROW_TIMESTAMP (p. 110).

CURRENT_TIME

Returns the current Amazon Kinesis Analytics system time when the query executes. Time is in UTC, not the local time zone.

For more information, see CURRENT_TIMESTAMP (p. 111), LOCALTIMESTAMP (p. 111), LOCALTIME (p. 111), CURRENT_ROW_TIMESTAMP (p. 110), and CURRENT_DATE (p. 110).

Example

```
+---------------+
| CURRENT_TIME  |
+---------------+
| 20:52:05      |
```

CURRENT_TIMESTAMP

Returns the current database system timestamp (as defined on the environment on which Amazon Kinesis Analytics is running) as a datetime value.

For more information, see CURRENT_TIME (p. 110), CURRENT_DATE (p. 110), LOCALTIME (p. 111), LOCALTIMESTAMP (p. 111), and CURRENT_ROW_TIMESTAMP (p. 110).

Example

```
+--------------------+
| CURRENT_TIMESTAMP  |
+--------------------+
| 20:52:05           |
+--------------------+
```

LOCALTIME

Returns the current time when the query executes as defined by the environment on which Amazon Kinesis Analytics is running. LOCALTIME is always returned as UTC (GMT), not the local timezone.

For more information, see CURRENT_TIME (p. 110), CURRENT_DATE (p. 110), CURRENT_TIMESTAMP (p. 111), LOCALTIMESTAMP (p. 111), and CURRENT_ROW_TIMESTAMP (p. 110).

Example

```
 VALUES localtime;
+------------+
| LOCALTIME  |
+------------+
| 01:11:15   |
+------------+
1 row selected (1.558 seconds)
```

Limitations

Amazon Kinesis Analytics does not support the optional <time precision> parameter specified in SQL:2008. This is a departure from the SQL:2008 standard.

LOCALTIMESTAMP

Returns the current timestamp as defined by the environment on Amazon Kinesis Analytics application is running. Time is always returned as UTC (GMT), not the local timezone.

For more information, see CURRENT_TIME (p. 110), CURRENT_DATE (p. 110), CURRENT_TIMESTAMP (p. 111), LOCALTIME (p. 111), and CURRENT_ROW_TIMESTAMP (p. 110).

Example

```
values localtimestamp;
+-------------------------+
|      LOCALTIMESTAMP     |
+-------------------------+
| 2008-08-27 01:13:42.206  |
+-------------------------+
1 row selected (1.133 seconds)
```

Limitations

Amazon Kinesis Analytics does not support the optional <timestamp precision> parameter specified in SQL:2008. This is a departure from the SQL:2008 standard.

Pattern Matching Functions

Amazon Kinesis Analytics features the following functions for pattern matching:

- REGEX_LOG_PARSE (p. 112) uses the default Java regular expression parser. For more information about this parser, see Pattern in the Java Platform documentation on the Oracle website.
- FAST_REGEX_LOG_PARSER (p. 114) works similarly to the regex parser, but takes several "shortcuts" to ensure faster results. For example, the fast regex parser stops at the first match it finds (known as "lazy" semantics.

REGEX_LOG_PARSE

```
REGEX_LOG_PARSE (<character-expression>,<regex-pattern>,<columns>)<regex-
pattern> := <character-expression>[OBJECT] <columns> := <columnname>
 [ <datatype> ] {, <columnname> <datatype> }*
```

Parses a character string based on Java Regular Expression patterns as defined in java.util.regex.pattern.

Columns are based on match groups defined in the regex-pattern. Each group defines a column, and the groups are processed from left to right. Failure to match produces a NULL value result: If the regular expression does not match the string passed as the first parameter, NULL is returned.

The columns returned will be COLUMN1 through COLUMNn, where n is the number of groups in the regular expression. The columns will be of type varchar(1024).

Example 1

The following code returns two columns with zero or more of [0-9] of the string 'abcde111fghij22klm'

```
SELECT t.r."COLUMN1", t.r."COLUMN2" from
(values (REGEX_LOG_PARSE('abcde111fghij22klm',
 '([^0-9]*)1*([^0-9]*)2*([^0-9]*)'))) t(r);

          +----------+-----------+
          | COLUMN1  | COLUMN2   |
          +----------+-----------+
          | 111      | 22        |
          +----------+-----------+
```

```
          1 row selected
```

Example 2

The following code returns three columns:

- one, labeled "Amount," with one or more digits
- one, labeled "Item," with one or more non-whitespace characters, followed by whitespace, followed by one or more non-whitespace characters
- one, labeled "Ship Date," with one or more non-whitespace characters

```
SELECT t.r."Amount", t.r."Item", t.r."Ship Date" from
(values (REGEX_LOG_PARSE('445 light bulbs should be in the basket. The order
 will ship on 5/4/2014.', '(^(\d+) (\S+\s\S+) should be in the basket. The
 order will ship on (\S+))'))) t(r);

          +----------+-------------+-------------+
          | Amount   |   Item      |  Ship Date  |
          +----------+-------------+-------------+
          | 445      | light bulbs |  5/4/2014   |
          +----------+-------------+-------------+
          1 row selected
```

For more information, see FAST_REGEX_LOG_PARSER (p. 114).

Quick Regex Reference

For full details on Regex, see java.util.regex.pattern

[xyz] Find single character of: x, y or z [^abc] Find any single character except: x, y, or z [r-z] Find any single character between r-z [r-zR-Z] Find any single character between r-z or R-Z ^ Start of line $ End of line \A Start of string \z End of string . Any single character \s Find any whitespace character \S Find any non-whitespace character \d Find any digit \D Find any non-digit	\w Find any word character (letter, number, underscore) \W Find any non-word character \b Find any word boundary (...) Capture everything enclosed (x|y) Find x or y (also works with symbols such as \d or \s) x? Find zero or one of x (also works with symbols such as \d or \s) x* Find zero or more of x (also works with symbols such as \d or \s) x+ Find one or more of x (also works with symbols such as \d or \s) x{3} Find exactly 3 of x (also works with symbols such as \d or \s) x{3,} Find 3 or more of x (also works with symbols such as \d or \s) x{3,6} Find between 3 and 6 of x (also works with symbols such as \d or \s)

FAST_REGEX_LOG_PARSER

```
FAST_REGEX_LOG_PARSE('input_string', 'fast_regex_pattern')
```

The FAST_REGEX_LOG_PARSE works by first decomposing the regular expression into a series of regular expressions, one for each expression inside a group and one for each expression outside a group. Any fixed length portions at the start of any expressions are moved to the end of the previous expression. If any expression is entirely fixed length, it is merged with the previous expression.
The series of expressions is then evaluated using lazy semantics with no backtracking. (In regular expression parsing parlance, "lazy" means don't parse more than you need to at each step. "Greedy" means parse as much as you can at each step.)

The columns returned will be COLUMN1 through COLUMNn, where n is the number of groups in the regular expression. The columns will be of type varchar(1024). See sample usage below at First FRLP Example and at Further FRLP Examples.

FAST_REGEX_LOG_PARSER (FRLP)

FAST_REGEX_LOG_PARSER uses a lazy search - it stops at the first match. By contrast, the REGEX_LOG_PARSE (p. 112) is greedy unless possessive quantifiers are used.

FAST_REGEX_LOG_PARSE scans the supplied input string for all the characters specified by the Fast Regex pattern.

- All characters in that input string must be accounted for by the characters and scan groups defined in the Fast Regex pattern. Scan groups define the fields-or-columns resulting when a scan is successful.
- If all characters in the input_string are accounted for when the Fast Regex pattern is applied, then FRLP creates an output field (column) from each parenthetical expression in that Fast Regex pattern, in left-to-right order. The first (leftmost) parenthetical expression creates the first output field, the next (second) parenthetical expression creates the second output field, up through the last parenthetical expression creating the last output field.
- If the input_string contains any characters not accounted for (matched) by applying Fast Regex pattern, then FRLP returns no fields at all.

Character Class Symbols for Fast Regex

Fast Regex uses a different set of character class symbols from the regular regex parser:

Symbol or Construct	Meaning
-	Character range, including endpoints
[charclasses]	Character class
[^ charclasses]	Negated character class
|	Union
&	Intersection
?	Zero or one occurrence
*	Zero or more occurrences
+	One or more occurrences
{n}	n occurrences

Symbol or Construct	Meaning
{n,}	n or more occurrences
{n,m}	n to m occurrences, including both
.	Any single character
#	The empty language
@	Any string
"<Unicode string without double-quotes>"	A string)
( )	The empty string)
(unionexp)	Precedence override
< <identifier> >	Named pattern
<n-m>	Numerical interval
charexp:=<Unicode character>	A single non-reserved character
\ <Unicode character>	A single character)

We support the following POSIX standard identifiers as named patterns:

<Digit> - "[0-9]"

<Upper> - "[A-Z]"

<Lower> - "[a-z]"

<ASCII> - "[\u0000-\u007F]"

<Alpha> - "<Lower>|<Upper>"

<Alnum> - "<Alpha>|<Digit>"

<Punct> - "[!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~]"

<Blank> - "[\t]"

<Space> - "[\t\n\f\r\u000B]"

<Cntrl> - "[\u0000-\u001F\u007F]"

<XDigit> - "0-9a-fA-F"

<Print> - "<Alnum>|<Punct>"

<Graph> - "<Print>"

First FRLP Example

This first example uses the Fast Regex pattern '(.*)_(._.*)_.*'

```
select t.r."COLUMN1", t.r."COLUMN2" from
. . . . . .> (values
 (FAST_REGEX_LOG_PARSE('Mary_had_a_little_lamb', '(.*)_(._.*)_.*'))) t(r);
```

```
+-----------------------+-----------------------+
|         COLUMN1       |         COLUMN2       |
+-----------------------+-----------------------+
| Mary_had              |     a_little_lamb     |
+-----------------------+-----------------------+
1 row selected
```

1. The scan of input_string ('Mary_had_a_little_lamb') begins with the 1st group defined in Fast Regex pattern: (.*), which means "find any character 0 or more times."

   '**(.*)**_(._.*)_.*'
2. This group specification, defining the first column to be parsed, asks the Fast Regex Log Parser to accept input string characters starting from the input string's first character until it finds the next group in the Fast Regex Pattern or the next literal character or string that is not inside a group (not in parentheses). In this example, the next literal character after the first group is an underscore:

   '(.*)_(._.*)_.*'
3. The parser scans each character in the input string until it finds the next specification in the Fast Regex pattern: an underscore:

   '(.*)**_(._.*)**_.*'
4. Group-2 thus begins with "a_l". Next, the parser needs to determine the end of this group, using the remaining specification in the pattern:

   '(.*)_(._.*)**_.***'

Note
Character-strings or literals specified in the pattern but not inside a group must be found in the input string but will not be included in any output field.
If the Fast Regex pattern had omitted the final asterisk, no results would be obtained.

Further FRLP Examples

The next example uses a "+", which means repeat the last expression 1 or more times ("*" means 0 or more times).

Example A

In this case, the longest prefix is the first underscore. The first field/column group will match on "Mary" and the second will not match.

```
select t.r."COLUMN1", t.r."COLUMN2" from
      . . . . . . . . . . . . .> (values
 (FAST_REGEX_LOG_PARSE('Mary_had_a_little_lamb',
        '(.*)_+(._.*)'))) t(r);
     +----------+----------+
     | COLUMN1  | COLUMN2  |
     +----------+----------+
     +----------+----------+
     No rows selected
```

The preceding example returns no fields because the "+" required there be at least one more underscore-in-a-row; and the input_string does not have that.

Example B

In the following case, the '+' is superfluous because of the lazy semantics:

```
select t.r."COLUMN1", t.r."COLUMN2" from
      . . . . . . . . . . . . .> (values
 (FAST_REGEX_LOG_PARSE('Mary____had_a_little_lamb',
        '(.*)_+(.*)'))) t(r);
     +------------------------+------------------------+
     |         COLUMN1        |         COLUMN2        |
     +------------------------+------------------------+
     | Mary                   |    had_a_little_lamb   |
     +------------------------+------------------------+
     1 row selected
```

The preceding example succeeds in returning two fields because after finding the multiple underscores required by the "_+" specification, the group-2 specification (.*) accepts all remaining characters in the .input_string. Underscores do not appear trailing "Mary" nor leading "had" because the "_+" specification is not enclosed in parentheses.

As mentioned in the introduction, "lazy" in regular expression parsing parlance means don't parse more than you need to at each step; "Greedy" means parse as much as you can at each step.

The first case in this topic, A, fails because when it gets to the first underscore, the regex processor has no way of knowing without backtracking that it can't use the underscore to match "_+", and FRLP doesn't backtrack, whereas REGEX_LOG_PARSE (p. 112) does.

The search directly above, B, gets turned into three searches:

```
(.*)_
_*(._
.*)
```

Notice that the second field group gets split between the second and third searches, also that "_+" is considered the same as "__*" (that is, it considers "underscore repeat-underscore-1-or-more-times" the same as "underscore underscore repeat-underscore-0-or-more-times".)

Case A demonstrates the main difference between REGEX_LOG_PARSE and FAST_REGEX_LOG_PARSE, because the search in A would work under REGEX_LOG_PARSE because that function would use backtracking.

Example C

In the following example, the plus is not superfluous, because the "<Alpha> (any alphabetic char) is fixed length thus will be used as a delimiter for the " +" search.

```
select t.r."COLUMN1", t.r."COLUMN2" from
. . . . . . . . . . . . .> (values
 (FAST_REGEX_LOG_PARSE('Mary____had_a_little_lamb', '(.*)_+(<Alpha>.*)')))
 t(r);
+---------------------------+---------------------------+
|          COLUMN1          |          COLUMN2          |
+---------------------------+---------------------------+
| Mary                      | had_a_little_lamb         |
+---------------------------+---------------------------+
1 row selected

'(.*) +(<Alpha>.*)' gets converted into three regular expressions:
'.* '
' *<Alpha>'
'.*$'
```

Each is matched in turn using lazy semantics.

The columns returned will be COLUMN1 through COLUMNn, where n is the number of groups in the regular expression. The columns will be of type varchar(1024).

Expressions and Literals

Value expressions

Value expressions are defined by the following syntax:

```
value-expression := <character-expression > | <number-expression> |
 <datetime-expression> | <interval-expression> | <boolean-expression>
```

Character (string) expressions

Character expressions are defined by the following syntax:

```
character-expression := <character-literal>
                  | <character-expression> || <character-expression>
                  | <character-function> ( <parameters> )

 character-literal  := <quote> { <character> }* <quote>
 string-literal     := <quote> { <character> }* <quote>
 character-function   :=   CAST | COALESCE | CURRENT_PATH
                  | FIRST_VALUE  | INITCAP | LAST_VALUE
                  | LOWER | MAX | MIN | NULLIF
                  | OVERLAY | SUBSTRING| SYSTEM_USER
                  | TRIM | UPPER
                  | <user-defined-function>
```

Note that Amazon Kinesis Analytics streaming SQL supports unicode character literals, such as u&'foo'. As in the use of regular literals, you can escape single quotes in these, such as u&'can''t'. Unlike regular literals, you can have unicode escapes: e.g., u&'\0009' is a string consisting only of a tab character. You can escape a \ with another \, such as u&'back\\slash'. Amazon Kinesis Analytics also supports alternate escape characters, such as u&'!0009!!' uescape '!' is a tab character.

Numeric expressions

Numeric expressions are defined by the following syntax:

```
number-expression := <number-literal>
                  | <number-unary-oper> <number-expression>
                  | <number-expression> <number-operator> <number-expression>
                  | <number-function> [ ( <parameters> ) ]
 number-literal :=   <UNSIGNED_INTEGER_LITERAL> | <DECIMAL_NUMERIC_LITERAL>
                        | <APPROX_NUMERIC_LITERAL>
```

```
--Note: An <APPROX_NUMERIC_LITERAL> is a number in scientific notation, such
 as with an
--exponent, such as 1e2 or -1.5E-6.
number-unary-oper := + | -
```

```
 number-operator      :=   + | - | / | *

 number-function      :=   ABS | AVG | CAST | CEIL
                         | CEILING | CHAR_LENGTH
                         | CHARACTER_LENGTH | COALESCE
                         | COUNT | EXP | EXTRACT
                         | FIRST_VALUE
                         | FLOOR | LAST_VALUE
                         | LN | LOG10
                         | MAX | MIN  | MOD
                         | NULLIF
                         | POSITION | POWER
                         | SUM| <user-defined-function>
```

Date / Time expressions

Date / Time expressions are defined by the following syntax:

```
datetime-expression := <datetime-literal>
                         | <datetime-expression> [ + | - ] <number-
expression>
                         | <datetime-function> [ ( <parameters> ) ]
 datetime-literal    :=  <left_brace> { <character-literal> } *
  <right_brace>
                         |  <DATE> { <character-literal> } *
                         |  <TIME> { <character-literal> } *
                         |  <TIMESTAMP> { <character-literal> } *
 datetime-function   :=    CAST | CEIL | CEILING
                         | CURRENT_DATE | CURRENT_ROW_TIMESTAMP
                         | CURRENT_ROW_TIMESTAMP
                         | FIRST_VALUE| FLOOR
                         | LAST_VALUE | LOCALTIME
                         | LOCALTIMESTAMP | MAX | MIN
                         | NULLIF | ROWTIME
                         | <user-defined-function>
 <time unit>         :=  YEAR | MONTH | DAY | HOUR | MINUTE | SECOND
```

Interval Expression

Interval expressions are defined by the following syntax:

```
interval-expression := <interval-literal>
                         | <interval-function>
 interval-literal    :=    <INTERVAL> ( <MINUS> | <PLUS> ) <QUOTED_STRING>
 <IntervalQualifier>
 IntervalQualifier  :=  <YEAR> ( <UNSIGNED_INTEGER_LITERAL> )
                         | <YEAR> ( <UNSIGNED_INTEGER_LITERAL> ) <TO>
 <MONTH>
                         |  <MONTH> [ ( <UNSIGNED_INTEGER_LITERAL> ) ]
                         |  <DAY> [ (  <UNSIGNED_INTEGER_LITERAL> )  ]
                         |  <DAY> [ (  <UNSIGNED_INTEGER_LITERAL> )  ] <TO>
                                  { <HOUR> | <MINUTE> | <SECOND>
 [ ( <UNSIGNED_INTEGER_LITERAL> ) ] }
                         |  <HOUR> [ (  <UNSIGNED_INTEGER_LITERAL> ) ]
                         |  <HOUR> [ (  <UNSIGNED_INTEGER_LITERAL> ) ] <TO>
                                  { <MINUTE> | <SECOND>
 [ <UNSIGNED_INTEGER_LITERAL> ] }
                         |  <MINUTE> [ ( <UNSIGNED_INTEGER_LITERAL> )  ]
```

```
                          |  <MINUTE> [ ( <UNSIGNED_INTEGER_LITERAL> )  ]
<TO>
                                      <SECOND>
[ ( <UNSIGNED_INTEGER_LITERAL> ) ]
                          |  <SECOND> [ ( <UNSIGNED_INTEGER_LITERAL> )  ]
interval-function   :=    ABS  | CAST | FIRST_VALUE
                          | LAST_VALUE | MAX | MIN
                          | NULLIF| <user-defined-function>
```

Boolean expression

Boolean expressions are defined by the following syntax:

```
boolean-expression := <boolean-literal>
                   | <boolean-expression> <boolean-operator> <boolean-
expression>
                   | <boolean-unary-oper> <boolean-expression>
                         | <boolean-function> ( <parameters> )
                         | ( <boolean-expression> )
 boolean-literal    :=  TRUE | FALSE
 boolean-operator   :=  AND | OR
 boolean-unary-oper :=  NOT
 boolean-function   :=    CAST | FIRST_VALUE | LAST_VALUE
                        | NULLIF | <user-defined-function>
```

Monotonic Expressions and Operators

Since Amazon Kinesis Analytics queries operate on infinite streams of rows, some operations are only possible if something is known about those streams.

For example, given a stream of orders, it makes sense to ask for a stream summarizing orders by day and product (because day is increasing) but not to ask for a stream summarizing orders by product and shipping state. We can never complete the summary of, say Widget X to Oregon, because we never see the 'last' order of a Widget to Oregon.

This property, of a stream being sorted by a particular column or expression, is called monotonicity.

Some time-related definitions:

- **Monotonic**. An expression is monotonic if it is ascending or descending. An equivalent phrasing is "non-decreasing or non-increasing."
- **Ascending**. An expression e is ascending within a stream if the value of e for a given row is always greater than or equal to the value in the previous row.
- **Descending**. An expression e is descending within a stream if the value of e for a given row is always less than or equal to the value in the previous row.
- **Strictly Ascending**. An expression e is strictly ascending within a stream if for the value of e for a given row is always greater than the value in the previous row.
- **Strictly Descending**. An expression e is strictly descending within a stream if the value of e for a given row is always less than the value in the previous row.
- **Constant**. An expression e is constant within a stream if the value of e for a given row is always equal to the value in the previous row.

Note that by this definition, a constant expression is considered monotonic.

Monotonic columns

The ROWTIME system column is ascending. The ROWTIME column is not strictly ascending: it is acceptable for consecutive rows to have the same timestamp.

Amazon Kinesis Analytics prevents a client from inserting a row into a stream whose timestamp is less than the previous row it wrote into the stream. Amazon Kinesis Analytics also ensures that if multiple clients are inserting rows into the same stream, the rows are merged so that the ROWTIME column is ascending.

Clearly it would be useful to assert, for instance, that the orderId column is ascending; or that no orderId is ever more than 100 rows from sorted order. However, declared sort keys are not supported in the current release.

Monotonic expressions

Amazon Kinesis Analytics can deduce that an expression is monotonic if it knows that its arguments are monotonic. (See also the Monotonic Function (p. 44).)

Another definition:

Functions or operators that are monotonic

A function or operator is monotonic if, when applied to a strictly increasing sequence of values, it yields a monotonic sequence of results.

For example, the FLOOR function, when applied to the ascending inputs {1.5, 3, 5, 5.8, 6.3}, yields {1, 3, 5, 5, 6}. Note that the input is strictly ascending, but the output is merely ascending (includes duplicate values).

Rules for deducing monotonicity

Amazon Kinesis Analytics requires that one or more grouping expressions are valid in order for a streaming GROUP BY statement to be valid. In other cases, Amazon Kinesis Analytics may be able to operate more efficiently if it knows about monotonicity; for example it may be able to remove entries from a table of windowed aggregate totals if it knows that a particular key will never be seen on the stream again.

In order to exploit monotonicity in this way, Amazon Kinesis Analytics uses a set of rules for deducing the monotonicity of an expression. Here are the rules for deducing monotonicity:

Expression	Monotonicity
c	Constant
FLOOR (p. 99)(m)	Same as m, but not strict
CEIL / CEILING (p. 95)(m)	Same as m, but not strict
CEIL / CEILING (p. 95)(m TO timeUnit)	Same as m, but not strict
FLOOR (p. 99)(m TO timeUnit)	Same as m, but not strict
SUBSTRING (p. 104)(m FROM 0 FOR c)	Same as m, but not strict
+ m	Same as m
- m	Reverse of m
m + c c + m	Same as m

Expression	Monotonicity
m1 + m2	Same as m1, if m1 and m2 have same direction; otherwise not monotonic
c - m	Reverse of m
m * c c * m	Same as m if c is positive; reverse of m is c is negative; constant (0) c is 0
c / m	Same as m if m is always positive or always negative, and c and m have same sign; reverse of m if m is always positive or always negative, and c and m have different sign; otherwise not monotonic
	Constant
LOCALTIME (p. 111) LOCALTIMESTAMP (p. 111) CURRENT_ROW_TIMESTAMP (p. 110) CURRENT_DATE (p. 110)	Ascending

Throughout the table, c is a constant, and m (also m1 and m2) is a monotonic expression.

Condition Clause

Referenced by:

- SELECT clauses: HAVING clause (p. 147), WHERE clause (p. 149), and JOIN clause (p. 142). (See also the SELECT chart and its SELECT clause (p. 137).)
- DELETE

A condition is any value expression of type BOOLEAN, such as the following examples:

- 2<4
- TRUE
- FALSE
- expr_17 IS NULL
- NOT expr_19 IS NULL AND expr_23 < expr>29
- expr_17 IS NULL OR (NOT expr_19 IS NULL AND expr_23 < expr>29)

Standard SQL Operators

The following topics discuss standard SQL operators:

Topics

CREATE statements

You can use the following CREATE statements with Amazon Kinesis Analytics:

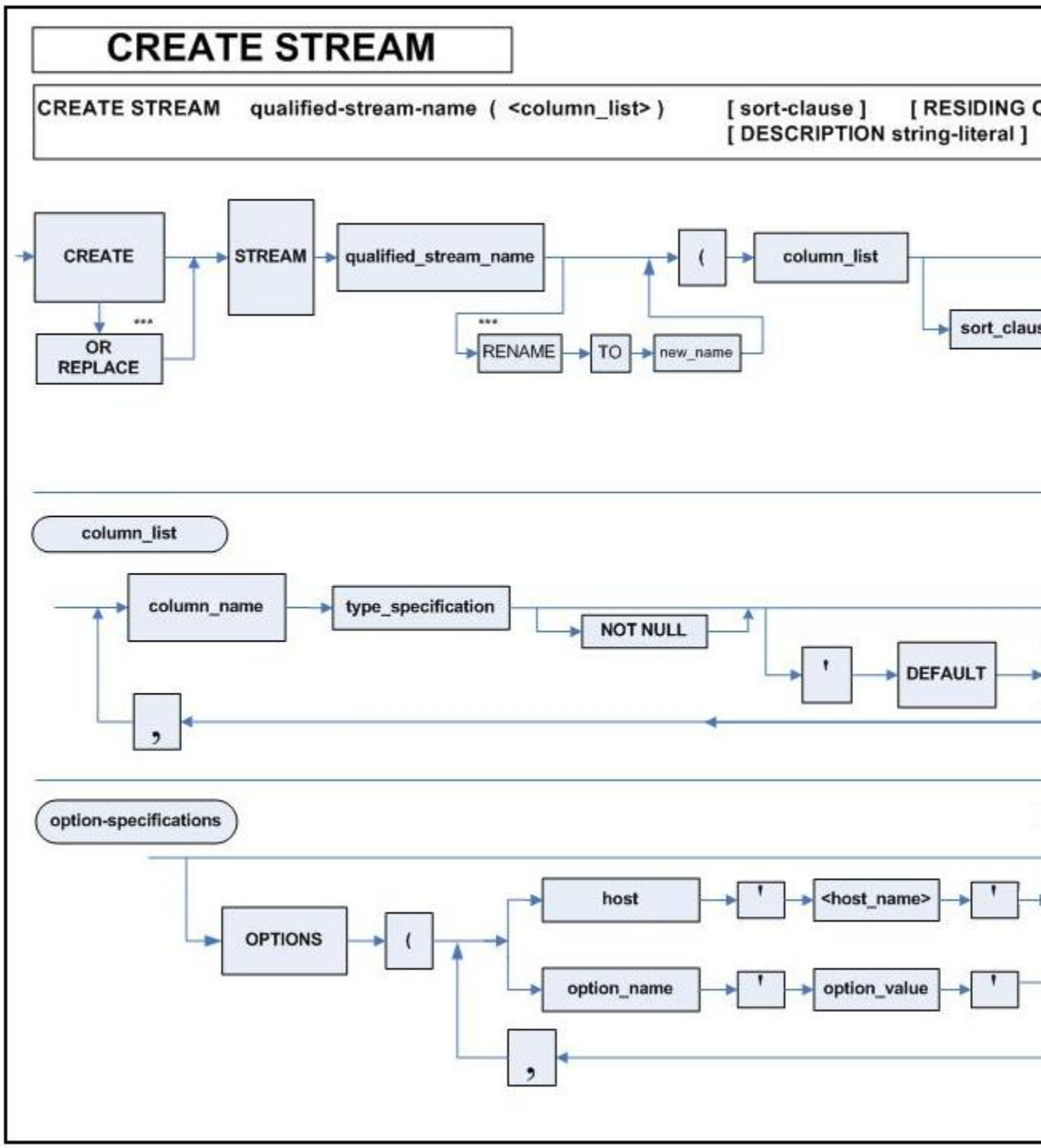
CREATE STREAM

The CREATE STREAM statement creates a (local) stream. The name of the stream must be distinct from the name of any other stream in the same schema. It is good practice to include a description of the stream.

Like tables, streams have columns, and you specify the data types for these in the CREATE STREAM statement. These should map to the data source for which you are creating the stream. For column_name, any valid non-reserved SQL name is usable. Column values cannot be null.

- Specifying OR REPLACE re-creates the stream if it already exists, enabling a definition change for an existing object, implicitly dropping it without first needing to use a DRP command. Using CREATE OR REPLACE on a stream that already has data in flight kills the stream and loses all history.
- RENAME can be specified only if OR REPLACE has been specified.
- For the complete list of types and values in type_specification, such as TIMESTAMP, INTEGER, or varchar(2), see the topic Amazon Kinesis Analytics Data Types in the Amazon Kinesis Analytics SQL Reference Guide.
- For option_value, any string can be used.

Syntax



The following are basic examples of streams defined for simple data sources. Note: All streams need to be defined within a schema.

Simple stream for unparsed log data

```
CREATE OR REPLACE STREAM logStream (
    source  VARCHAR(20),
    message VARCHAR(3072))
DESCRIPTION 'Head of webwatcher stream processing';
```

Stream capturing sensor data from Intelligent Travel System pipeline

```
CREATE OR REPLACE STREAM "LaneData" (
    -- ROWTIME is time at which sensor data collected
    LDS_ID  INTEGER,        -- loop-detector ID
    LNAME   VARCHAR(12),
    LNUM    VARCHAR(4),
    OCC     SMALLINT,
    VOL     SMALLINT,
    SPEED   DECIMAL(4,2)
) DESCRIPTION 'Conditioned LaneData for analysis queries';
```

Stream capturing order data from e-commerce pipeline

```
CREATE OR REPLACE STREAM "OrderData" (
    "key_order"    BIGINT NOT NULL,
    "key_user"     BIGINT,
    "country"      SMALLINT,
    "key_product"  INTEGER,
    "quantity"     SMALLINT,
    "eur"          DECIMAL(19,5),
    "usd"          DECIMAL(19,5)
) DESCRIPTION 'conditioned order data, ready for analysis';
```

CREATE FUNCTION

Amazon Kinesis Analytics provides a number of Standard Functions (p. 30), and also allows users to extend its capabilities by means of user-defined functions (UDFs). Amazon Kinesis Analytics supports UDFs defined in SQL only.

User-defined functions may be invoked using either the fully-qualified name or by the function name alone.

Values passed to (or returned from) a user-defined function or transformation must be exactly the same data types as the corresponding parameter definitions. In other words, implicit casting is not allowed in passing parameters to (or returning values from) a user-defined function.

User-Defined Function (UDF)

A user-defined function can implement complex calculations, taking zero or more scalar parameters and returning a scalar result. UDFs operate like built-in functions such as FLOOR() or LOWER(). For each occurrence of a user-defined function within a SQL statement, that UDF is called once per row with scalar parameters: constants or column values in that row.

Syntax

```
 CREATE FUNCTION ''<function_name>'' ( ''<parameter_list>'' )
```

```
  RETURNS ''<data type>''
  LANGUAGE SQL
  [ SPECIFIC ''<specific_function_name>''  | [NOT] DETERMINISTIC ]
  CONTAINS SQL
  [ READS SQL DATA ]
  [ MODIFIES SQL DATA ]
  [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
  RETURN ''<SQL-defined function body>''
```

SPECIFIC assigns a specific function name that is unique within the application. Note that the regular function name does not need to be unique (two or more functions may share the same name, as long as they are distinguishable by their parameter list).

DETERMINISTIC / NOT DETERMINISTIC indicates whether a function will always return the same result for a given set of parameter values. This may be used by your application for query optimization.

READS SQL DATA and MODIFIES SQL DATA indicate whether the function potentially reads or modifies SQL data, respectively. If a function attempts to read data from tables or streams without READS SQL DATA being specified, or insert to a stream or modify a table without MODIFIES SQL DATA being specified, an exception will be raised.

RETURNS NULL ON NULL INPUT and CALLED ON NULL INPUT indicate whether the function is defined as returning null if any of its parameters are null. If left unspecified, the default is CALLED ON NULL INPUT.

A SQL-defined function body consists only of a single RETURN statement.

Examples

```
CREATE FUNCTION get_fraction( degrees DOUBLE )
    RETURNS DOUBLE
    CONTAINS SQL
    RETURN degrees - FLOOR(degrees)
;
```

CREATE PUMP

A pump is an Amazon Kinesis Analytics Repository Object (an extension of the SQL standard) that provides a continuously running INSERT INTO stream SELECT ... FROM query functionality, thereby enabling the results of a query to be continuously entered into a named stream.

You need to specify a column list for both the query and the named stream (these imply a set of source-target pairs). The column lists need to match in terms of datatype, or the SQL validator will reject them. (These need not list all columns in the target stream; you can set up a pump for one column.)

For more information, see SELECT statement (p. 134).

The following code first creates and sets a schema, then creates two streams in this schema:

- "OrderDataWithCreateTime" which will serve as the origin stream for the pump.
- "OrderData" which will serve as the destination stream for the pump.

```
CREATE SCHEMA "Test";
```

```
SET SCHEMA '"Test"';

CREATE OR REPLACE STREAM "OrderDataWithCreateTime" (
"key_order" VARCHAR(20),
"key_user" VARCHAR(20),
"key_billing_country" VARCHAR(20),
"key_product" VARCHAR(20),
"quantity" VARCHAR(20),
"eur" VARCHAR(20),
"usd" VARCHAR(20))
DESCRIPTION 'Creates origin stream for pump';

CREATE OR REPLACE STREAM "OrderData" (
"key_order" VARCHAR(20),
"key_user" VARCHAR(20),
"country" VARCHAR(20),
"key_product" VARCHAR(20),
"quantity" VARCHAR(20),
"eur" INTEGER,
"usd" INTEGER)
DESCRIPTION 'Creates destination stream for pump';
```

The following code uses these two streams to create a pump. Data is selected from "OrderDataWithCreateTime" and inserted into "OrderData".

```
CREATE OR REPLACE PUMP "200-ConditionedOrdersPump" STOPPED AS
INSERT INTO "OrderData" (
"key_order", "key_user", "country",
"key_product", "quantity", "eur", "usd")
//note that this list matches that of the query
SELECT STREAM
"key_order", "key_user", "key_billing_country",
"key_product", "quantity", "eur", "usd"
//note that this list matches that of the insert statement
FROM "OrderDataWithCreateTime";
```

For more detail, see the topic In-Application Streams and Pumps in the *Amazon Kinesis Analytics Developer Guide*.

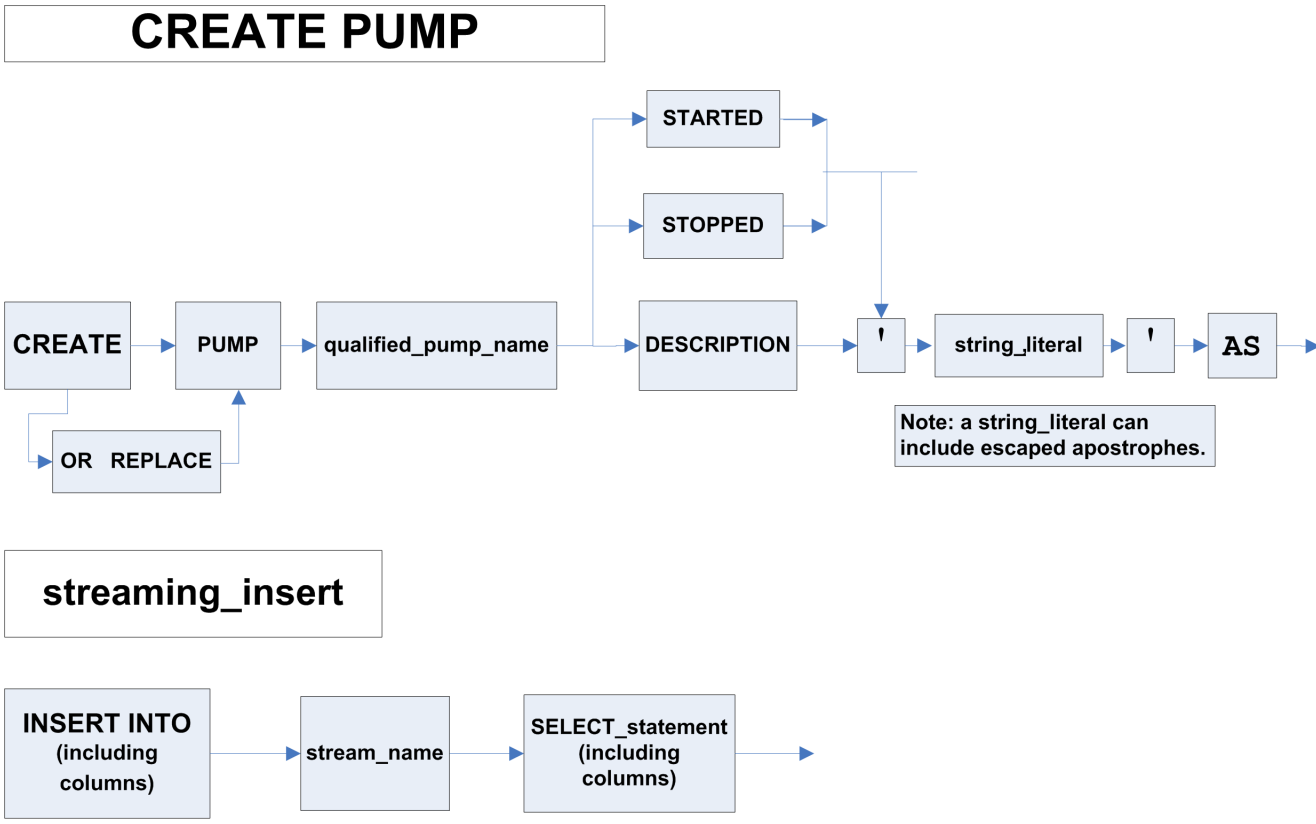
Syntax

```
 CREATE [ OR REPLACE ] PUMP <qualified-pump-name> [ STARTED | STOPPED ]
                       [ DESCRIPTION '<string-literal>' ] AS <streaming-
insert>
```

where streaming-insert is an insert statement such as:

```
INSERT INTO ''stream-name'' SELECT "columns" FROM <source stream>
```

Syntax Chart



INSERT

INSERT is used to insert rows into a stream. It can also be used in a pump to insert the output of one stream into another.

Syntax

```
<insert statement> :=
  INSERT [ EXPEDITED ]
  INTO  <table-name > [ ( insert-column-specification ) ]
  <  query  >
<insert-column-specification> := < simple-identifier-list >
<simple-identifier-list> :=
   <simple-identifier> [ , < simple-identifier-list > ]
```

For a discussion of VALUES, see SELECT statement (p. 134).

Pump Stream Insert

INSERT may also be specified as part of a CREATE PUMP (p. 126) statement.

```
CREATE PUMP "HighBidsPump" AS INSERT INTO "highBids" ( "ticker", "shares",
"price")
SELECT  "ticker", "shares", "price"
```

```
FROM SALES.bids
WHERE "shares"*"price">100000.00
```

Here the results to be inserted into the "highBids" stream should come from a UNION ALL expression that evaluates to a stream. This will create a continuously running stream insert. Rowtimes of the rows inserted will be inherited from the rowtimes of the rows output from the select or UNION ALL. Again rows may be initially dropped if other inserters, ahead of this inserter, have inserted rows with rowtimes later than those initially prepared by this inserter, since the latter would then be out of time order. See the topic CREATE PUMP (p. 126) in this guide.

MERGE statements

The MERGE statement modifies an UPDATE, along with a JOIN clause (p. 142) condition. The JOIN condition defines how each source row matches one target row. When there is a match, the matching target row is changed according to the UPDATE part of the merge. When there is no match, the INSERT part of the merge is used to add a new row to the target table.

It is allowed for several source rows to match the same target row. This differs from the SQL:2008 standard, but makes more sense when the source is a stream. If the source stream contains several rows with the same values in the ON condition, they will match the same target row, and as each source row arrives it will update the target row with new values.

(There is no way to delete a target row. Amazon Kinesis Analytics application implements the standard SQL:2008 MERGE statement, except for the DELETE clause).

The merge runs til the end of data of its source, so if the source is a streaming query, the merge will run forever. A merge like this is generally run as a pump.

> **Note**
> To avoid a bug in the query planner, you must refer to a native table using a loopback alias. This makes a native table appear to be an external table, using the SQL/MED plugin, but pointing back to same Amazon Kinesis Analytics application instance. (Hence the name "loopback").

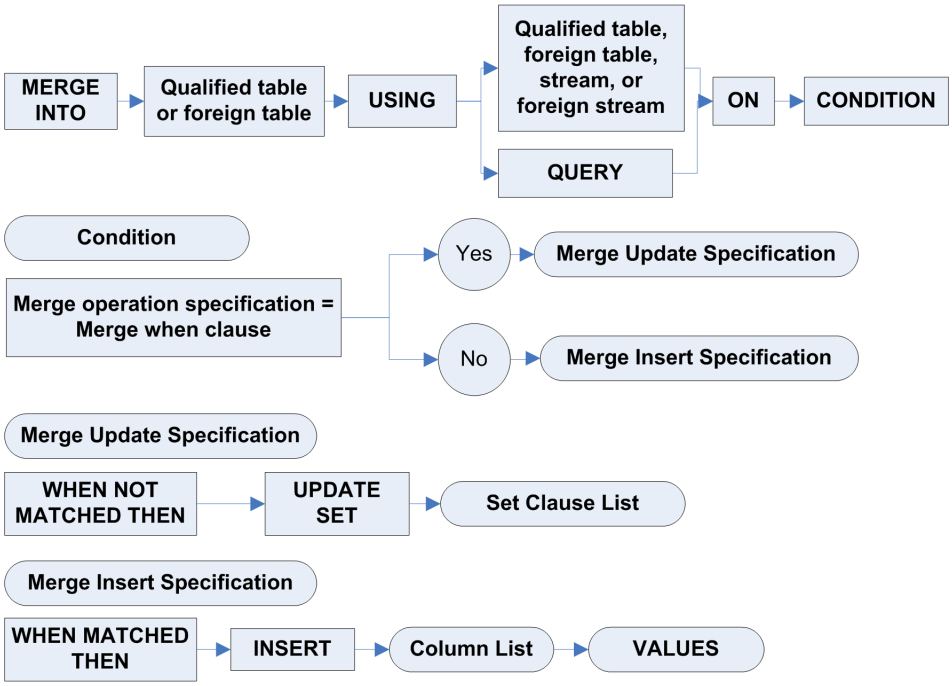
MERGE and Rowtime

In a stream, ROWTIME is a special column: every row has a rowtime. In a table however, there is no special rowtime column. A table may contain an ordinary column named ROWTIME of type TIMESTAMP, or it may not, according to its definition (see CREATE TABLE). A merge statement can save a value as the ROWTIME of a target row, but it must do so explicitly in the UPDATE and INSERT clauses.

Syntax

```
MERGE INTO <target table> USING <source query> ON <conditions>
WHEN MATCHED THEN <merge update>
WHEN NOT MATCHED THEN <merge insert>;
where:
<conditions> = a boolean expression in columns from the target and the source
 row
<merge update> = UPDATE SET <column1> = <expr1>, ... <columnN> = <exprN>
<merge insert> = INSERT (<column1>,...<columnN>) VALUES (<expr1>,...<exprN>)
and <expr> is any scalar expression based on columns from the target and the
 source row.
```

Diagram



Semantics

INTO

The INTO clause indicates the target table or foreign table to be updated.

USING

The USING clause indicates the source of the data to be updated or inserted. The source can be a table, foreign table, or the result of a subquery.

ON

The ON clause indicates the condition for the MERGE statement updating or inserting rows. When the search condition is met, Amazon Kinesis Analytics application updates the target table row with corresponding data from the source. If the condition is not true for any rows, then Amazon Kinesis Analytics application inserts into target table based on the corresponding source row.

WHEN MATCHED THEN

The WHEN MATCHED THEN clause indicates new column values for the target table. Amazon Kinesis Analytics application updates columns if the ON clause is true.

WHEN NOT MATCHED THEN

The WHEN NOT MATCHED THEN clause indicates columns and values for rows to be inserted if the ON clause is false.

Example

The following example creates a pump implementing a MERGE on a loopback table called "LOOPBACK"."MOCHI_VIZ"."CityAttackTotals" using a query. The merge condition asks if the columns

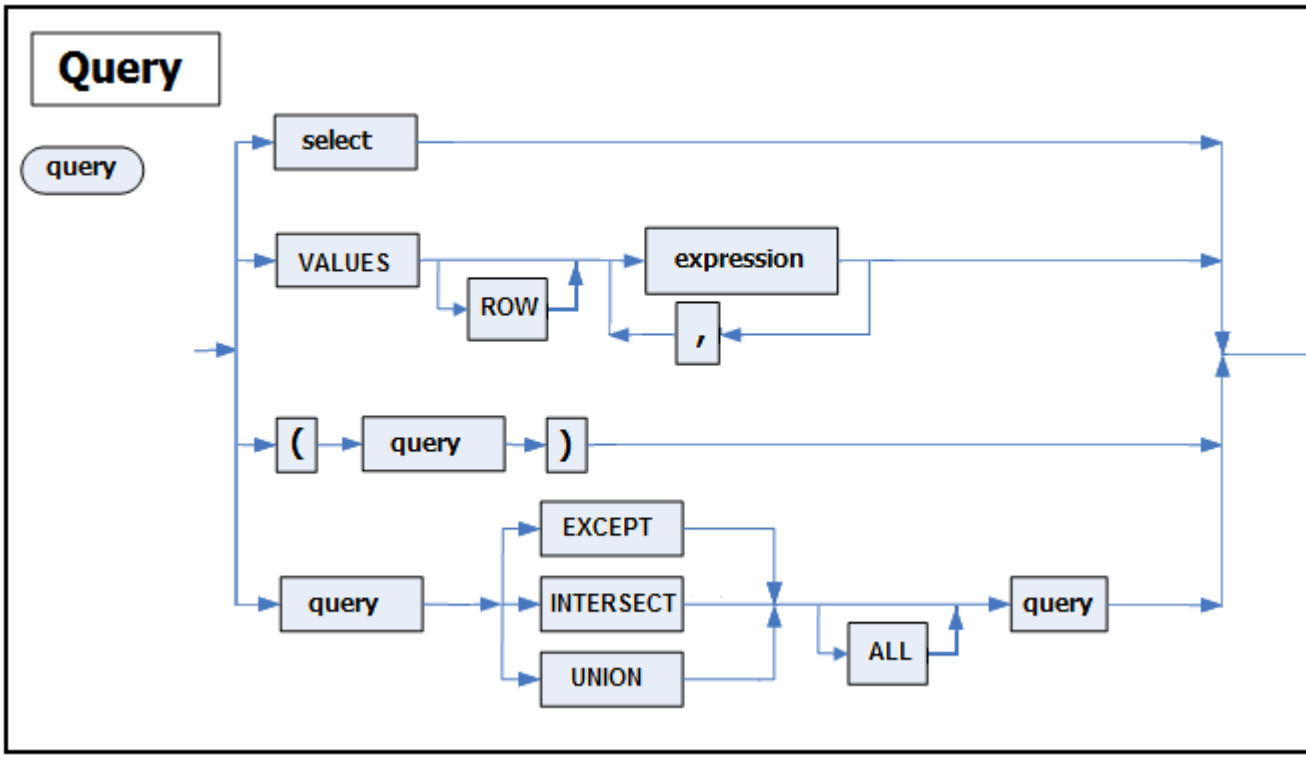
state and city match between the target table and the query. When the merge condition is met, the columns state, city, value, and lastmod are updated. When the merge condition is not met, new rows are inserted into the table.

```
CREATE OR REPLACE PUMP "100-persistCityAttacks" STOPPED
  DESCRIPTION 'persist city attack totals to local/loopback table'
  AS
  MERGE INTO "LOOPBACK"."MOCHI_VIZ"."CityAttackTotals" AS "CAT"
    --query:
    USING
      (SELECT STREAM
       COALESCE("region", '**') AS "state",
       "city", "totalFails", CURRENT_ROW_TIMESTAMP AS "lastmod"
      FROM "MOCHI"."SuspectLoginLocations") AS "SLL"
  --merge condition:
  ON ("CAT"."state" = "SLL"."state") AND ("CAT"."city" = "SLL"."city")
  --merge update specification:
  WHEN MATCHED THEN
    UPDATE SET
      "state" = "SLL"."state",
      "city" = "SLL"."city",
      "value" = "SLL"."totalFails",
      "lastmod" = "SLL"."lastmod"
  --merge insert specification:
  WHEN NOT MATCHED THEN
      INSERT ("state", "city", "value", "lastmod")
        VALUES ("SLL"."state", "SLL"."city", "SLL"."totalFails",
 "SLL"."lastmod");
```

Query

Syntax

```
 <query> :=
     <select>
   | <query> <set-operator> [ ALL ] <query>
   | VALUES <row-constructor> { , <row-constructor> }...
   | '(' <query> ')'
 <set-operator> :=
     EXCEPT
   | INTERSECT
   | UNION
 <row-constructor> :=
     [ ROW ] ( <expression> { , <expression> }... )
```



select

The select box in the chart above represents any SELECT command; that command is described in detail on its own page.

Set operators (EXCEPT, INTERSECT, UNION)

Set operators combine rows produced by queries using set operations:

- EXCEPT returns all rows that are in the first set but not in the second
- INTERSECT returns all rows that are in both first and second sets
- UNION returns all rows that are in either set

In all cases, the two sets must have the same number of columns, and the column types must be assignment-compatible. The column names of the resulting relation are the names of the columns of the first query.

With the ALL keyword, the operators use the semantics of a mathematical multiset, meaning that duplicate rows are not eliminated. For example, if a particular row occurs 5 times in the first set and 2 times in the second set, then UNION ALL will emit the row 3 + 2 = 5 times.

ALL is not currently supported for EXCEPT or INTERSECT.

All operators are left-associative, and INTERSECT has higher precedence than EXCEPT or UNION, which have the same precedence. To override default precedence, you can use parentheses. For example:

```
SELECT * FROM a
UNION
SELECT * FROM b
```

```
INTERSECT
SELECT * FROM c
EXCEPT
SELECT * FROM d
EXCEPT
SELECT * FROM E
```

is equivalent to the fully-parenthesized query

```
( ( SELECT * FROM a
    UNION
    ( SELECT * FROM b
      INTERSECT
      SELECT * FROM c) )
  EXCEPT
  SELECT * FROM d )
EXCEPT
SELECT * FROM e
```

Streaming set operators

UNION ALL is the only set operator that can be applied to streams. Both sides of the operator must be streams; it is an error if one side is a stream and the other is a relation.

For example, the following query produces a stream of orders taken over the phone or via the web:

```
SELECT STREAM *
  FROM PhoneOrders
UNION ALL
SELECT STREAM *
  FROM WebOrders
```

Rowtime generation. The rowtime of a row emitted from streaming UNION ALL is the same as the timestamp of the input row.

Rowtime bounds. Amazon Kinesis Analytics ensures the property, required of all streams, that the ROWTIME column is ascending by merging the incoming rows on the basis of timestamp. If the first set has rows timestamped 10:00 and 10:30 and the second set has only reached 10:15, Amazon Kinesis Analytics will pause the first set, and wait for the second set to reach 10:30. It would be advantageous, in this case, if the producer of the second set were to send a Rowtime Bound in this guide.

VALUES operator

The VALUES operator expresses a constant relation in a query. (See also the discussion of VALUES in the topic SELECT in this guide.)

VALUES can be used as a top-level query, as follows:

```
VALUES 1 + 2 > 3;
EXPR$0
======
FALSE
VALUES
    (42, 'Fred'),
    (34, 'Wilma');
```

```
EXPR$0 EXPR$1
====== ======
    42 Fred
    34 Wilma
```

Note that the system has generated arbitrary column names for anonymous expressions. You can assign column names by putting VALUES into a subquery and using an AS clause:

```
SELECT *
FROM (
    VALUES
        (42, 'Fred'),
        (34, 'Wilma')) AS t (age, name);
AGE NAME
=== =====
 42 Fred
 34 Wilma
```

SELECT statement

SELECT retrieves rows from streams. You can use SELECT as a top-level statement, or as part of a query involving set operations, or as part of another statement, including (for example) when passed as a query into a UDX. For examples, see the topics INSERT, IN, EXISTS, CREATE PUMP (p. 126) in this guide.

The subclauses of the SELECT statement are described in the topics SELECT clause (p. 137), GROUP BY clause (p. 148), Streaming GROUP BY, ORDER BY clause (p. 158), HAVING clause (p. 147), WINDOW clause (Sliding Windows) (p. 152) and WHERE clause (p. 149) in this guide.

Syntax

```
<select> :=
   SELECT [ STREAM] [ DISTINCT | ALL ]
   <select-clause>
   FROM <from-clause>
   [ <where-clause> ]
   [ <group-by-clause> ]
   [ <having-clause> ]
   [ <window-clause> ]
   [ <order-by-clause> ]
```

The STREAM keyword and the principle of streaming SQL

The SQL query language was designed for querying stored relations, and producing finite relational results.

The foundation of streaming SQL is the STREAM keyword, which tells the system to compute the time differential of a relation. The time differential of a relation is the change of the relation with respect to time. A streaming query computes the change in a relation with respect to time, or the change in an expression computed from several relations.

To ask for the time-differential of a relation in Amazon Kinesis Analytics, we use the STREAM keyword:

```
SELECT STREAM * FROM Orders
```

If we start running that query at 10:00, it will produce rows at 10:15 and 10:25. At 10:30 the query is still running, waiting for future orders:

```
ROWTIME  orderId custName   product quantity
======== ======= ========== ======= ========
10:15:00     102 Ivy Black  Rice           6
10:25:00     103 John Wu    Apples         3
```

Here, the system is saying 'At 10:15:00 I executed the query SELECT * FROM Orders and found one row in the result that was not present at 10:14:59.999'. It generates the row with a value of 10:15:00 in the ROWTIME column because that is when the row appeared. This is the core idea of a stream: a relation that keeps updating over time.

You can apply this definition to more complicated queries. For example, the stream

```
SELECT STREAM * FROM Orders WHERE quantity > 5
```

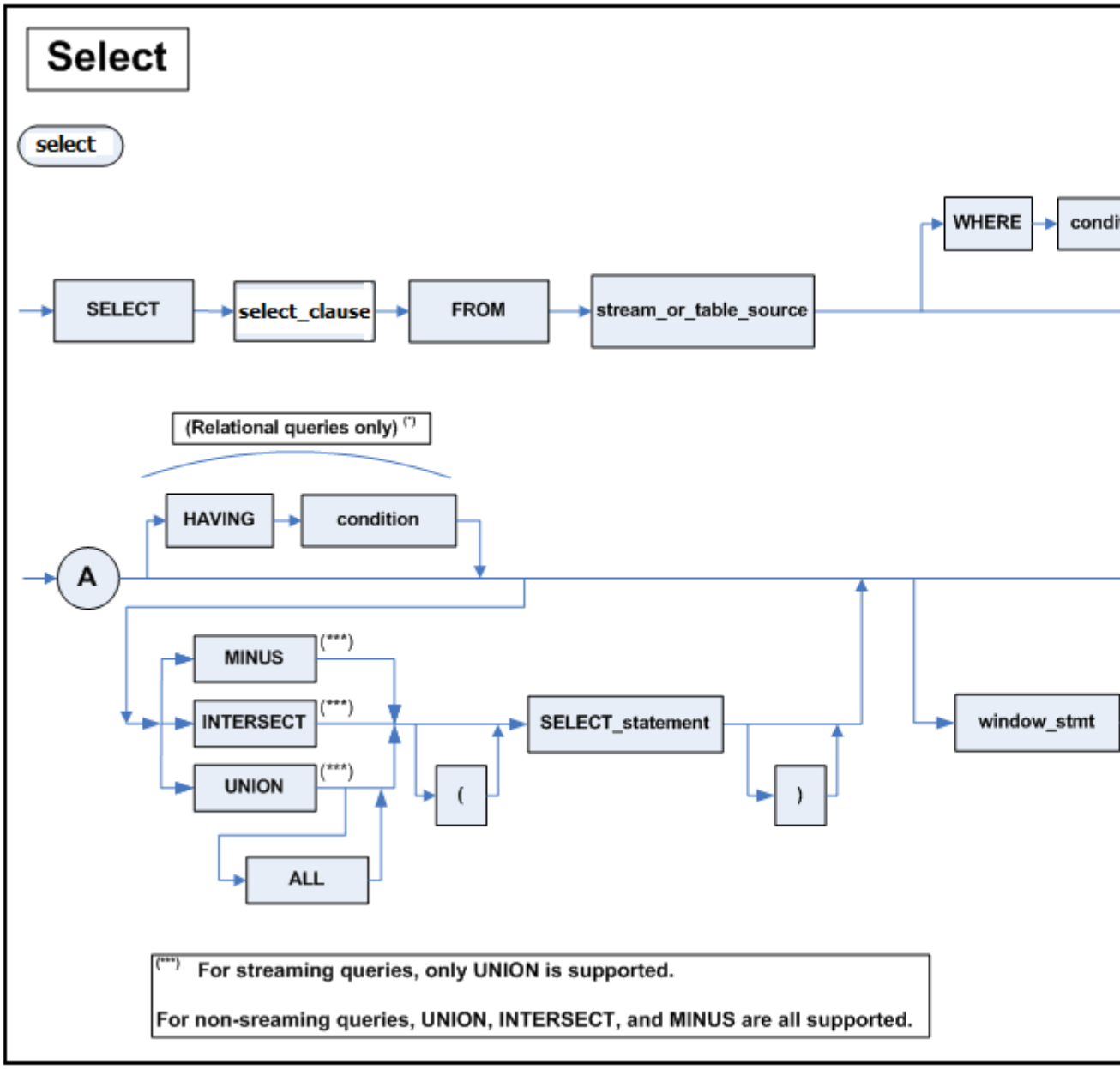
has a row at 10:15 but no row at 10:25, because the relation

```
SELECT * FROM Orders WHERE quantity > 5
```

goes from empty to one row when order 103 is placed at 10:15, but is not affected when order 104 is placed at 10:25.

We can apply the same logic to queries involving any combination of SQL operators. Queries involving JOIN, GROUP BY, subqueries, set operations UNION, INTERSECT, EXCEPT, and even qualifiers such as IN and EXISTS, are well-defined when converted to streams. Queries combining streams and stored relations are also well-defined.

Syntax



SELECT ALL and SELECT DISTINCT

If the ALL keyword is specified, the query does not eliminate duplicate rows. This is the default behavior if neither ALL nor DISTINCT is specified.

If the DISTINCT keyword is specified, a query eliminates rows that are duplicates according to the columns in the SELECT clause.

Note that for these purposes, the value NULL is considered equal to itself and not equal to any other value. These are the same semantics as for GROUP BY and the IS NOT DISTINCT FROM operator.

Streaming SELECT DISTINCT

SELECT DISTINCT can be used with streaming queries as long as there is a non-constant monotonic expression in the SELECT clause. (The rationale for the non-constant monotonic expression is the

same as for streaming GROUP BY.) Amazon Kinesis Analytics emits rows for SELECT DISTINCT as soon as they are ready.

If ROWTIME is one of the columns in the SELECT clause, it is ignored for the purposes of duplicate-elimination. Duplicates are eliminated on the basis of the other columns in the SELECT clause.

For example:

```
SELECT STREAM DISTINCT ROWTIME, prodId, FLOOR(Orders.ROWTIME TO DAY)
FROM Orders
```

displays the set of unique products that are ordered in any given day.

If you are doing "GROUP BY floor(ROWTIME TO MINUTE)" and there are two rows in a given minute -- say 22:49:10 and 22:49:15 -- then the summary of those rows is going to come out timestamped 22:50:00. Why? Because that is the earliest time that row is complete.

Note: "GROUP BY ceil(ROWTIME TO MINUTE)" or "GROUP BY floor(ROWTIME TO MINUTE) - INTERVAL '1' DAY" would give identical behavior.

It is not the value of the grouping expression that determines row completion, it's when that expression changes value.

If you want the rowtimes of the output rows to be the time they are emitted, then in the following example you would need to change from form 1 to use form 2 instead:

```
(Form 1)
   select distinct floor(s.rowtime to hour), a,b,c
   from s
(Form 2)
   select min(s.rowtime) as rowtime, floor(s.rowtime to hour), a, b, c
   from s
   group by floor(s.rowtime to hour), a, b, c
```

As a separate example, if you have 'FIRST_VALUE(orders.rowtime) AS rowtime' in the SELECT clause and you have no aggregate functions, you can see the first order of each product immediately by saying SELECT DISTINCT floor(ROWTIME TO DAY), prodId FROM orders.

SELECT clause

The <select-clause> uses the following items after the STREAM keyword:

```
 <select-list> :=
   <select-item> { , <select-item> }...
<select-item> :=
   <select-expression> [ [ AS ] <simple-identifier> ]
<simple-identifier> :=
   <identifier> | <quoted-identifier>
<select-expression> :=
   <identifier> . *  | *  | <expression>
```

Expressions

Each of these expressions may be:

- a scalar expression

- a call to an Aggregate Functions (p. 55), if this is an aggregating query (see GROUP BY clause (p. 148))
- a call to an Analytic Functions (p. 66), if this is not an aggregating query
- the wildcard expression * expands to all columns of all relations in the FROM clause
- the wildcard expression alias.* expands to all columns of the relation named alias
- the ROWTIME (p. 160)
- a CASE expression (p. 138)

Each expression may be assigned an alias, using the AS column_name syntax. This is the name of the column in the result set of this query. If this query is in the FROM clause of an enclosing query, this will be the name that will be used to reference the column. The number of columns specified in the AS clause of a stream reference must match the number of columns defined in the original stream.

Amazon Kinesis Analytics has a few simple rules to derive the alias of an expression that does not have an alias. The default alias of a column expression is the name of the column: for example, EMPS.DEPTNO is aliased DEPTNO by default. Other expressions are given an alias like EXPR$0. You should not assume that the system will generate the same alias each time.

In a streaming query, aliasing a column AS ROWTIME has a special meaning: For more information, see ROWTIME (p. 160).

Note
All streams have an implicit column called ROWTIME. This column may impact your use of the syntax 'AS t(c1, c2, ...)' that is now supported by SQL:2008. Previously in a FROM clause you could only write

```
SELECT ... FROM r1 AS t1 JOIN r2 as t2
```

but t1 and t2 would have the same columns as r1 and t2. The AS syntax enables you to rename r1's columns by writing the following:

```
SELECT ... FROM r1 AS t1(a, b, c)
```

(r1 must have precisely 3 columns for this syntax to work).

If r1 is a stream, then ROWTIME is implicitly included, but it doesn't count as a column. As a result, if a stream has 3 columns without including ROWTIME, you cannot rename ROWTIME by specifying 4 columns. For example, if the stream Bids has three columns, the following code is invalid.

```
SELECT STREAM * FROM Bids (a, b, c, d)
```

It is also invalid to rename another column ROWTIME, as in the following example.

```
SELECT STREAM * FROM Bids (ROWTIME, a, b)
```

because that would imply renaming another column to ROWTIME. For more information about expressions and literals, see Expressions and Literals (p. 118).

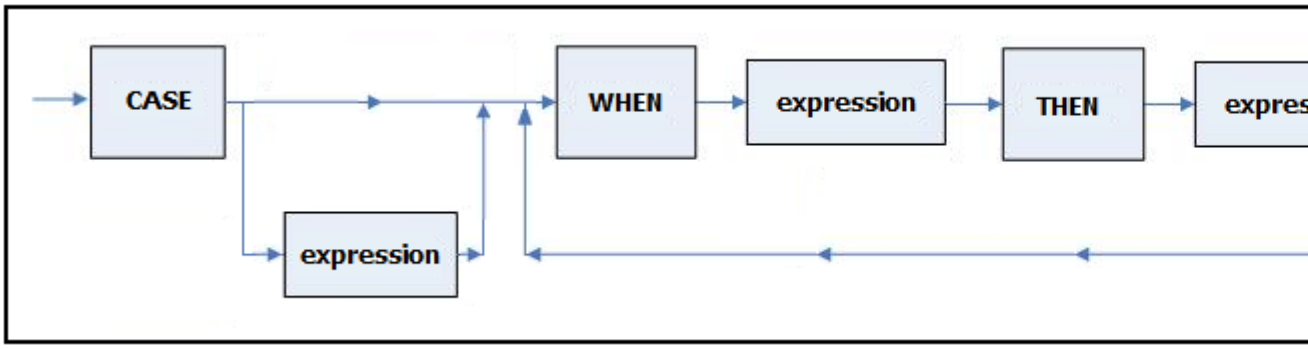
CASE expression

The CASE expression enables you to specify a set of discrete test expressions and a specific return-value (expression) for each such test. Each test expression is specified in a WHEN clause; each

return-value expression is specified in the corresponding THEN clause. Multiple such WHEN-THEN pairs can be specified.

If you specify a comparison-test-expression before the first WHEN clause, then each expression in a WHEN clause is compared to that comparison-test-expression. The first one to match the comparison-test-expression causes the return-value from its corresponding THEN clause to be returned. If no WHEN clause expression matches the comparison-test-expression, the return-value is null unless an ELSE clause is specified, in which case the return-value in that ELSE clause is returned.

If you do not specify a comparison-test-expression before the first WHEN clause, then each expression in a WHEN clause is evaluated (left to right) and the first one to be true causes the return-value from its corresponding THEN clause to be returned. If no WHEN clause expression is true, the return-value is null unless an ELSE clause is specified, in which case the return-value in that ELSE clause is returned.



VALUES

VALUES uses expressions to calculate one or more row values, and is often used within a larger command. When creating more than one row, the VALUES clause must specify the same number of elements for every row. The resulting table-columns data-types are derived from the explicit or inferred types of the expressions appearing in that column. VALUES is allowed syntactically wherever SELECT is permitted. See also the discussion of VALUES as an operator, in the topic Query in this guide.

SYNTAX

```
VALUES ( expression [, ...] ) [, ...]
    [ ORDER BY sort_expression [ ASC | DESC | USING operator ] [, ...] ]
```

VALUES is a SQL operator, on a par with SELECT and UNION, enabling the following types of actions:

- You can write VALUES (1), (2) to return two rows each with a single anonymous column.
- You can write VALUES (1, 'a'), (2, 'b') to return two rows of two columns.
- You can name the columns using AS, as in the following example:

```
SELECT * FROM (VALUES (1, 'a'), (2, 'b')) AS t(x, y)
```

The most important use of VALUES is in an INSERT statement, to insert a single row:

```
  INSERT INTO emps (empno, name, deptno, gender)
    VALUES (107, 'Jane Costa', 22, 'F');
```

However, you can also insert multiple rows:

```
   INSERT INTO Trades (ticker, price, amount)
     VALUES ('MSFT', 30.5, 1000),
            ('ORCL', 20.25, 2000);
```

When you use VALUES in the FROM clause of a SELECT statement, the entire VALUES clause must be enclosed in parentheses, consistent with the fact that it operates as a query, not a table expression. For additional examples, see FROM clause (p. 140).

Note
Using INSERT with streams engages some additional considerations as to rowtimes, pumps, and INSERT EXPEDITED. For more information, see INSERT (p. 128).

FROM clause

The FROM clause is the source of rows for a query.

```
 <from-clause> :=
    FROM <table-reference> { , <table-reference> }...
 <table-reference> :=
    <table-name> [ <table-name> ] [ <correlation> ]
| <joined-table>
 <table-name> :=  <identifier>
 <table-over> :=  OVER <window-specification>
 <window-specification> :=
 (   <window-name>
| <query_partition_clause>
| ORDER BY <order_by_clause>
| <windowing_clause>
    )
 <windowing-clause> :=
    { ROWS | RANGE }
    { BETWEEN
    { UNBOUNDED PRECEDING
      | CURRENT ROW
      | <value-expression> { PRECEDING | FOLLOWING }
    }
       AND
    { UNBOUNDED FOLLOWING
      | CURRENT ROW
      | <value-expression> { PRECEDING | FOLLOWING }
    }
    | { UNBOUNDED { PRECEDING | FOLLOWING }
      | CURRENT ROW
      | <value-expression> { PRECEDING | FOLLOWING }
    }
    }
```

For charts on window-specification and windowing-clause, see the WINDOW clause (Sliding Windows) (p. 152) under the Window statement.

```
<correlation> :=
   [ AS ] <correlation-name> [ '(' <column> { , <column> }... ')' ]
<joined-table> :=
   <table-reference> CROSS JOIN <table-reference>
 | <table-reference> NATURAL <join-type> JOIN <table-reference>
 | <table-reference> <join-type> JOIN <table-reference>
```

```
      [ USING '(' <column> { , <column>}... ')'
      | ON <condition>
      ]
<join-type> :=
   INNER
 | <outer-join-type> [ OUTER ]
<outer-join-type> :=
   LEFT
 | RIGHT
 | FULL
```

Relations

Several types of relation can appear in a FROM clause:

- A named relation (table, stream)
- A subquery enclosed in parentheses.
- A join combining two relations (see the topic JOIN in this guide).
- A transform expression.

Subqueries are described in more detail in the topic Query in this guide.

Here are some examples of subqueries:

```
// set operation as subquery
// (finds how many departments have no employees)
SELECT COUNT(*)
FROM (
  SELECT deptno FROM Dept
  EXCEPT
  SELECT deptno FROM Emp);
// table-constructor as a subquery,
// combined with a regular table in a join
SELECT *
FROM Dept AS d
  JOIN (VALUES ('Fred', 10), ('Bill', 20)) AS e (name, deptno)
  ON d.deptno = e.deptno;
```

Unlike subqueries in other parts of the SELECT statement, such as in the WHERE clause (p. 149) clause (WHERE Condition Clause (p. 122)), a subquery in the FROM clause cannot contain correlating variables. For example:

```
// Invalid query. Dept.deptno is an illegal reference to
// a column of another table in the enclosing FROM clause.
SELECT *
FROM Dept,
  (SELECT *
   FROM Emp
   WHERE Emp.deptno = Dept.Deptno)
```

FROM clause with multiple relations

If a FROM clause contains multiple, comma-separated relations, the query constructs the cartesian product of those relations; that is, it combines each row from each relation with each row from every other relation.

The comma in the FROM clause is therefore equivalent to the CROSS JOIN operator.

Correlation names

Each relation in the FROM clause can have a correlation name assigned using AS correlation-name. This name is an alternative name by which the relation can be referenced in expressions throughout the query. (Even though the relation may be a subquery or stream, it is conventionally called a 'table alias' to distinguish it from column aliases defined in the SELECT clause.)

Without an AS clause, a named relation's name becomes its default alias. (In streaming queries, the OVER clause does not prevent this default assignment from happening.)

An alias is necessary if a query uses the same named relation more than once, or if any of the relations are subqueries or table expressions.

For example, in the following query, the named relation EMPS is used twice; once with its default alias EMPS, and once with an assigned alias MANAGERS:

```
SELECT EMPS.NAME || ' is managed by ' || MANAGERS.NAME
FROM LOCALDB.Sales.EMPS,
  LOCALDB.Sales.EMPS AS MANAGERS
WHERE MANAGERS.EMPNO = EMPS.MGRNO
```

An alias can optionally be followed by a list of columns:

```
SELECT e.empname,
FROM LOCALDB.Sales.EMPS AS e(empname, empmgrno)
```

OVER clause

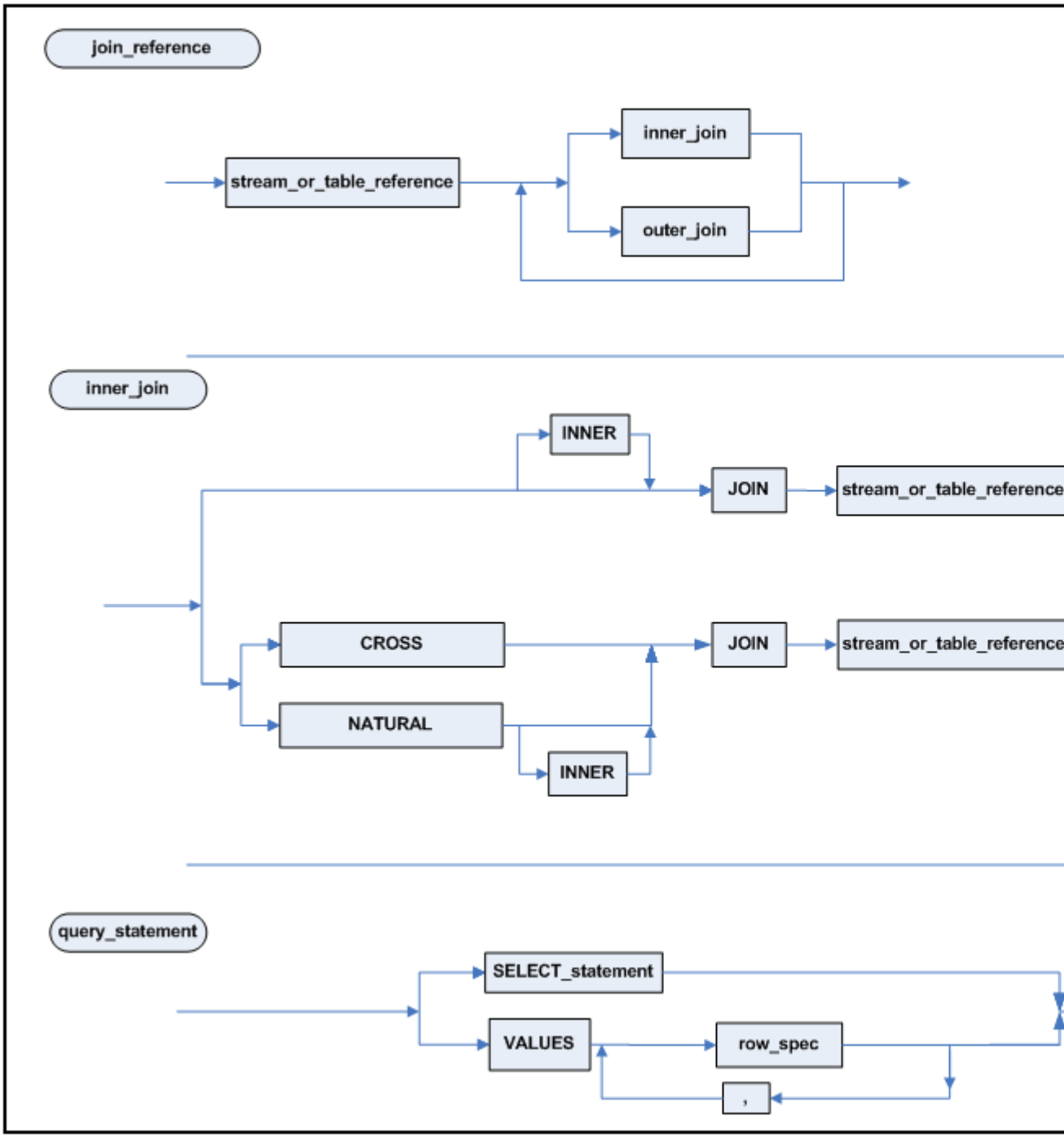
The OVER clause is only applicable for streaming joins. For more detail, see the topic JOIN clause (p. 142) in this guide.

JOIN clause

Amazon Kinesis Analytics supports joining a stream with another stream and/or joining a stream with a reference table.

Syntax Chart for the JOIN Clause

A join combines two relations according to some condition. The relation resulting from a join has the columns of the left and right inputs to the join.



Note

Join types

There are five types of joins:

INNER JOIN (or just JOIN)	Returns all pairs of rows from left and right for which the join condition evaluates to TRUE.

LEFT OUTER JOIN (or just LEFT JOIN)	As INNER JOIN, but rows from the left are kept even if they do not match any rows on the right; NULL values are generated on the right.
RIGHT OUTER JOIN (or just RIGHT JOIN)	As INNER JOIN, but rows from the right are kept even if they do not match any rows on the left; NULL values are generated on the left for these rows.
FULL OUTER JOIN (or just FULL JOIN)	As INNER JOIN, but rows from both sides are kept even if they do not match any rows on the other side; NULL values are generated on the other side for these rows.
CROSS JOIN	Returns the cartesian product of the inputs: every row from the left is paired with every row from the right. A cross join never has an ON or USING condition.

For information about streaming joins, see Streaming JOINs (p. 144), which includes stream-to-stream joins and conceptual stream-to-table joins using a UDX.

The NATURAL keyword is actually a condition. It is described with ON and USING in Join Conditions later in this topic.

Join Conditions

All types of join except CROSS JOIN accept a join condition.

There are three ways to specify a join condition:

- The ON condition evaluates a Boolean condition. It is the most general and powerful way to specify a join condition.
- USING (column {, column }...) matches columns from left and right. For each named column, left and right must both have a column of that name. r1 JOIN r2 USING (c1, c2) is equivalent to r1 JOIN r2 ON r1.c1 = r1.c1 AND r1.c2 = r2.c2.
- Inserting the NATURAL keyword before INNER, LEFT, RIGHT or FULL JOIN creates a condition that matches each pair of columns that have the same name on left and right side of the join.

A WHERE clause (p. 149) (WHERE Condition Clause (p. 122)) can achieve a similar effect to ON, except that it filters the rows after they have been emitted from the join. For an inner join, WHERE is equivalent to ON, but for an outer join, the partially NULL rows are only generated correctly if the condition is evaluated for each pair of candidate rows, and a WHERE clause cannot do that. For more details, see the topic WHERE clause (p. 149) in this guide.

Join limitations

Amazon Kinesis Analytics does not support left, right or full outer join operations applied to relations.

Streaming JOINs

JOIN can be used in a streaming query provided that at least one of the relations being joined is a stream. Streaming joins work just like regular table joins, but subject to the considerations implicit in dealing with streams, that is, rolling windows and rowtimes:

- (a) the rowtime of the input row for which a match was not found, or
- (b) the later bound of the window of the other input stream at the point any possible match passed out of the window.

- For an inner join, the rowtime of an output row is the later of the rowtimes of the two input rows. This is also true for an outer join in which matching input rows are found.
- For outer joins in which a match is not found, the rowtime of an output row is the later of the following two times:

- Rolling windows -- A window defined on a stream is a rolling (sliding) window. As the current time progresses, the window excludes some rows while adding others. As a result, rows output by a join with a sliding window are generated incrementally. It is important to note that an output row is only produced once by a match on a given pair of columns from the left and right input streams. In other words, an output row already produced by a prior match will not be produced anew by a subsequent identical match.
- Output rowtimes -- All output rows are produced in order of non-descending rowtime. (It is valid to have multiple output rows with the same rowtime.)
- As a rule, the rowtime of a given output row is the rowtime at the point it was possible to calculate the output row. In other words:

All streaming joins are implicitly windowed joins between the affected streams. If no explicit window is specified, the window specification CURRENT ROW is used.

Stream-to-table Joins

If one of the relations is a stream and the other is a finite relation, it is referred to as a stream-table join. For each row in the stream, the query looks up the row or rows in the table that match the join condition. Amazon Kinesis Analytics accomplishes conceptual stream-table joins by using a UDX named TableLookup.

For example, Orders is a stream and PriceList is a table. The effect of the join is to add price list information to the order.

Stream-to-stream Joins

If both of the relations being joined are streams, it is referred to as a stream-stream join. Clearly it is not practical to join the entire history of the left stream to the entire history of the right, so at least one stream must be restricted to a time window by the use of an OVER clause.

The OVER clause defines a window of rows that are to be considered for joining at a given time.

The window can be time-based or row-based:

- A time-based window uses the RANGE keyword, and defines the window as the set of rows whose ROWTIME column falls within a particular time interval of the query's current time.
- A row-based window uses the ROWS keyword, and defines the window as a given count of rows before or after the row with the current timestamp.

To illustrate, let's look at an example.

```
SELECT STREAM ROWTIME, o.orderId, o.ROWTIME AS orderTime
FROM Shipments AS s
  JOIN Orders OVER (RANGE INTERVAL '1' HOUR PRECEDING) AS o
  ON o.orderId = s.orderId;
```

We have not specified an OVER clause for the Shipments stream, so at any moment in time we are considering matching the current row of the Shipments stream against rows from the Orders stream in the hour preceding.

`Orders` and `Shipments` inputs produce the following output:

Orders		Shipments	
rowtime	orderId	rowtime	orderId
10:00	100		
10:10	101		
10:20	102		
10:25	103		
		10:30	101
10:40	104		
		10:45	100
		10:55	103
		11:05	103
		11:30	104

The output rows are as follows:

rowtime	orderId	orderTime
10:30	101	10:10
10:45	100	10:00
10:55	103	10:25
11:05	103	10:25
11:30	104	10:40

First of all, notice that output rows have the timestamp of the Shipments.ROWTIME column, and are sorted in that order. Order 100, 101 and 104 are each matched by a shipment within the window, and order 103 is matched by two shipments. But order 102 is omitted, because its shipment is not made within its one hour time window 10:20-11:20.

The window specification may also be the name of a window defined in the WINDOW clause (Sliding Windows) (p. 152). However, windows specified by name have the same limitation as windows specified inline: see the subtopic entitled Allowed and Disallowed Window Specifications of the WINDOW clause topic.

Additional window limitations specific to streaming joins are as follows:

- PARTITION BY must not be present.
- ORDER BY, if present, must sort by the ROWTIME column of one of the inputs.

Rowtime generation

The timestamp of the generated row is the earliest time that rows necessary to make the match are in both sides' windows. For example, in the previous example, the output row (10:45, 100, 10:00) could only be made when the left window is 10:00-10:00 and the right window is 10:00-11:00. A minute

earlier, the windows would have been 09:59-09:59 and 09:59-10:59, in which case the necessary row would have been in the right hand window but not the left.

Rowtime bounds

Rowtime bounds received on the left and right side of the join help the join to make progress. In the above example, the (10:30, 101, 10:10) output record can be emitted only when the order 100's window has expired. That expiration was only evident when the (11:05, 103) row arrived in the Shipments stream, taking the time past 11:00. If the process writing the Shipments stream had sent an 11:01 rowtime bound, the output record could have been emitted 4 minutes sooner.

Multi-way JOINs

To do a three way join, you use a joined table-reference as the table-reference in a JOIN statement. Here, stream/table 1 (b1) relates to stream/table 2 (asks) and stream/table 2 relates to stream/table 3 (b2), on the column "ticker."

```
select stream * from bids over (range interval '1' hour preceding) as b1
       join asks over (range interval '2' second preceding)
           on b1."ticker" = asks."ticker"
       join bids over (range interval '3' minute preceding) as b2
           on b2."ticker" = asks."ticker";
```

HAVING clause

The HAVING clause in a SELECT specifies a condition to apply within a group or aggregate. In other words, HAVING filters rows after the aggregation of the GROUP BY clause has been applied. Since HAVING is evaluated after GROUP BY, it can only reference expressions constructed (or derivable) from grouping keys, aggregate expressions, and constants. (These are the same rules that apply to expressions in the SELECT clause of a GROUP BY query.) A HAVING clause must come after any GROUP BY clause and before any ORDER BY clause. HAVING is like WHERE clause (p. 149), but applies to groups. Results from a HAVING clause represent groupings or aggregations of original rows, whereas results from a WHERE clause are individual original rows.

In non-streaming applications, if there is no GROUP BY clause, GROUP BY () is assumed (though since there are no grouping expressions, expressions can consist only of constants and aggregate expressions). In streaming queries, HAVING cannot be used without a GROUP BY clause.

WHERE and HAVING can both appear in a single SELECT statement. The WHERE selects from the stream or table those individual rows that satisfy its condition (the WHERE-condition). The GROUP BY criteria apply only to the rows selected by the WHERE condition.

Such a grouping, for example "GROUP BY CustomerID", can be further qualified by a HAVING-condition, which then selects aggregations of rows satisfying its condition within the specified grouping. For example, "GROUP BY ClientID HAVING SUM(ShipmentValue) > 3600" would select only those clients whose various shipments that fit the WHERE criteria also had values that added up to exceed 3600.

See the WHERE clause syntax chart for the conditions, which applies to both HAVING and WHERE clauses.

The condition must be a Boolean predicate expression. The query returns only rows for which the predicate evaluates to TRUE.

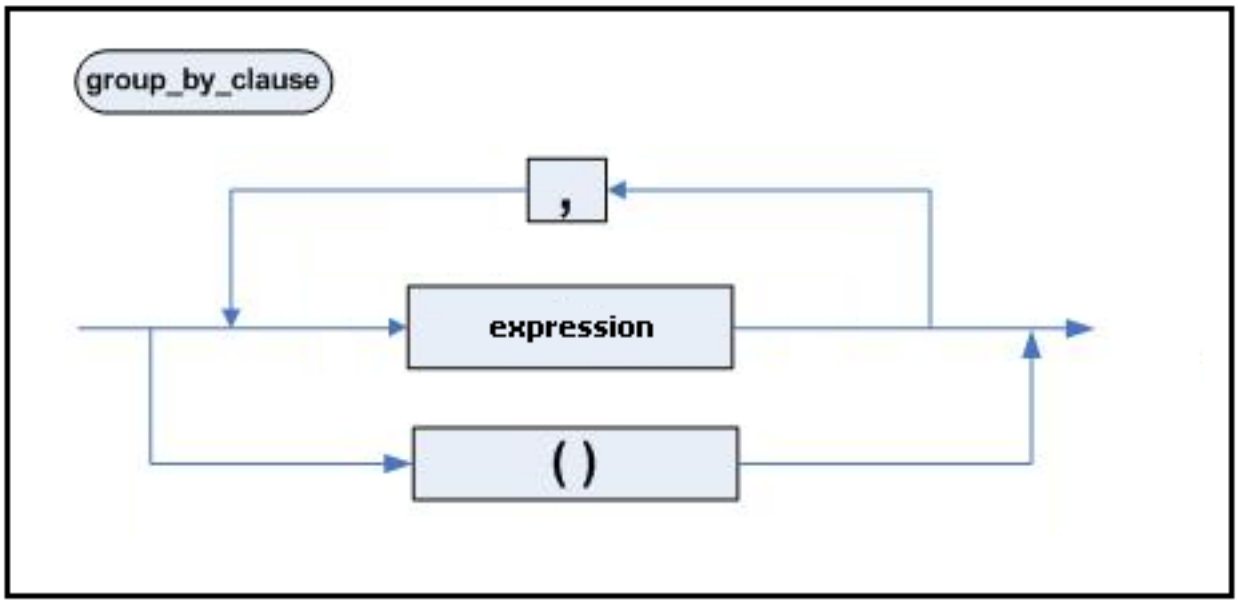
The example below shows a streaming query that displays products for which there are more than $1000 of orders in the past hour.

```
SELECT STREAM "prodId"
FROM "Orders"
```

```
GROUP BY FLOOR("Orders".ROWTIME TO HOUR), "prodId"
HAVING SUM("quantity" * "price") > 1000;
```

GROUP BY clause

Syntax Chart for the GROUP BY Clause



(To see where this clause fits, see SELECT statement (p. 134))

For example, GROUP BY <column name-or-expression>, where:

- the expression can be an aggregate; and,
- any column name used in the GROUP BY clause must also be in the SELECT statement.

Additionally, a column that is not named in or derivable from the GROUP BY clause cannot appear in the SELECT statement except within aggregations, such as SUM (allOrdersValue).

What derivable means is that a column specified in the GROUP BY clause enables access to the column you want to include in the SELECT clause. If a column is derivable, the SELECT statement can specify it even though it is not explicitly named in the GROUP BY clause.

Example: If the key to a table is in the GROUP BY clause, then any of that table's columns can appear in the select-list because, given that key, such columns are considered accessible.

The GROUP BY clause groups selected rows based on the value of the grouping expressions, returning a single summary row of information for each group of rows that have identical values in all columns.

Note that for these purposes, the value NULL is considered equal to itself and not equal to any other value. These are the same semantics as for the IS NOT DISTINCT FROM operator.

Streaming GROUP BY

GROUP BY can be used in a streaming query as long as one of the grouping expressions is a non-constant monotonic or time-based expression. This requirement is necessary in order for Amazon Kinesis Analytics to make progress, as explained below.

A monotonic expression is one that always moves in the same direction: it either ascends-or-stays-the-same, or it descends-or-stays the same; it doesn't reverse direction. It does not need to be strictly ascending or strictly descending, that is, every value always above the previous one or every value always below the previous one. A constant expression falls under the definition of monotonic -- it is technically both ascending and descending -- but is clearly unsuitable for these purposes. For more information about monotonicity, see Monotonic Expressions and Operators (p. 120).

Consider the following query:

```
SELECT STREAM prodId, COUNT(*)
FROM Orders
GROUP BY prodId
```

The query is intended to compute the number of orders for each product, as a stream. However, since Orders is an infinite stream, Amazon Kinesis Analytics can never know that it has seen all orders for a given product, can never complete a particular row's total, and therefore can never output a row. Rather than allow a query that can never emit a row, the Amazon Kinesis Analytics validator rejects the query.

The syntax for streaming GROUP BY is as follows:

```
GROUP BY <monotonic or time-based expression> ,

<column name-or-expression, ...>
```

where any column name used in the GROUP BY clause needs to be in the SELECT statement; the expression can be an aggregate. Additionally, a column name that does not appear in the GROUP BY clause cannot appear in the SELECT statement except within aggregations, or if, as above, access to the column can be created from column that you specify in the GROUP BY clause.

For example, the following query, which computes the product counts per hour, uses the monotonic expression FLOOR(Orders.ROWTIME TO HOUR) is therefore valid:

```
SELECT STREAM FLOOR(Orders.ROWTIME TO HOUR) AS theHour, prodId, COUNT(*)
FROM Orders
GROUP BY FLOOR(Orders.ROWTIME TO HOUR), prodId
```

One of the expressions in the GROUP BY must be monotonic or time-based. For example GROUP BY FLOOR(S.ROWTIME) TO HOUR will yield one output row per hour for the previous hour's input rows. The GROUP BY can specify additional partitioning terms. For example, GROUP BY FLOOR(S.ROWTIME) TO HOUR, USERID will yield one output row per hour per USERID value. If you know for a fact that an expression is monotonic, you can declare it so by using the Monotonic Function (p. 44). If the actual data are not monotonic, the resulting system behavior is indeterminate: results may not be as expected or desired.

See the topic Monotonic Function (p. 44) in this guide for more details.

Duplicate rowtimes can occur in a stream, and as long as the ROWTIME value is the same, the GROUP BY operation will keep accumulating rows. In order to emit a row, the ROWTIME value has to change at some point.

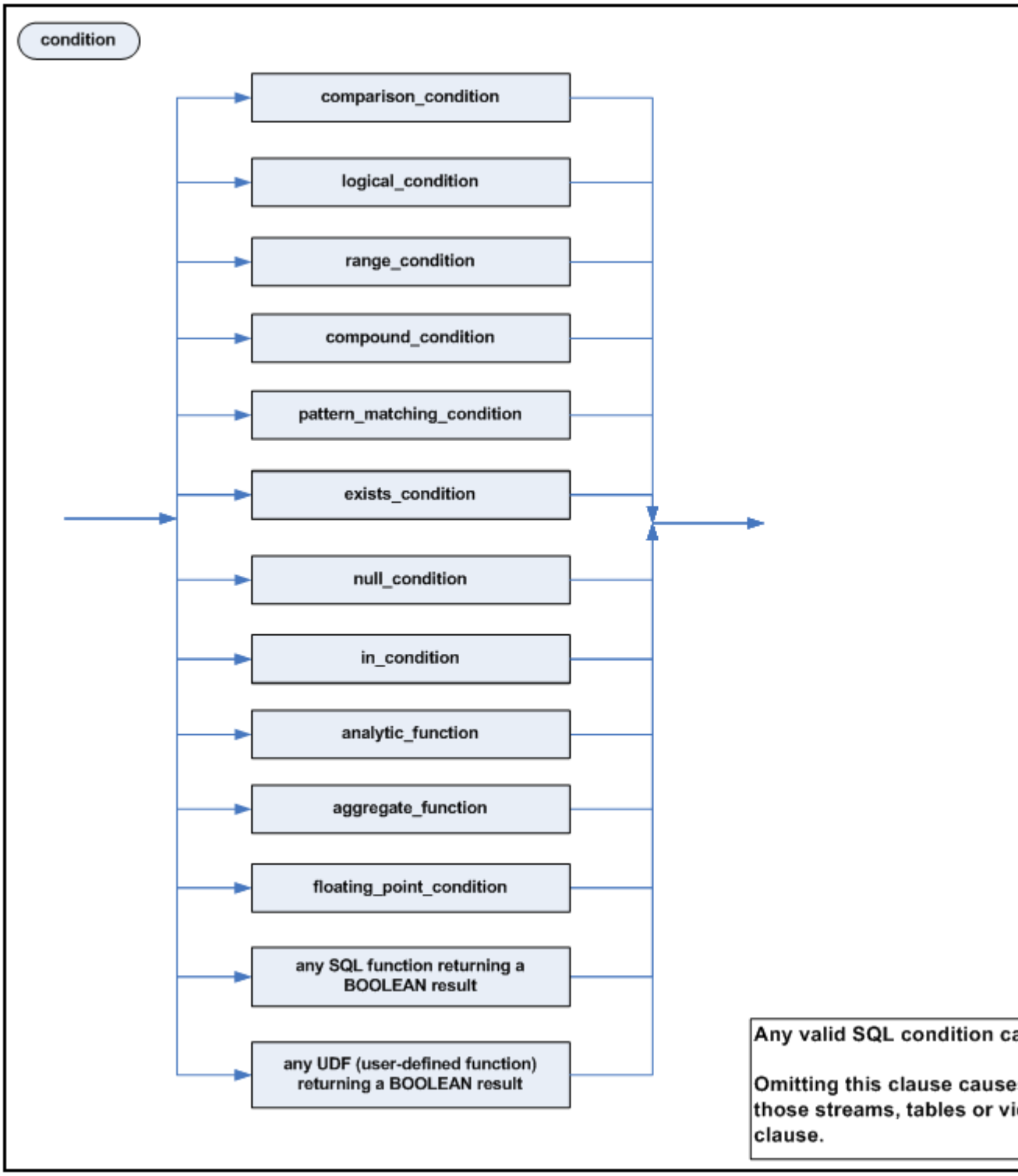
WHERE clause

The WHERE clause extracts records that meet a specified condition. The condition can be a numeric or string comparison, or use the BETWEEN, LIKE, OR IN operators: see Streaming SQL Operators (p. 11). Conditions can be combined using logical operators such as AND, OR, and NOT.

The WHERE clause is like the HAVING clause (p. 147) clause. It applies to groups, that is, results from a WHERE clause are individual original rows, whereas results from a HAVING clause represent groupings or aggregations of original rows.

WHERE and HAVING can both appear in a single SELECT statement. The WHERE selects from the stream or table those individual rows that satisfy its condition (the WHERE-condition). The GROUP BY criteria apply only to the rows selected by the WHERE condition. Such a grouping, for example "GROUP BY CustomerID", can be further qualified by a HAVING-condition, which then selects aggregations of rows satisfying its condition within the specified grouping. For example, "GROUP BY ClientID HAVING SUM(ShipmentValue) > 3600" would select only those clients whose various shipments that fit the WHERE criteria also had values that added up to exceed 3600.

To see where this clause fits into the SELECT statement, see SELECT statement (p. 134).



The condition must be a Boolean predicate expression. The query returns only rows for which the predicate evaluates to TRUE; if the condition evaluates to NULL, the row is not emitted.

The condition in the WHERE clause cannot contain windowed aggregation expressions, because if the where clause condition caused rows to be dropped, it would alter the contents of the window.

WHERE is also discussed in the topics JOIN clause (p. 142) and HAVING clause (p. 147) in this guide.

WINDOW clause (Sliding Windows)

The WINDOW clause allows you to define named window specifications that can be used in Analytic Functions (p. 66) calls and streaming JOIN clauses elsewhere in the query. The windows so defined are inherited by sub-queries of the current query.

```
<window-clause> :=
   WINDOW <window-definition> { , <window-definition> }...
<window-definition> :=
   <window-name> AS <window-specification>
<window-specification> :=
   <window-name>
 | <query_partition_clause>
 | ORDER BY <order_by_clause>
 | <windowing_clause>
<query_partition_clause> :=
  PARTITION BY <expression> { , <expression> }...
 |  ( <expression> { , <expression> }... )
```

Window clause and endpoints

Streaming SQL follows the SQL Standards for windows over a range. This means, for example that the syntax

```
WINDOW HOUR AS (RANGE INTERVAL '1' HOUR PRECEDING)
```

will include the end points of the hour

To ensure that the endpoint of the previous hour is not included, you need to use the following syntax for the window:

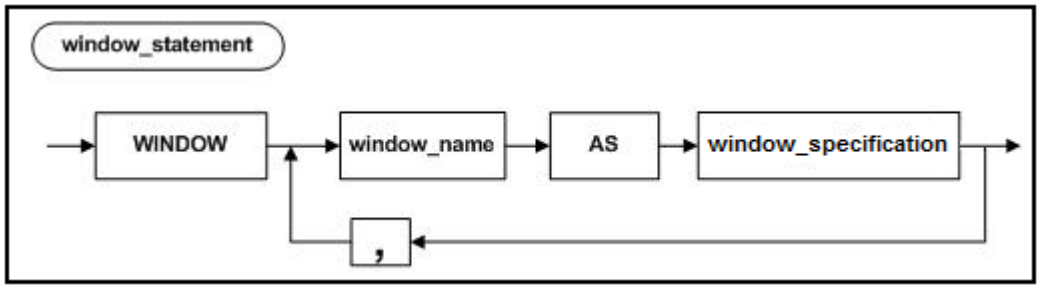
```
WINDOW HOUR AS (RANGE INTERVAL '59:59.999' MINUTE TO SECOND(3) PRECEDING);
```

See Allowed and Disallowed Window Specifications (p. 155) for more details.

Syntax Charts for Window Statement and Window Specification

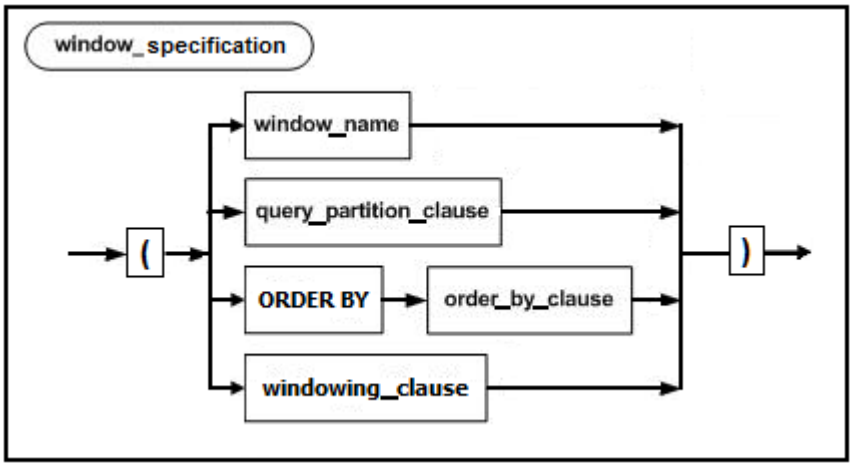
The chart for Window Statement includes the window-specification, query-partition, and windowing-clause charts that directly follow below it.

Window Statement



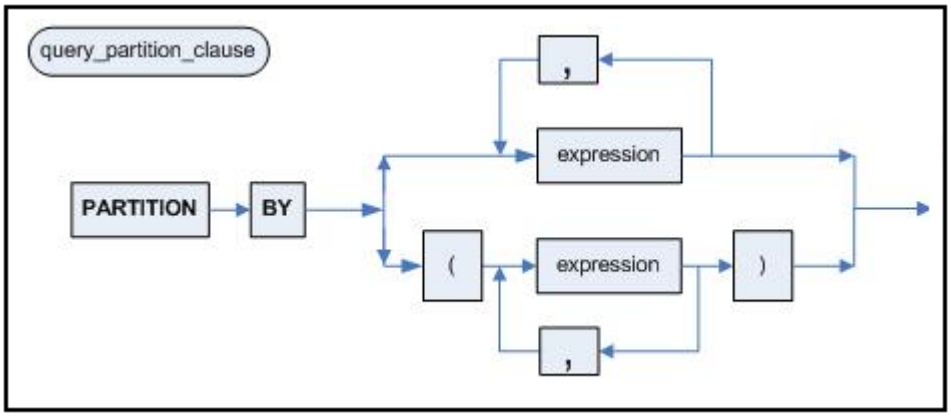
Window Specification

The "window-specification" is also referenced in the stream-or-table-reference chart that appears within the select-clause chart.



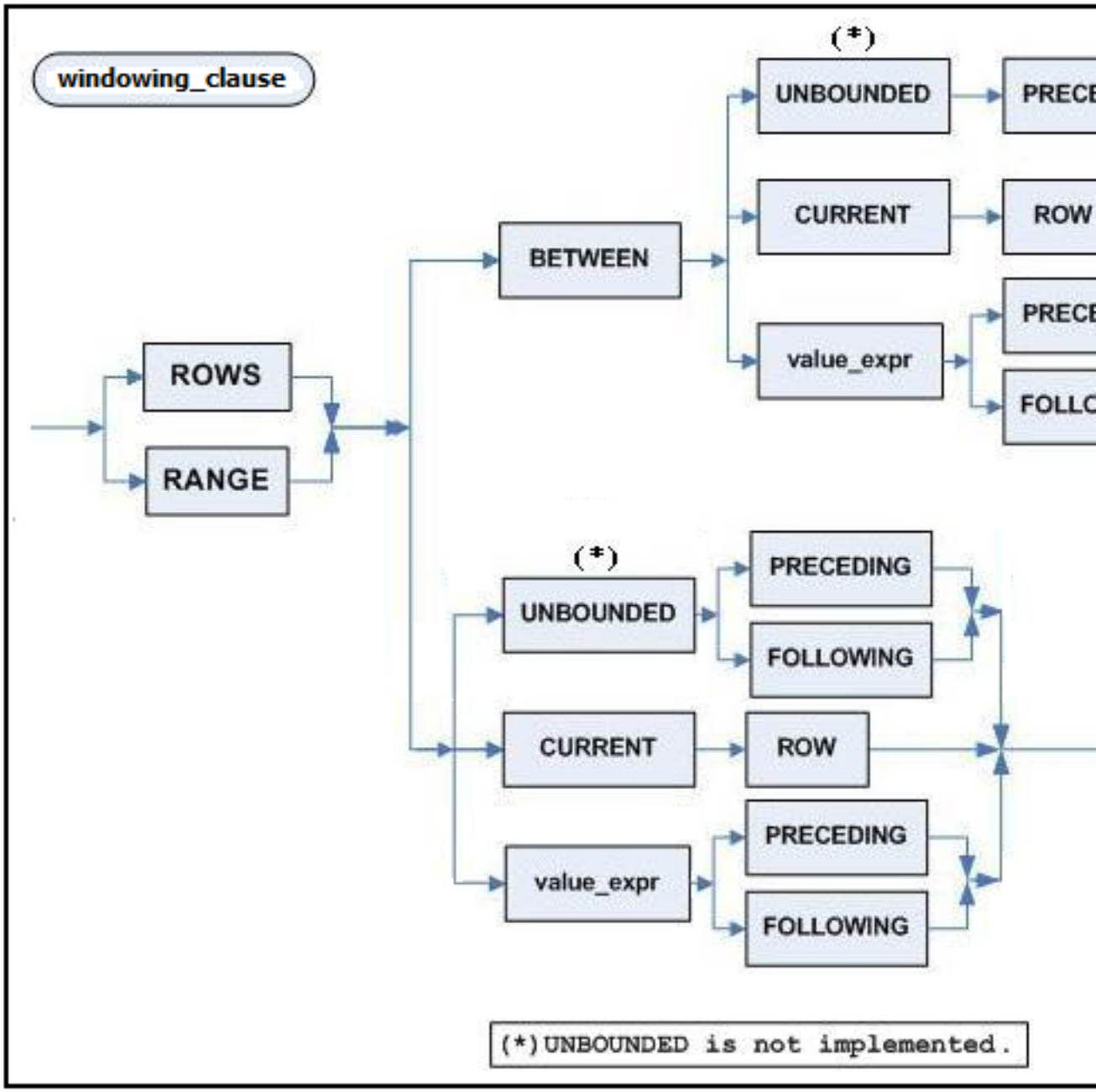
The order_by_clause chart appears after the Window examples (p. 155).

Syntax Chart for the query-partition clause



Syntax Chart for the windowing-clause

In addition to being part of the window-specification, the windowing-clause appears in the FROM clause (p. 140) clause and the Analytic Functions (p. 66) clause of an analytic function.



Note: Partitions are evaluated before windows.

```
<windowing-clause> :=
   { ROWS | RANGE }
   { BETWEEN
     { UNBOUNDED PRECEDING
     | CURRENT ROW
     | <value-expression> { PRECEDING | FOLLOWING }
     }
     AND
     { UNBOUNDED FOLLOWING
```

```
      | CURRENT ROW
      | <value-expression> { PRECEDING | FOLLOWING }
      }
    | { UNBOUNDED { PRECEDING | FOLLOWING }
      | CURRENT ROW
      | <value-expression> { PRECEDING | FOLLOWING }
      }
    }
```

Allowed and Disallowed Window Specifications

Amazon Kinesis Analytics supports nearly all windows that end with the current row.

You cannot define an infinite window, a negative-sized window, or use negative integers in the window specification. Offset windows are currently unsupported.

- Infinite windows are windows with no bounds. Typically these point into the future, which for streams is infinite. For example "ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING" is not supported, because in a streaming context such a query would not produce a result, since streams are continually expanding as new data arrives. All uses of UNBOUNDED FOLLOWING are unsupported.
- Negative windows . For example, "ROWS BETWEEN 0 PRECEDING AND 4 PRECEDING" is a window of negative size and is therefore illegal. Instead, you would use: "ROWS BETWEEN 4 PRECEDING AND 0 PRECEDING" in this case.

- Offset windows are windows that do not end with CURRENT ROW. These are not supported in the current release. For example, "ROWS BETWEEN UNBOUNDED PRECEDING AND 4 FOLLOWING" is not supported. (Window spans CURRENT ROW rather than starting or ending there.)

- Windows defined with negative integers. For example, "ROWS BETWEEN -4 PRECEDING AND CURRENT ROW" is invalid because negative integers are disallowed.

Also, the special case of ... 0 PRECEDING (and ... 0 FOLLOWING) cannot be used for windowed aggregation; instead, the synonym CURRENT ROW can be used.

For windowed aggregation, partitioned windows are allowed, but ORDER BY must not be present.

For windowed join, partitioned windows are NOT allowed, but ORDER BY can be present if it sorts by the ROWTIME column of one of the inputs.

Window examples

The following examples show a sample input data set, the definitions for several windows, and the contents of those windows at various times after 10:00, the time data starts to arrive for this example.

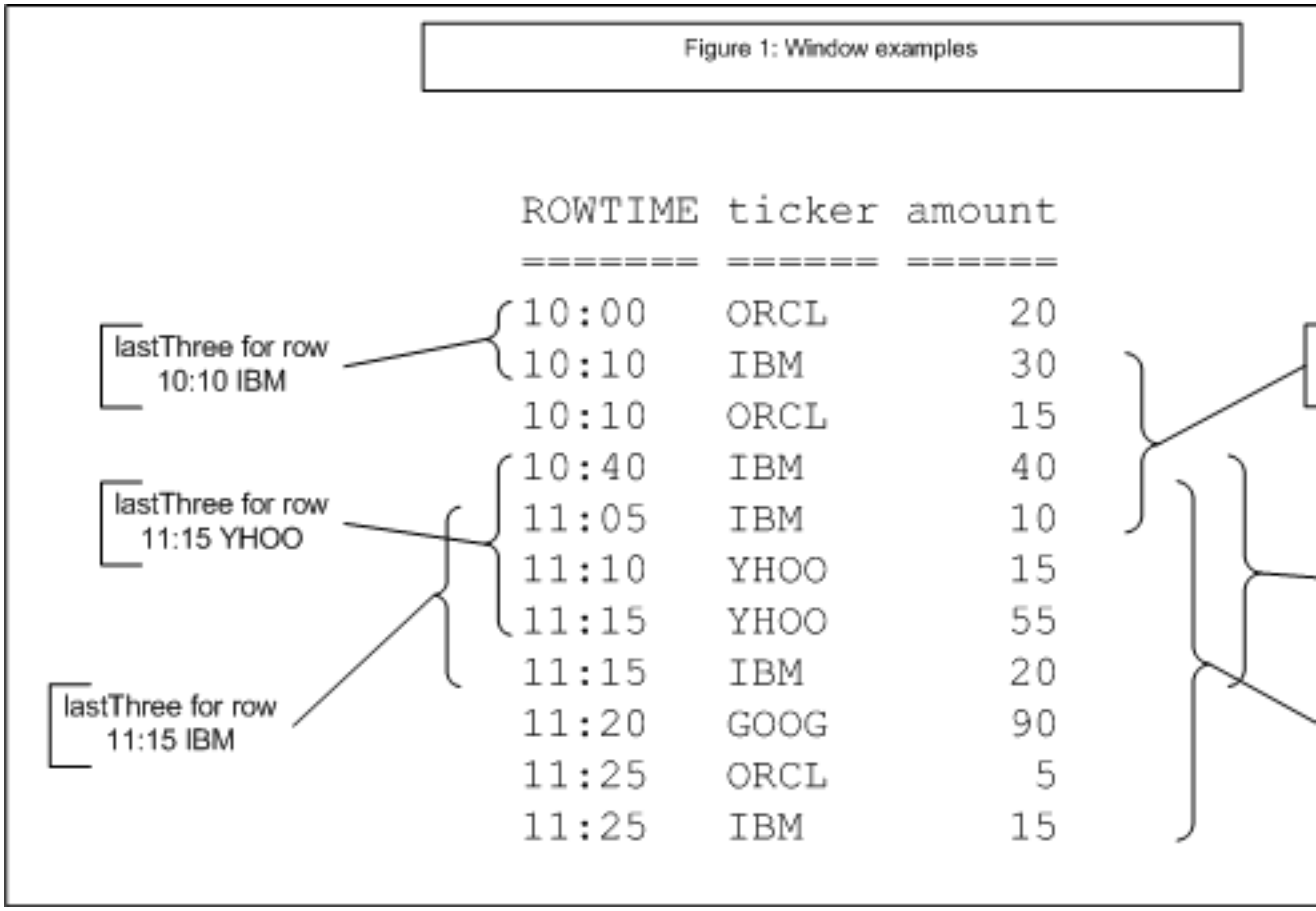
The windows are defined as follows:

```
SELECT STREAM
  ticker,
  sum(amount) OVER lastHour,
  count(*) OVER lastHour
  sum(amount) OVER lastThree
FROM Trades
WINDOW
```

```
  lastHour AS (RANGE INTERVAL '1' HOUR PRECEDING),
  lastThree AS (ROWS 3 PRECEDING),
  lastZeroRows AS (ROWS CURRENT ROW),
  lastZeroSeconds AS (RANGE CURRENT ROW),
  lastTwoSameTicker AS (PARTITION BY ticker ROWS 2 PRECEDING),
  lastHourSameTicker AS (PARTITION BY ticker RANGE INTERVAL '1' HOUR
PRECEDING)
```

First Example: time-based windows versus row-based windows

As shown on the right side of the figure below, the time-based lastHour window contains varying numbers of rows, because window membership is defined by time range.



Examples of windows containing rows

The row-based lastThree window generally contains four rows: the three preceding and the current row. However for the row 10:10 IBM, it only contains two rows, because there is no data before 10:00.

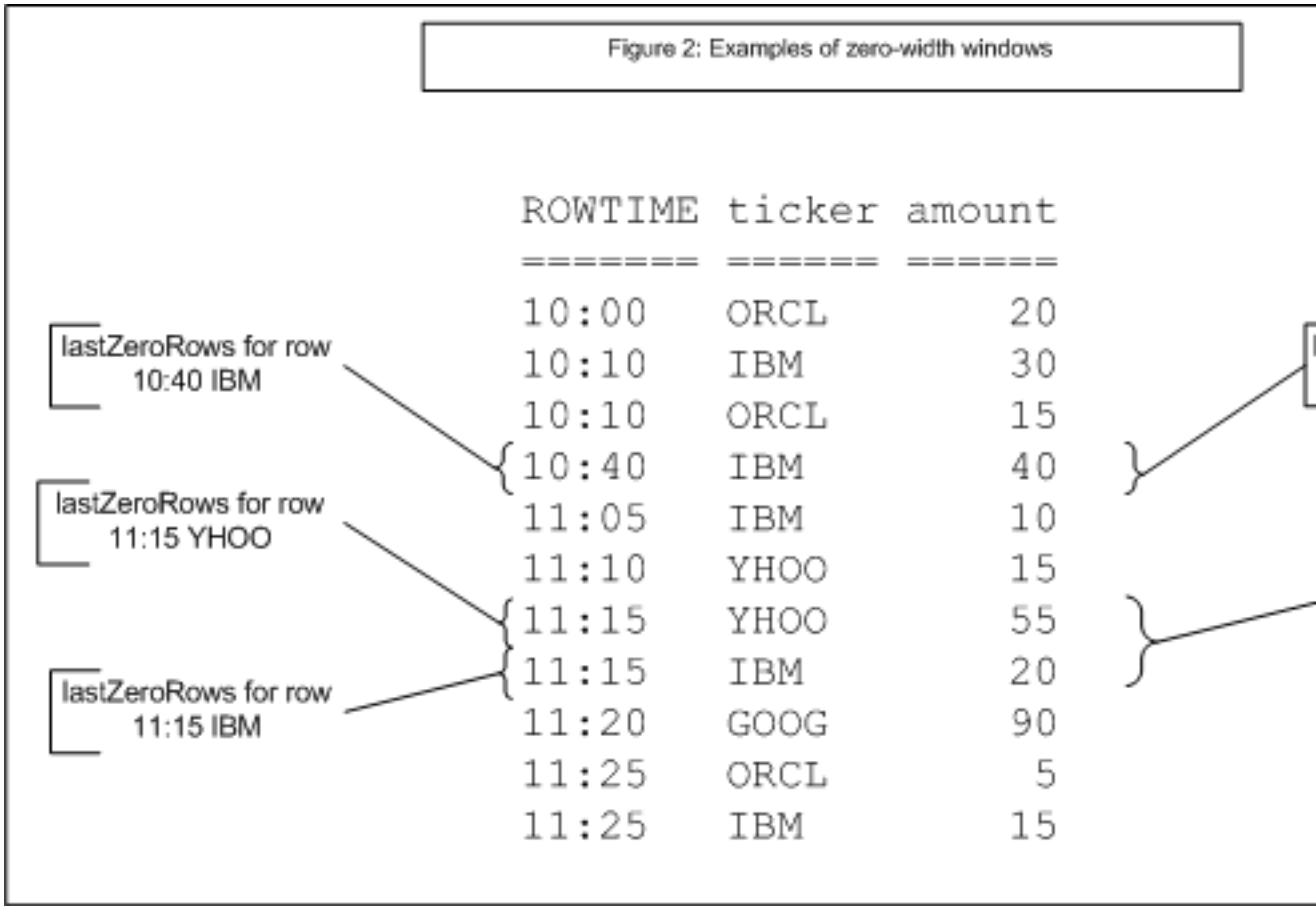
A row-based window can contain several rows whose ROWTIME value is the same, though they arrive at different times (wall-clock times). The order of such a row in the row-based window depends on its arrival time; indeed, the row's arrival time can determine which window includes it.

For example, the middle lastThree window in Figure 1 shows the arrival of a YHOO trade with ROWTIME 11:15 (and the last three trades before it). However, this window excludes the next trade, for IBM, whose ROWTIME is also 11:15 but which must have arrived later than the YHOO trade. This 11:15 IBM trade is included in the 'next' window, as is the 11:15 YHOO trade, its immediate predecessor.

Second Example: zero width windows, row-based and time-based

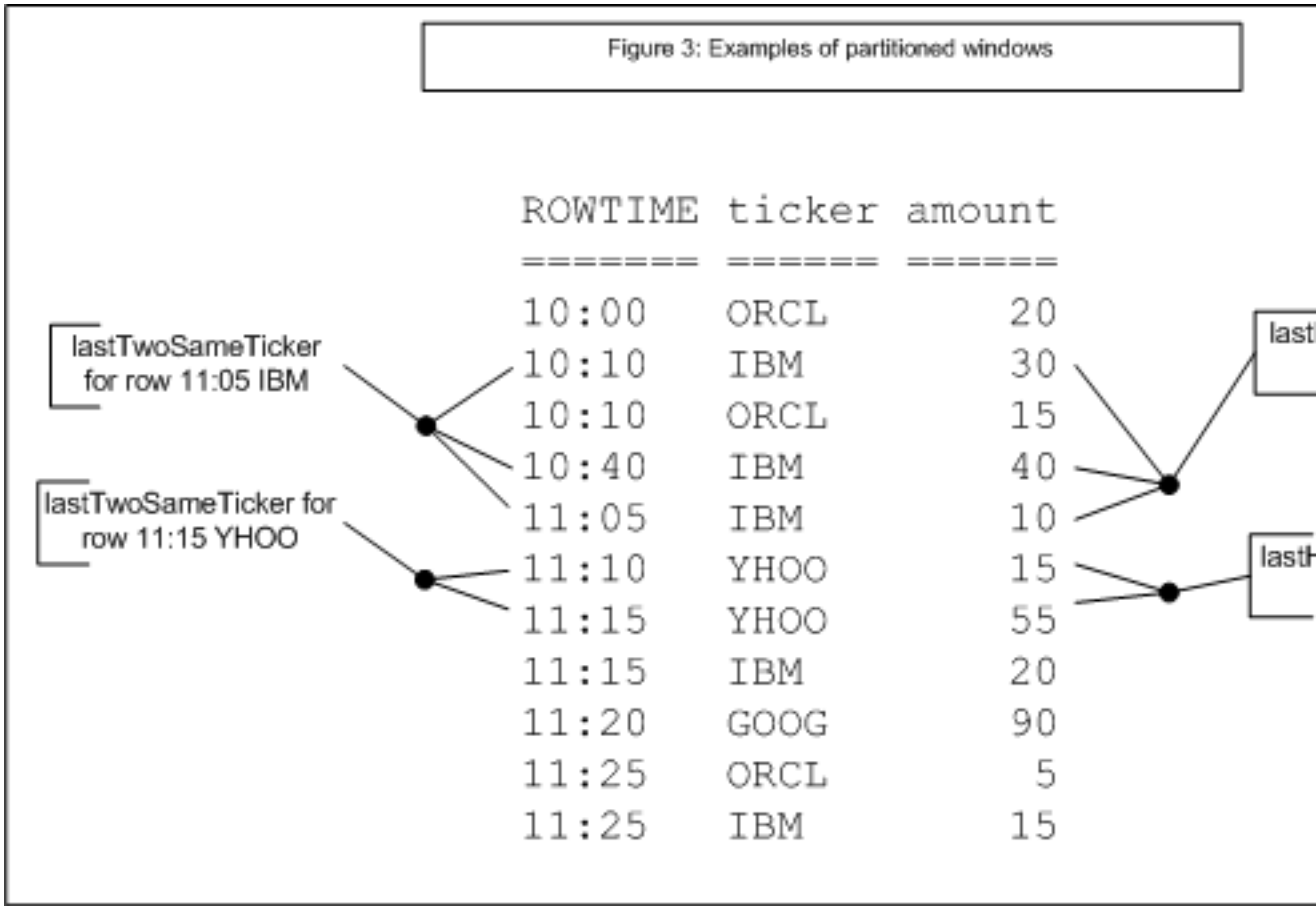
Figure 2: Examples of zero-width windows shows row-based and time-based windows of zero width. The row-based window lastZeroRows includes just the current row, and therefore always contains precisely one row. Note that ROWS CURRENT ROW is equivalent to ROWS 0 PRECEDING.

The time-based window lastZeroSeconds contains all rows with the same timestamp, of which there may be several. Note that RANGE CURRENT ROW is equivalent to RANGE INTERVAL '0' SECOND PRECEDING.



Third Example: Partitioning applied to row-based and time-based windows

Figure 3 shows windows that are similar to those in Figure 1 but with a PARTITION BY clause. For time-based window lastTwoSameTicker and the row-based window lastHourSameTicker, the window contains rows that meet the window criteria and have the same value of the ticker column. Note: Partitions are evaluated before windows.



ORDER BY clause

A streaming query can use ORDER BY if its leading expression is time-based and monotonic. For example, a streaming query whose leading expression is based on the ROWTIME column can use ORDER BY to do the following operations:

- Sort the results of a streaming GROUP BY.
- Sort a batch of rows arriving within a fixed time window of a stream.
- Perform streaming ORDER BY on windowed-joins.

The "time-based and monotonic" requirement on the leading expression means that the query

```
SELECT STREAM DISTINCT ticker FROM trades ORDER BY ticker
```

will fail, but the query

```
SELECT STREAM DISTINCT rowtime, ticker FROM trades ORDER BY ROWTIME, ticker
```

will succeed.

Streaming ORDER BY sorts rows using SQL-2008 compliant syntax for the ORDER BY clause. It can be combined with a UNION ALL statement, and can sort on expressions, such as:

```
SELECT STREAM x, y FROM t1
UNION ALL
SELECT STREAM a, b FROM t2 ORDER BY ROWTIME, MOD(x, 5)
```

The ORDER BY clause can specify ascending or descending sort order, and can use column ordinals, as well as ordinals specifying (referring to) the position of items in the select list.

Streaming ORDER BY SQL Declarations

The streaming ORDER BY clause includes the following functional attributes:

- Gathers rows until the monotonic expression in streaming ORDER BY clause does not change.
- Does not require streaming GROUP BY clause in the same statement.
- Can use any column with a basic SQL data type of TIMESTAMP, DATE, DECIMAL, INTEGER, FLOAT, CHAR, VARCHAR.
- Does not require that columns/expressions in the ORDER BY clause be present in the SELECT list of the statement.
- Applies all the standard SQL validation rules for ORDER BY clause.

The following query is an example of streaming ORDER BY:

```
SELECT STREAM state, city, SUM(amount)
FROM orders
GROUP BY FLOOR(ROWTIME TO HOUR), state, city
ORDER BY FLOOR(ROWTIME TO HOUR), state, SUM(amount)
```

T-sorting Stream Input

Amazon Kinesis Analytics real-time analytics use the fact that arriving data is ordered by ROWTIME. However, sometimes data arriving from multiple sources may not be time-synchronized.

While Amazon Kinesis Analytics can sort data from individual data sources that have been independently inserted into Amazon Kinesis Analytics application's native stream, in some cases such data may have already combined from multiple sources (such as for efficient consumption at an earlier stage in processing). At other times, high volume data sources could make direct insertion impossible.

In addition, an unreliable data source could block progress by forcing Amazon Kinesis Analytics application to wait indefinitely, unable to proceed until all connected data sources deliver. In this case, data from this source could be unsynchronized.

You can use the ORDER BY clause to resolve these issues. Amazon Kinesis Analytics uses a sliding time-based window of incoming rows to reorder those rows by ROWTIME.

Syntax

You specify the time-based parameter for sorting and the time-based window in which the streaming rows are to be time-sorted, using the following syntax:

```
 ORDER BY <timestamp_expr> WITHIN
     <interval_literal>
```

Restrictions

The T-sort has the following restrictions:

- The datatype of the ORDER BY expression must be timestamp.
- The partially-ordered expression <timestamp_expr> must be present in the select list of the query with the alias ROWTIME.
- The leading expression of the ORDER BY clause must not contain the ROWTIME function and must not use the DESC keyword.
- The ROWTIME column needs to be fully qualified. For example:

- `ORDER BY FLOOR(ROWTIME TO MINUTE), ...` fails.
- `ORDER BY FLOOR(s.ROWTIME TO MINUTE), ...` works.

If any of these requirements are not met, the statement will fail with errors.

Additional notes:

- You cannot use incoming rowtimebounds. These are ignored by the system.
- If <timestamp_expr> evaluates to NULL, the corresponding row is discarded.

ROWTIME

ROWTIME is an operator and system column that returns the time at which a particular row of a stream was created.

It is used in four distinct ways:

- As an operator
- As a system column of a stream
- As a column alias, to override the timestamp on the current row
- As an ordinary column in a table

For more details, see the topics Timestamp, ROWTIME, and CURRENT_ROW_TIMESTAMP (p. 110) in this guide.

ROWTIME operator

When used in the SELECT clause of a streaming query, without being qualified by a preceding 'alias.' , ROWTIME is an operator that evaluates to the timestamp of the row that is just about to be generated.

Its type is always TIMESTAMP NOT NULL.

ROWTIME system column

Every stream has a ROWTIME column. To reference this column from within a query, qualify it with the stream name (or alias). For example, the following join query returns three timestamp columns: the system columns of its input streams, and the timestamp of the generated row.

```
SELECT STREAM
  o.ROWTIME AS leftRowtime,
  s.ROWTIME AS rightRowtime,
  ROWTIME AS joinRowtime
FROM Orders AS o
  JOIN Shipments OVER (RANGE INTERVAL '1' HOUR FOLLOWING) AS s
  ON o.orderId = s.orderId

leftRowtime         rightRowtime        joinRowtime
```

```
=================== =================== ===================
2008-02-20 10:15:00 2008-02-20 10:30:00 2008-02-20 10:15:00
2008-02-20 10:25:00 2008-02-20 11:15:00 2008-02-20 10:25:00
2008-02-20 10:25:30 2008-02-20 11:05:00 2008-02-20 10:25:30
```

As it happens, leftRowtime is always equal to joinRowtime, because the join is specified such that the output row timestamp is always equal to the ROWTIME column from the Orders stream.

It follows that every streaming query has a ROWTIME column. However, the ROWTIME column is not returned from a top-level JDBC query unless you explicitly include it in the SELECT clause. For example:

```
CREATE STREAM Orders(
  "orderId" INTEGER NOT NULL,
  "custId" INTEGER NOT NULL);
SELECT columnName
FROM ALL_STREAMS;

columnName
==========
orderId
custId

SELECT STREAM *
FROM Orders;

orderId custId
======= ======
    100    501
    101     22
    102    699

SELECT STREAM ROWTIME, *
FROM Orders;

ROWTIME             orderId custId
=================== ======= ======
2008-02-20 10:15:00     100    501
2008-02-20 10:25:00     101     22
2008-02-20 10:25:30     102    699
```

This is mainly to ensure compatibility with JDBC: the stream Orders declares two columns, so it makes sense that "SELECT STREAM *" should return two columns.

Setting a row's timestamp

Amazon Kinesis Analytics assigns each row of a stream a timestamp value based on the streaming relational operators that created it. You can override that value by giving one column or expression in the query a column alias of ROWTIME, though this is not recommended.

For example, the following query returns rows with a constant timestamp:

```
SELECT STREAM
  TIMESTAMP '1970-01-01 00:00:00' AS ROWTIME,
  *
FROM Orders
```

In fact, it is not strictly necessary to use AS ROWTIME. A column is promoted to the row's timestamp column if its name, derived by the usual rules for column aliases, turns out to be ROWTIME. For example:

```
// s.ROWTIME implicitly becomes the timestamp of the generated row
SELECT STREAM
  o.orderId,
  s.ROWTIME
FROM Orders AS o
  JOIN Shipments OVER (RANGE INTERVAL '1' HOUR FOLLOWING) AS s
  ON o.orderId = s.orderId
// invalid, because no stream can have more than one ROWTIME column
SELECT STREAM
  o.orderId,
  o.ROWTIME,
  s.ROWTIME
FROM Orders AS o
  JOIN Shipments OVER (RANGE INTERVAL '1' HOUR FOLLOWING) AS s
  ON o.orderId = s.orderId
```

A word of caution. Amazon Kinesis Analytics requires that rows have ascending timestamps, and this is difficult to achieve. Therefore overriding row timestamps is not recommended in general.

Reserved Words and Keywords

Reserved Words

The following is a list of reserved words in Amazon Kinesis Analytics application as of version 5.0.1.

ABS	EXPLAIN	PRIMARY
ALL	EXP_AVG	PROCEDURE
ALLOCATE	EXTERNAL	RANGE
ALLOW	EXTRACT	RANK
ALTER	FALSE	READS
ANALYZE	FETCH	REAL
AND	FILTER	RECURSIVE
ANY	FIRST_VALUE	REF
ARE	FLOAT	REFERENCES
ARRAY	FLOOR	REFERENCING
AS	FOR	REGR_AVGX
ASENSITIVE	FOREIGN	REGR_AVGY
ASYMMETRIC	FREE	REGR_COUNT
AT	FROM	REGR_INTERCEPT
ATOMIC	FULL	REGR_R2
AUTHORIZATION	FUNCTION	REGR_SLOPE
AVG	FUSION	REGR_SXX
BEGIN	GET	REGR_SXY
BETWEEN	GLOBAL	RELEASE
BIGINT	GRANT	RESPECT
BINARY	GROUP	RESULT

BIT	GROUPING	RETURN
BLOB	HAVING	RETURNS
BOOLEAN	HOLD	REVOKE
BOTH	HOUR	RIGHT
BY	IDENTITY	ROLLBACK
CALL	IGNORE	ROLLUP
CALLED	IMPORT	ROW
CARDINALITY	IN	ROWS
CASCADED	INDICATOR	ROW_NUMBER
CASE	INITCAP	SAVEPOINT
CAST	INNER	SCOPE
CEIL	INOUT	SCROLL
CEILING	INSENSITIVE	SEARCH
CHAR	INSERT	SECOND
CHARACTER	INT	SELECT
CHARACTER_LENGTH	INTEGER	SENSITIVE
CHAR_LENGTH	INTERSECT	SESSION_USER
CHECK	INTERSECTION	SET
CHECKPOINT	INTERVAL	SIMILAR
CLOB	INTO	SMALLINT
CLOSE	IS	SOME
CLUSTERED	JOIN	SORT
COALESCE	LANGUAGE	SPECIFIC
COLLATE	LARGE	SPECIFICTYPE
COLLECT	LAST_VALUE	SQL
COLUMN	LATERAL	SQLEXCEPTION
COMMIT	LEADING	SQLSTATE
CONDITION	LEFT	SQLWARNING
CONNECT	LIKE	SQRT
CONSTRAINT	LIMIT	START
CONVERT	LN	STATIC
CORR	LOCAL	STDDEV

CORRESPONDING	LOCALTIME	STDDEV_POP
COUNT	LOCALTIMESTAMP	STDDEV_SAMP
COVAR_POP	LOWER	STOP
COVAR_SAMP	MATCH	STREAM
CREATE	MAX	SUBMULTISET
CROSS	MEMBER	SUBSTRING
CUBE	MERGE	SUM
CUME_DIST	METHOD	SYMMETRIC
CURRENT	MIN	SYSTEM
CURRENT_CATALOG	MINUTE	SYSTEM_USER
CURRENT_DATE	MOD	TABLE
CURRENT_DEFAULT_TRANSFORM_GROUP	MODIFIES	TABLESAMPLE
CURRENT_PATH	MODULE	THEN
CURRENT_ROLE	MONTH	TIME
CURRENT_SCHEMA	MULTISET	TIMESTAMP
CURRENT_TIME	NATIONAL	TIMEZONE_HOUR
CURRENT_TIMESTAMP	NATURAL	TIMEZONE_MINUTE
CURRENT_TRANSFORM_GROUP_FOR_TYPE	NCHAR	TINYINT
CURRENT_USER	NCLOB	TO
CURSOR	NEW	TRAILING
CYCLE	NO	TRANSLATE
DATE	NODE	TRANSLATION
DAY	NONE	TREAT
DEALLOCATE	NORMALIZE	TRIGGER
DEC	NOT	TRIM
DECIMAL	NTH_VALUE	TRUE
DECLARE	NULL	TRUNCATE
DEFAULT	NULLIF	UESCAPE
DELETE	NUMERIC	UNION
DENSE_RANK	OCTET_LENGTH	UNIQUE
DEREF	OF	UNKNOWN
DESCRIBE	OLD	UNNEST

DETERMINISTIC	ON	UPDATE
DISALLOW	ONLY	UPPER
DISCONNECT	OPEN	USER
DISTINCT	OR	USING
DOUBLE	ORDER	VALUE
DROP	OUT	VALUES
DYNAMIC	OUTER	VARBINARY
EACH	OVER	VARCHAR
ELEMENT	OVERLAPS	VARYING
ELSE	OVERLAY	VAR_POP
END	PARAMETER	VAR_SAMP
END-EXEC	PARTITION	WHEN
ESCAPE	PERCENTILE_CONT	WHENEVER
EVERY	PERCENTILE_DISC	WHERE
EXCEPT	PERCENT_RANK	WIDTH_BUCKET
EXEC	POSITION	WINDOW
EXECUTE	POWER	WITH
EXISTS	PRECISION	WITHIN
EXP	PREPARE	WITHOUT
		YEAR

Document History

The following table describes the documentation for this release of Amazon Kinesis Analytics SQL Reference.

- **API version: 2015-08-14**
- **Latest documentation update:** August 11, 2016

Change	Description	Date
New guide	This is the first release of the *Amazon Kinesis Analytics SQL Reference* guide.	August 11, 2016