SYBASE®
An SAP® Company

CCL Reference Guide

Sybase CEP Option R4

DOCUMENT ID: DC01031-01-0400-02
LAST REVISED: April 2011

Sybase, Inc., One Sybase Drive, Dublin, CA 94568

Contents

Introduction

An introduction to CCL.

Continuous Computation Language

Continuous Computation Language (CCl) is used to manage and analyze streaming data.

CCL is based on the Structured Query Language (SQL) used with relational databases. While CCL and SQL are not identical, familiarity with SQL will enable you to learn and use CCL.

CCL Lexical Conventions

CCL Statement enable users to use tabs, carriage returns, and spaces within the definition of the statement.

CCL statements can include one or more tabs, carriage returns, spaces, or comments anywhere a space occurs within the definition of the statement. For example, the following two statements are evaluated the same way:

```
INSERT INTO AverageStockPrice
SELECT stocks.price, AVG(stocks.price)
FROM stocks
GROUP BY stocks.Symbol
```

```
INSERT INTO AverageStockPrice
SELECT stocks.price,
AVG(stocks.price)
FROM stocks
GROUP BY stocks.Symbol
```

Case is insignificant in CCL keywords, also called reserved words, and identifiers (stream names, column names, parameter names, and so on). However, CCL text literals are case-sensitive.

CCL names include CCL query identifiers, as well as project and module names, adapter names, aliases and the names of windows. Names and identifiers must start with a letter. Subsequent characters can be letters, digits, or underscores (_). The prefixes C8_ and XML are reserved for Sybase® CEP Engine names.

None of the CCL reserved words can be used as an identifier. All reserved words are listed in Reserved Words.

CCL programs support the following UTF-8 characters:

#x9	tab
#xA	carriage return
#xD	line feed
#x20-#xD7FF	Legal characters of Unicode and ISO/IEC 10646
#xE000-#xFFFD	
#x10000-#x10FFFF	

If you need to use other UTF-8 characters, you can use the corresponding integer and convert it to the desired character with the CHR function.

Order of CCL Statement Evaluation

Rules for which order the CCL Statements are evaluated.

CCL statements inside a project execute in order, according to the following rules:

- Statements with data dependencies on other statements execute after the statements on which they are dependent. For example, if one statement publishes rows to a second statement, the first statement executes before the second. This is also true in more complex data dependency cases. For example, if a data stream is joined to an unnamed window based on the same stream, the window is updated before the join executes.
- If statements do not include data dependencies, they execute in the order in which they appear in the project's query modules; statements in a submodule execute after statements in a parent module.

Language Components

This section outlines the different usable languages components.

Data Types

Different data types are assigned to columns in a CCL stream.

Each value that Sybase CEP Engine manipulates has a *data type*. When you define a data stream in CCL, you must specify a data type for each of its columns. Note that Sybase CEP Engine can implicitly convert some data types to the correct type for a column. You can also use conversion functions to generate the correct data type.

The following table summarizes CCL data types (note that all data types can also be Null):

Data Type	Description
BLOB	Binary Large Object. BLOB columns contain varying amounts of data as a sequence of bytes. Maximum size is platform-dependent, but no less than 65535 bytes. Sybase CEP Engine supports comparison between BLOBs, by comparing the same byte from each BLOB value in sequence. If all bytes are identical, the two values are considered to be equal. If the two BLOBs contain different values, the smaller value is the one with a smaller value in the first byte that differs.
BOOLEAN	TRUE or FALSE.
FLOAT	A 64-bit floating point number with binary precision. The maximum number for this type corresponds to IEEE standards.
INTEGER	A 32-bit integer.
INTERVAL	Represents an integer number of microseconds between two timestamps, using 64 bits of precision.
LONG	A 64-bit integer.
STRING	Variable-length character string, with byte values encoded in UTF-8. Maximum string length is platform-dependent, but no less than 65535 bytes.
TIMESTAMP	Date and time, with a precision of microseconds
XML	A sequence of XML element trees (a forest). Maximum size is platform-dependent, but no less than 65535 bytes.

The following table lists the minimum and maximum values for numeric data types:

Data Type	Minimum/Maximum
FLOAT	Approximately 16 significant digits.
INTEGER	-2147483648 to +2147483647 (-2^{31} to 2^{31}-1). Overflow wraps silently.
INTERVAL	-99999999 days 23:59:59.999999 to +99999999 days 23:59:59.999999.
LONG	-9223372036854775808 to +9223372036854775807 (-2^{63} to 2^{63}-1). Overflow wraps silently.
TIMESTAMP	1970-01-01 00:00:00.000000 to 2099-12-31 23:59:59.999999. Timestamp values between 0001-01-01 00:00:00.000000 and 1969-12-31 23: 59:59.999999 and between 2100-01-01 00:00:00.000000 and 9999-12-31 23: 59:59.999999 are also valid, but may result in errors related to leap years and daylight savings time. Timestamps prior to midnight January 1, 1970 are represented as negative numbers.

Float

Floats are used to represent floating-point numbers such as decimal points anywhere within the number.

Float is a 64-bit data type used to represent floating-point numbers. Floating-point numbers can either include a decimal point anywhere within the number or the decimal point entirely. An exponent can optionally be used following the number to increase the number's range, for example:

```
1.777 e-20
```

Binary floating-point numbers differ from Integer in the way the values are represented internally during processing by Sybase CEP Engine. Binary floating-point numbers are stored using binary precision (the digits 0 and 1). Such a storage scheme cannot represent all values exactly. Converting a value from decimal to binary may result in an error, but the error is often eliminated when the number is converted back to decimal precision.

XML data type

XML data type stores a sequence of one or more XML element trees.

An XML document has one root element tree, which is treated by Sybase CEP Engine as a sequence of one XML element. However, the Sybase CEP XML data type can also store multiple elements, which are not joined by a single root. This is sometimes called an XML forest.

It can also store a value of NULL.

The Sybase CEP XML data type does not support comparison. Two XML values cannot be compared for equality or inequality. Thus XML cannot be used in places that require comparison. This precludes the use of the XML data type in a GROUP BY, ORDER BY, or PER clause and in the SMALLEST and LARGEST functions.

The XML data type uses the Infoset data model. This means that the structure of the tree is represented in the data model, but not the details of the string encoding. For example, `<a></a>` and `<a/>` are identical in Infoset.

Implicit Data Type Conversions

Sybase CEP automatically converts data types to other data types, expressions to integers, long, and data types to Strings.

Sybase CEP automatically converts certain data types to other data types when necessary. Sybase CEP Engine automatically converts expressions that evaluate to Integer that are destined for a Long or Float function argument, operator argument, or column. Sybase CEP Engine also automatically converts expressions that evaluate to Long that are destined for a Float function argument, operator argument, or column.

Also, if needed and if possible, Sybase CEP Engine automatically converts any other data type to String and from String to any other data type. To implicitly convert to String, Sybase CEP Engine uses TO_STRING(). For Timestamp, the format is YYYY-MM-DD HH24:MI:SS.FF TZD. To implicitly convert from String, Sybase CEP Engine uses the appropriate TO_xxx() function (such as TO_BLOB). For Timestamp, the format is the same. The value Null converts to Null, no matter what the types are. If Sybase CEP Engine detects an ambiguity, then an error is generated.

Data Type Conversions

Sybase CEP is able to perform datatype conversions.

CCL Data Type Conversions

Conversion of different data types conducted by Sybase CEP.

The following table shows all permitted CCL data type conversions (also known as type casting). The source datatypes are listed in the leftmost column and the destination datatypes are listed in the top row of the chart. “X” indicates that the conversion is supported; a blank space indicates that the conversion is not supported.

Permitted Conversions	Blob	Boolean	Float	Integer	Interval	Long	String	Time-stamp	XML
Blob							X		
Boolean							X		

Float				X	X	X	X	X	
Integer			X			X	X		
Interval			X			X	X		
Long			X	X	X		X	X	
String	X	X	X	X	X	X		X	X
Timestamp			X			X	X		
XML							X		

The following table shows the implicit (automatic) conversions Sybase CEP Engine performs when necessary. The source datatypes are listed in the leftmost column and the destination datatypes are listed in the top row of the chart. “X” indicates that the conversion is supported; a blank space indicates that the conversion is not supported. Note that the Sybase CEP compiler generates a warning message whenever it performs implicit type casting to or from a String:

Implicit Conversions	**Blob**	**Boolean**	**Float**	**Integer**	**Interval**	**Long**	**String**	**Time-stamp**	**XML**
Blob							X		
Boolean							X		
Float							X		
Integer			X			X	X		
Interval							X		
Long			X				X		
String	X	X	X	X	X	X		X	X
Timestamp							X		
XML							X		

Conversion Between CCL and ODBC and Oracle

Convert CCL columns into ODBC/ SQL and Oracle.

The two tables on this page show how CCL column data types are converted into ODBC/SQL and Oracle database types and vice versa. The table assumes a Native Oracle Driver (OCI based).

Table 1. How CCL Types Convert to ODBC and Oracle Types

CCL Type	ODBC Database Type	Oracle Database Type
BLOB	Base64 String	Base64 String
BOOLEAN	Integer	NUMBER (false=0, true=1)
FLOAT	Double Precision	NUMBER
INTEGER	Integer	NUMBER
INTERVAL	BIGINT	NUMBER
LONG	BIGINT	NUMBER
STRING	VARCHAR (implementation defined length)	VARCHAR/VARCHAR2 (maximum length 4096)
TIMESTAMP	TIMESTAMP	DATE

Note that, before using a BLOB column in a SQL SELECT statement within a CCL database subquery, the BLOB must first be converted to a base64- encoded string. See the Sybase CEP *Integration Guide* for more information on handling BLOB data.

Note that when the FLOAT value is expressed in scientific notation, ORACLE limits the exponent range to values between -130 and 125. CCL FLOAT values do not have this limitation, but must fall within this range when passing values to an Oracle database FLOAT column to avoid errors.

Table 2. How ODBC and Oracle Types Convert to CCL Types

Oracle Database Type	ODBC Type	CCL Type
NUMBER (false=0, true=1)	Integer	BOOLEAN
NUMBER	Double Precision	FLOAT
NUMBER	Integer	INTEGER
NUMBER	BIGINT	INTERVAL
NUMBER	BIGINT	LONG
CHAR(N)	CHAR(N)	STRING
VARCHAR/VARCHAR2	VARCHAR	STRING
DATE	TIMESTAMP	TIMESTAMP

Note that Sybase supports one-byte character sets; it does not support multi-byte character sets such as Unicode.

An error message will appear if the value that you are converting from will not fit into the data type that you are converting to.

Conversion Between CCL and Q

Learn how to convert CCL to q datatypes, q types to CCL, and q values to CCL rows.

If you wish to convert a CCL datatype to a q datatype other than the type to which it automatically converts, explicitly cast the CCL datatype as the desired q type, as described in EXECUTE STATEMENT DATABASE.

Table 3. How CCL datatypes convert to q datatypes

CCL datatype	Q datatype
BLOB	list of byte
BOOLEAN	boolean
FLOAT	float
INTEGER	int
INTERVAL	long
LONG	long
STRING	list of char
TIMESTAMP	datetime
XML	list of char

Table 4. How q datatypes convert to CCL datatypes

Q datatype	CCL datatype
boolean	BOOLEAN INTEGER

Q datatype	CCL datatype
byte date int minute month time second short	INTEGER LONG Time types are set to their q integer representation.
long	INTEGER (On overflow, if value is greater than MAX_INT, the value is set to NULL.) LONG
real float	FLOAT
char list of char symbol	STRING
datetime	TIMESTAMP
Other types	Conversion not supported

When a database subquery includes a statement using the SQL dialect of q (as opposed to regular q syntax), data is read into CCL from kdb+ as it would be from any database: each row returned by the SQL dialect of q is read into a row in Sybase CEP Engine with the same number of columns. When a database subquery includes a statement using simple q (not its SQL dialect), the results returned by the q statement are mapped as shown in the following table.

Table 5. Q value mapping to CCL rows and columns

Q	CCL
Single value (atom)	One column, one row.
Simple list	One column, multiple rows.
Dictionary	Two columns, multiple rows.

Q	CCL
Flip	Multiple columns, multiple rows (the same as from the SQL dialect of q).

Conversion Between CCL and SOAP

Converts a number of input columns from the CCL statement and produces a number of input columns in SOAP.

The SOAP Remote Procedure Call (RPC) plugin expects any number of input columns from the CCL statement and can produce any number of input columns. The column types are mapped to SOAP RPC types as follows.

CCL Type	XSI Type	Description
BLOB	xsd:base64Binary	Base64-encoded binary.
BOOLEAN	xsd:boolean	A boolean value, 1 or 0, true or false.
FLOAT	xsd:double or xsd:float (in the response only)	Signed floating point number.
INTEGER	xsd:int	32-bit signed integer.
INTERVAL	xsd:duration	Time durations in the format *PnYnMnDTnHnMnS.*
LONG	xsd:long	64-bit signed integer.
STRING	xsd:string	String of characters.
TIMESTAMP	xsd:dateTime	Date/time in the format: [-] *CCYY* - *MM* - *DD*T*hh* : *mm* : *ss* [Z |(+ | -)hh:mm].
XML	not applicable	Not supported.

Literals

Literals are fixed data values.

The terms literal and constant value are synonymous and refer to a fixed data value. For example, 'STOCK', 'SUDDEN ALERT', and '512' are all string literals; 1024 is a numeric literal. String literals are enclosed in single or double quotation marks to distinguish them from object names.

Neither BLOB nor XML data types have literals.

Boolean Literals

Boolean literals are True and False statements which are not sensitive to case.

Boolean literals are case insensitive. True, false, TRUE, FALSE, tRuE, fAlSe, true, False, truE, and falsE are all valid.

String Literals

String literals appear as a part of expressions. Provide commands for String literals.

String literals are also sometimes called character literals or text literals. When a string literal appears as part of AN expression in this documentation, it is indicated by the word TEXT. The syntax for single-line string literals is:

```
"character_string"
```

or

```
'character_string'
```

The syntax for multi-line string literals is:

```
[[character_string]]
```

In all cases, character_string is a combination of alphabetic characters, numeric characters, punctuation marks, spaces, and tabs.

- A single-line string literal must be enclosed in single (' ') or double (" ") quotes.
- Two adjacent quotes with no character string between represent an empty string.
- Newline characters (either CR or NL) can only be used with the multi-line string syntax.

Note that, to include a single quotation mark (or an apostrophe) in a text string that is already surrounded by single quotes, you must enter the inside quotation mark twice. For example, to put the apostrophe in the word that's, use two apostrophe/single-quote characters in a row:

```
'And that''s the truth.'
```

Similarly, to put a double quote inside a string delimited by double quotes, use a *pair* of double quotes inside the string, for example:

```
"He said ""No!"""
```

Sybase CEP Engine treats the doubled characters as single characters in this situation.

Some examples of valid string literals are:

```
'abc123'
'abc 123'
```

```
'It''s a good idea.'
"20030"
"abc123"
"abc 123"
"""What?"" he asked."
```

Internationalization impacts string literals. All the literals in the preceding list are 7-bit ASCII literals. But this:

```
'   123'
```

is also a literal.

Numeric Literals

Numeric literals are used to specify integers and floating-point numbers.

Integer Literals

You must use the integer notation to specify an integer whenever an integer appears in expressions, conditions, Sybase functions, and CCL statements described elsewhere.

The syntax of an integer literal is as follows:

```
[+|-]
integer
```

where *integer* refers to any whole number or zero.

Some valid integers are:

```
3
-45
+10023
```

Long Literals

LONG literals follow the same rules as INTEGER literals. To force a literal that can be either INTEGER or LONG into a LONG data type, add the letter "L" to the end of the literal.

For example, the following are valid LONG literals:

```
2147483648L
-2147483649L
-9223372036854775808L
0L
```

To ensure that an expression is evaluated as LONG rather than as INTEGER, make sure that the first constant is followed by an L, even if that specific value fits within the range of INTEGER data types.

Float Literals

A float literal is a floating point number, usually used to represent numbers that include a decimal point. Use the float literal syntax whenever an expression is described as type FLOAT elsewhere in this documentation.

The syntax of a float literal is as follows:

```
[+|-]
floating_point_number
[E[+|-]
exponent
]
```

where floating_point_number is a number that includes a decimal point.

The optional letter e or E indicates that the number is specified in scientific notation. The digits after the E specify the exponent. The exponent can range from approximately - 308 to +308.

Some valid float literals are:

```
1.234
-45.02
+10023.
3.
.024
-7.2e+22
```

Note that FLOAT values are accurate to approximately 16 or 17 significant digits.

Time Literals

Time literals are used to specify timestamps and intervals.

Timestamp Literals

The syntax of a Timestamp literal is as follows: **TIMESTAMP** '*YYYY-MM-DD* [*HH:MI*[*:SS*[*.FF*]]]'

Where:

- *YYYY-MM-DD* are numeric designations of the year, month, and day.
- *HH:MI* are numeric designations for hour and minute.
- *:SS* is a designation for seconds, used only if the hour and minute are specified.
- *.FF* is a designation for fractions of a second, using zero to six digits and only if seconds are specified.

Note that one or more blank spaces must be used to separate the date from the time specification.

Some valid timestamps are:

```
TIMESTAMP '2002-03-12 06:05:32.474003'
```

```
TIMESTAMP '2005-02-01'
TIMESTAMP '2003-11-30 15:04'
```

In some contexts, such as when putting row timestamps into CSV files, timestamps can be entered as a number of microseconds elapsed since midnight January 1, 1970. In this case, the numbers are treated as though they are relative to UTC, rather than local time. For example, if you use 1 as the timestamp, and your local time zone is Pacific Standard Time (eight hours behind UTC), the result is the following timestamp:

```
1969-12-31 16:00:00.000001
```

Interval Literals

Use either of two formats for an Interval literal. The first form is similar to that of Timestamp literals and is as follows:

```
INTERVAL '{[D [day[s]]][ ][HH:MI[:SS[.FF]]]}'
```

Where:

- *D* is the number of days. The space between the day specification and the hour and minute specification is optional.
- *HH:MI* are the hour and minute.
- *:SS* is the seconds, used only when hours and minutes are specified.
- *.FF* is fractions of a second, using zero to six digits and only if seconds are specified.

Here is an example of this syntax form:

```
INTERVAL '999 days 23:59:59.999999'
```

The second syntax form of Interval literals is as follows:

```
{W week[s]][ ][D  day[s]][ ][HH hour[s]][ ][MI  minute[s]][ ][SS[.FF]
second[s]][ ][NNN millisecond[s][ ][NNN  microsecond[s>
```

All components of the interval are optional, but you must include at least one (not counting spaces). Here is an example:

```
2 days 3 hours 4 minutes 5.6 seconds
```

Both forms of Interval literals require that the values in each component be in the proper range. For example, you will get an error if you enter 61 minutes; you must enter this value as 1 hour 1 minute.

Null Values

Null values are columns in a row which have no value.

If a column in a row has no value, then the column is said to be Null, or to contain Null. Use a Null when the actual value is not known or when a value would not be meaningful.

Null Values in Sybase CEP Functions

Nearly all predefined scalar functions return Null when given a Null argument.

Most aggregate functions ignore Nulls. For example, consider a query that averages the five values 500, NULL, NULL, NULL, and 1500. Such a query ignores the Nulls and calculates the average to be (500+1500)/2 = 1000.

Null Values and Comparison Conditions

To test for Nulls, use IS NULL and IS NOT NULL. Do not use a comparison operator to test whether a value is Null. Since Null represents an unknown value, the result is also unknown, and therefore the result of the comparison is Null.

```
WHERE x != NULL -- WRONG!
WHERE X IS NOT NULL -- RIGHT
```

Comments

Comments explain CCL statements and other Sybase CEP Engine components.

CCL statements and other Sybase CEP Engine components can be associated with explanatory comments. These comments do not affect the execution of a statement. Sybase supports two distinct types of comments: regular comments and documentation tags.

Regular comments:

- Can be inserted before or after any CCL clause or statement.
- Appear in the form of text demarcated with special characters, using any of the three methods shown in the following table:

	Begin Comment with	End Comment with	Example
Method 1:	Two hyphens (--)	New line	-- This is a comment line.
Method 2:	Two slashes (//)	New line	// This is a comment line.

	Begin Comment with	End Comment with	Example
Method 3:	A slash and asterisk (/*)	An asterisk and slash (*/)	/* This type of comment can be contained on one line, or can extend across multiple lines. */

- Cannot be nested.

Documentation tags can be inserted into a CCL module and appear as text or labels within Sybase CEP Studio. For more information about documentation tags, see CCL Documentation Tags.

CCL Object Names and Identifiers

CCL objects must have a name, the name of the object is represented with an identifier.

CCL objects must be named, and some objects are made up of parts that you can or must name. The latter includes columns in a data stream.

Every Sybase CEP object has a name. In a CCL statement, the name of an object is represented with an identifier. The following list of rules applies to identifiers:

- Identifiers have no maximum character length.
- The prefix C8_ is reserved for internal Sybase CEP Engine names.
- The Sybase CEP CCL language contains other words that have special meaning. In particular, do not use the names of Sybase CEP built-in functions for the names of objects.
- You must use only ASCII characters in identifiers.
- Identifiers must begin with an ASCII alphabetic character.
- Identifiers can contain only ASCII alphanumeric characters and the underscore (_). Thus, characters in an identifier can contain letters, digits, and _.
- Within a given scope, no two objects can have the same identifier.
- Identifiers are not case sensitive.
- Sybase interprets the following identifiers as being the same, so they cannot be used for different objects in the same scope:

  futures

  FUTURES
- Columns in the same data stream or named window cannot have the same identifier. However, columns in different data streams or named windows can have the same identifier.
- Names include not only CCL Query identifiers, but also things like project names, module names, adapter names, stream names, column names, aliases, variables, and the names of windows.
- Variables cannot be named with the same name as columns used by a query module.

- Identifiers cannot consist of CCL reserved words. See Reserved Words for the list of reserved words.

Column References in CCL Queries

Column references refer to columns in one of the query's data sources, defined in the FROM clause.

Column references within CCL clauses in the Query statement, Database statement, and Remote Procedure statement must refer to columns in one of the query's data sources, defined in the FROM clause. Column references in clauses of the Delete statement and Set Variable statement must refer to the data stream or named window specified in the ON or DELETE FROM clauses.

In cases where the column reference may refer to more than one entity, column references must explicitly state the entity to which the column belongs, using the syntax

```
stream-or-window-name
.
column-name
```

or

```
alias
.
column-name
```

where *stream-or-window-name* is the name of the data stream or window identified elsewhere in the statement, and *alias* is the alias assigned to a data stream, named window, database subquery, remote subquery, or CCL subquery by the AS subclause of the FROM clause.

One of these syntax forms for column references is required when:

- A query includes multiple data sources, and the column name is contained in more than one of these data sources.
- The same column in the same data source is referenced multiple times. In this case, every instance of the column reference must include a unique data source alias.
- The data source to which the column reference is referring is not a data stream or window.

Operators

The Sybase CEP is capable of utilizing a variety of operator components.

Unary and Binary Operators

Unary operator operates on one operand, binary operator operates on two.

The two general classes of operators are unary and binary.

A unary operator typically appears with its operand in this format:

```
operator operand
```

A binary operator appears with its operands in this format:

```
operand1 operator operand2
```

If an operator is given a NULL operand, the result is always NULL, with the following exceptions:

```
AND
OR
XOR
```

See Null Values in Expressions for rules on using NULLs with the AND, XOR, and OR operators.

Precedence of Operators

Precedence is the order in which Sybase CEP Engine evaluates different operators in the same expression.

When evaluating an expression containing multiple operators, Sybase CEP Engine evaluates operators with higher precedence before evaluating those with lower precedence. Sybase CEP Engine evaluates operators with equal precedence from left to right within an expression.

The following table lists the order of precedence among CCL operators from high to low. Operators listed on the same line have the same precedence.

Operator	Operation
+, - (as unary operators)	Unary plus, negation
^	Exponent
*, /, mod	Multiplication, division, modulo
+, - (as binary operators), ||	Addition, subtraction, concatenation
=, !=, <>, <, >, <=, >=	Comparison
LIKE, REGEXP_LIKE, IN, IS NULL, IS NOT NULL	Like
NOT	Logical negation
AND	Conjunction
OR, XOR	Disjunction

Note that the ^ operator is right associative. Thus a ^ b ^ c = a ^ (b ^ c), not (a ^ b) ^ c.

Precedence Example

In the following expression, multiplication has a higher precedence than addition, so Sybase CEP Engine first multiplies 30 by 5 and then adds the result to 10, so that the computed expression is 160:

```
10+30*5
```

Use parentheses in an expression to override operator precedence. Sybase CEP Engine evaluates expressions inside parentheses before evaluating those outside. In the following expression, the computed value is 200 because the sum of 10 and 30 is computed first.

```
(10+30)*5
```

Arithmetic CCL Operators

Arithmetic CCL operators are used to negate, add, subtract, multiply, or divide numeric values.

You can use an arithmetic operator with one or two arguments to negate, add, subtract, multiply, or divide numeric values. The arguments to the operator must resolve to numeric data types or to any type that can be implicitly converted to a numeric type (see Implicit Data Type Conversions for more information).

Unary arithmetic operators return the same data type as the argument. For binary arithmetic operators, Sybase determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that type.

The following table lists the CCL arithmetic operators.

Operator	Purpose
+, -	Positive, negative (unary) Addition, subtraction (binary)
*, /, mod	Multiplication, division, modulo (binary)
^	Exponent/power (binary)

Concatenation CCL Operator

Concatenation CCL operator manipulates character strings.

The following table describes the concatenation operator:

Operator	Purpose
‖, +	Concatenates character strings.

The result of concatenating two character strings is another character string.

Concatenation where one or more operands is Null results in Null.

Comparison CCL Operators

Comparison CCL operators conducts a comparison of TRUE, FALSE, or NULL.

Comparison operators compare one expression to another. The result of such a comparison can be TRUE, FALSE, or NULL. Comparison operators use the following syntax:

```
expression1 comparison_operator expression2
```

where expression1 and expression2 are a pair of numeric, Boolean, String, or Timestamp expressions and comparison_operator is one of the operators described in the following table:

Operator	Purpose
=	Equality, which can also be used in a multi-equality expression, such as A=B=C.
!=, <>	Inequality.
>, <	Greater-than and less-than .
>=, <=	Greater-than-or-equal-to and less-than-or-equal-to.

When comparing numeric expressions, Sybase uses numeric precedence to determine whether the condition compares INTEGER or FLOAT values.

Logical CCL Operators

A Logical CCL operator combines the results of two Boolean expressions or conditions to produce a single result, or inverts the result of a single condition.

The following table describes logical operators:

Operator	Purpose
NOT	Returns true if the following condition is false. Returns false if it is true.
AND	Returns true if both component conditions are true. Returns false if either is false.
OR	Returns true if either component condition is true. Returns false if both are false.
XOR	Returns true if one component condition is true and the other is false. Returns false if both components are true or both are false.

For details of how logical operators behave with NULL values, see Null Values in Expressions.

Row CCL Operators

Row CCL Operators performs functions on rows.

The square brackets [ ] operator allows you to perform functions on rows other than the current row in a stream or window. This operator uses the following syntax:

stream-or-window-name[*index*].*column*

where *stream-or-window-name* is the name of a data stream or named window and *column* indicates a column in the stream or window. *index* is an expression that can include literals, parameters, and/or operators, and that evaluates to an integer. This integer indicates the stream or window row, in relation to the current row or to the window's sort order. (Note that, in this case, the square brackets refer to actual CCL syntax, and do *not* indicate an optional CCL component.) For example:

```
MyNamedWindow[1].MyColumn
```

When used with a data stream, or with a window that is *not* sorted by largest or smallest value, [0] indicates the current row (as determined by timestamp and order of arrival), [1] indicates the row immediately previous to the current row, and so on.

When [ ] is used with windows that have a LARGEST or SMALLEST clause, index refers to the sorting order specified by the LARGEST or SMALLEST keywords and by their BY column reference subclause. For example, if a window TopTrades is defined to keep the 10 largest values as sorted by its Price column, then TopTrades[0].Price accesses the highest price retained by the window, TopTrades[1].Price refers to the second-highest price, and so on.

When [ ] is used with an unnamed window that is partitioned with the GROUP BY clause, or by any window partitioned by one or more PER clauses, index calculation is based on the partition to which the row belongs, not on the entire window.

The [ ] operator can be used in the WHERE selection condition or the SELECT list.

When used on data streams, the [ ] operator is semantically identical to the PREV() function. When used on windows, the [ ] operator is semantically identical to the LAST() function:

Like CCL Operators

LIKE operators are supported by WHERE clause expressions to match WHERE clause string expressions to stings that closely resemble each other.

WHERE clause expressions support the use of the LIKE and REGEXP_LIKE operators to match WHERE clause string expressions to strings that closely resemble each other but don't exactly match.

Operator	Syntax	Purpose
LIKE	compare_expresion LIKE pattern_match_expression	Matches the pattern specified in pattern_match_expression to the contents of compare_expression and returns a value of true or false. Both compare_expression and pattern_match_expression must evaluate to a STRING type. The LIKE operator returns a value of true if compare_expression matches pattern_match_expression, or false if it does not. compare_expression and pattern_match_expression can contain wild cards, where the percent sign (%) matches any length string, and the underscore (_) matches any single character. If you want to use % or _ as a literal in your pattern, instead of as a wild card, preface these characters with a backslash (_ or \%): for example, x_y. You must also preface the backslash character with a backslash (\\) if you want to use it as a literal in your pattern. Using a backslash character as the last character in the pattern, or before a character other than an underscore, percent sign, or backslash, generates an error.
REGEXP_LIKE	compare_expression REGEXP_LIKE POSIX_regular_expression	Matches the pattern specified in POSIX_regular_expression to the contents of compare_expression and returns a value of true if they match or false if they do not. Both compare_expression and POSIX_regular_expression must evaluate to a STRING type.

The following example matches any row with a value in the column StockName that contains "Corp" in any position.

```
INSERT INTO OutStream
SELECT Trades.StockName
FROM Trades
WHERE Trades.StockName LIKE "%Corp%"
```

IN operator

The IN operator asks a set membership question.

The syntax for the IN operator is:

```
expression1 IN (expression [, ...] )
```

It asks a set membership question on the expression list. If the value of expression1 is in the expression lists values, then the result is true. If expression1 is NULL, then the expression is TRUE if and only if there is a NULL in the expression list. If expression1 is not NULL, then a

NULL in the expression list does not force the result to be NULL. Rather, the NULL is ignored.

Expressions

The Sybase CEP is capable of utilizing a variety of expression components.

General CCL Expressions

An expression is a combination of one or more values, operators, and Sybase functions that evaluate to a value. An expression generally assumes the data type of its components.

The following simple expression evaluates to 15 and is type INTEGER (the same type as its components):

```
5*3
```

The following expression is an example of a more complex expression that uses both functions and operators. The expression adds two to the price of a stock, rounds it to the nearest hundredth, and then divides it by four:

```
ROUND(Stock.price+2.0, 2))/4.
```

You can use expressions in many places including:

- The select list of the SELECT clause.
- A condition of the WHERE clause or HAVING clause.
- The ON clause in a Delete statement.

Simple CCL Expressions

A simple CCL expression specifies a column, constant, or NULL.

- A column name can be specified by itself or with the name of the data stream of which it is a part. To specify both the column and data stream, use this format: data_stream_name.column_name.
- A constant can be a number or a text string.
- The literal NULL denotes a null value. However, it can only appear in SELECT and in the results of IF/THEN/ELSE and CASE/THEN. It is never in an expression, but is just the literal NULL as the expression.

Some valid simple expressions are:

```
stocks.volume
'this is a text string'
26
```

Compound CCL Expressions

A compound CCL expression is a combination of simple or compound expressions.

Numerical compound expressions can be used with arithmetic functions and parentheses can be used to change the order of precedence of the expression's components.

Some valid compound expressions are:

```
('T' || ',' || 'IBM')
LENGTH('NAME') * 10
SQRT(9.) + 1.
```

Sybase CEP Function Expressions

Sybase CEP functions that are predefined can be used as an expression.

Some valid predefined function expressions are:

```
LENGTH('TICK 4.03')
ROUND(1234.567*43.0, 2)
```

IF Expressions

IF expressions are conditions that can are used in an expression or are expressions.

You can use the IF-THEN-ELSE subexpression in an expression or as an expression by itself. The IF-THEN-ELSE expression uses the following syntax:

```
IF
condition
THEN
expression
[ELSE
expression
]
END [IF]
```

Note that the ELSE expression is optional. If none of the conditions of an IF-THEN-ELSE expression are met and no ELSE subexpression is specified, the output is NULL.

CCL also supports the following IF-THEN-ELSEIF-THEN syntax:

```
IF
condition
THEN
```

```
expression
ELSEIF
condition
THEN
expression
[ELSEIF...THEN...]
[...]
END [IF]
```

The effect of the IF-THEN-ELSEIF-THEN expression is similar to the CASE expression. Here is an example:

```
SELECT (IF Price<1000 THEN 1
  ELSEIF Price >= 1000 AND Price <1500 THEN 1.5
  ELSE 2
  END)
```

Specify the ELSEIF condition as one word. Sybase CEP Engine interprets the condition ELSEIF as an ELSE condition with a nested IF condition.

CASE expression

Used as an expression. If WHEN is true it will then perform the corresponding THEN expression.

You can use CASE-WHEN-THEN as an expression or in an expression. With this form, the WHEN-THEN clause can be repeated any number of times. Each WHEN expression is a Boolean that must be true in order to perform the corresponding THEN expression. However, the representation is order-dependent so that the first WHEN expression that evaluates to true at runtime is the only one that performs its corresponding THEN expression. If none of the WHEN expressions are true, the ELSE expression is performed, if present.

CASE { **WHEN** *condition* **THEN** *expression* } [...] [**ELSE** *expression*] **END** [**CASE**]

The following is an example of a SELECT clause using a CASE expression:

```
SELECT
CASE
WHEN price>2000.00 THEN 4
WHEN price>1000.00 THEN 3
WHEN price>500.00 THEN 2
ELSE 1
END
```

CCL Subqueries in Expressions

It is useful to use a value in an expression that gets computed by a separate continuous query.

It is often useful to use a value in an expression that gets computed by a separate continuous query. For example, the moving average of a stock price often represents a significant threshold for making decisions. A query that computes the average of the stock price in a window would produce a continuous stream of values representing the current moving average of stock price.

Subqueries can be included in the FROM clause and WHERE clause of the outer query and use similar syntax to a Query statement, but without an INSERT clause. For details about the syntax and usage of subclauses, see CCL Subqueries in the FROM Clause and CCL Subqueries in the WHERE Clause.

Null Values in Expressions

Null values do not always provide a result that is null, in a logical expression the result might be a non-Null.

In general, when any operand of an expression is Null, then the answer is Null. (Use the syntax IS NULL and IS NOT NULL to test whether an expression is Null or not Null.)

However, when a Null is an operand in a logical expression, the result still might be non-Null. The following table shows how logical operators AND, OR, XOR, and NOT behave with TRUE, FALSE, and Null operands:

x	y	x AND y	x OR y	NOT x	x XOR y
TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE	TRUE
TRUE	NULL	NULL	TRUE	FALSE	NULL
FALSE	TRUE	FALSE	TRUE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
FALSE	NULL	FALSE	NULL	TRUE	NULL
NULL	TRUE	NULL	TRUE	NULL	NULL
NULL	FALSE	FALSE	NULL	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL

The reason for this is that in some cases a subset of the expression is sufficient to determine the truth or falsehood of the expression. For example, when evaluating X AND Y, if the server sees

that X is false, then the server knows that X AND Y cannot be true, so the server simply knows that it can returns FALSE regardless of the value of Y. Similarly, if the expression is X OR Y, and X is true, then the server returns TRUE regardless of the value of Y.

Statements

A description of various CCL statements.

ATTACH ADAPTER statement

Attaches an input or output adapter to an data stream.

Syntax

```
ATTACH { INPUT | OUTPUT } ADAPTER name TYPE type TO STREAM
stream [PROPERTIES { prop = value } [, ...] ] ;
```

Table 6. Components

name	A name for this instance of the adapter.
type	The adapter type, identifying the adapter. See "Adapters Supplied by Sybase CEP" in the Sybase CEP *Integration Guide* for more information about valid adapter types.
stream	The name of the stream.
prop	The name of a property for the specified adapter type. See the section for the specific adapter type in the Sybase CEP *Integration Guide* for more information about the properties available for the adapter.
value	The value for the specified property.

Usage

The Attach Adapter statement attaches an input or output adapter to the specified data stream in the current query module or submodule. Adapters can also be attached using the Sybase CEP Studio interface (see the Sybase CEP *Studio Guide* for more information), which inserts the Attach Adapter statement for you. Input adapters can only be attached to input streams, but output adapters can be attached to all types of streams.

The optional property names must match the adapter property names from the adapter's **.adl** file. Adapter property names are not case sensitive, and allow the arbitrary insertions of spaces inside the name of the property.

For more information about adapter properties and Adapter Definition Language **.adl** files, see the appropriate adapter section in the Sybase CEP *Integration Guide* .

Restrictions

- The type must be a defined adapter type. See "Adapters Supplied by Sybase CEP" in the Sybase CEP *Integration Guide* for more information about adapter types.
- Adapter property names must match the properties from the adapter's **.adl** file.
- Input adapters can only be attached to input data streams.

See Also

- Create Stream Statement

Example

```
ATTACH INPUT ADAPTER AdapterReadReaders TYPE
ReadFromCsvFileAdapterType
  TO STREAM InStreamReaders
  PROPERTIES
  FILENAME = "examples/RfidAndSensorNetworks/ParentChildrenTracking/
     data/rfid-readers.csv",
  LOOPCOUNT = "1",
  USECURRENTTIMESTAMP = "Yes",
  TITLEROW = "TRUE";
```

CREATE FUNCTION statement

CREAT FUNTION enables users to create a user-defined function with the Sybase CEP Function Language.

Syntax

CREATE FUNCTION *fname* ([*pname type* [, ...]] **) RETURNS** *type* { statement [, ...] }**END FUNCTION ;**

Table 7. Components

fname	The name of the function.
pname	The name of a parameter.
type	A CCL data type, either for a specific parameter or the return value of the function.
statement	A CFL statement. See statement for syntax.

statement

{ create_variable_statement | assignment_statement | return_statement | break_statement | continue_statement } | { if_statement | case_statement | while_statement }

Table 8. Components

fname	The name of the function.
pname	The name of a parameter.
type	A CCL data type, either for a specific parameter or the return value of the function.
statement	A CFL statement. See statement for syntax.

Usage

The CCL Create Function statement creates a user-defined function (UDF). The function definition is valid only inside the query module in which it is defined, or in any other query module which imports it by using the CCL Import statement. In all other ways, CCL UDFs can be used according to the same rules, and subject to the same restrictions, as predefined Sybase CEP functions.

Restrictions

- CCL UDF definitions are valid only inside the current query module (.ccl file). UDF definitions in other files must be imported with the Import statement.

See Also

- Import Statement
- Sybase CEP Function Language

Example

The following example shows an **EARTHDISTANCE()** function definition that calculates the distance between two geographical points:

```
CREATE FUNCTION EARTHDISTANCE(
  Latitude1 FLOAT, Longitude1 FLOAT,
  Latitude2 FLOAT, Longitude2 FLOAT)
  RETURNS FLOAT;
  RETURN (180/pi()) * 60 * 1.1515 *
    ACOS(
      SIN(Latitude1*pi()/180) * SIN(Latitude2*pi()/180) +
      COS(Latitude1*pi()/180) * COS(Latitude2*pi()/180) *
      COS((Longitude1 - Longitude2)*pi()/180)
    );
END FUNCTION;
```

CREATE PARAMETER statement

Creates a parameter within the module or submodule.

Syntax

CREATE PARAMETER *data_type name* [= *value*] ;

Table 9. Components

data_type	The CCL data type of the parameter.
name	The name of the parameter.
value	A literal of the specified type.

Usage

The Create Parameter statement creates a parameter that can be used by other CCL statements in the module. You can also create parameters using the Sybase CEP Studio interface. For instructions, see the Sybase CEP *Studio Guide* .

An expression evaluating to a literal of the appropriate data type can be used to set the default value for the new parameter. Parameters created with this statement can be used in all the same ways as those created with Sybase CEP Studio, but do not display in Sybase CEP Studio.

When the parameter is used in CCL statements, it must be preceded by a $ character using the following syntax: $parameter-name.

Restrictions

- Parameters created with the Create Parameter statement are not displayed in Sybase CEP Studio.
- Variables of a data type that does not permit literal constants cannot be initialized with the Create Parameter statement.

Example

Here is an example of a Create Parameter statement, followed by a Query statement that uses the parameter.

```
CREATE PARAMETER INTEGER AcceptableThreshold = 10;
INSERT INTO StreamOut
SELECT *
FROM StreamIn
WHERE Quantity > $AcceptableThreshold;
```

CREATE SCHEMA statement

Defines a named schema that can be referenced later and reused by one or more queries in the module.

Syntax

```
CREATE SCHEMA name { (col_name type [, ...]) | INHERITS [FROM]
schema_name [, ...] [ (col_name type [, ...]) ] } ;
```

Table 10. Components

name	The name of the schema.
col_name	A column name.
type	The CCL data type of the column.
schema_name	The name of another schema.

Usage

The named schema can inherit the column names and data types of one or more named schemas, created with Create Schema statements elsewhere in the same module. If you include an INHERITS clause in the schema definition, the new schema definition uses the column names and data types of the specified schemas as its first (left-most) columns, in the order listed, and adds any additional columns after the inherited columns.

Restrictions

- Named schemas are recognized by all CCL statements within a module, but not by statements outside the module. However, external schema definitions can be imported with the Import statement.
- The schema name must be unique within the query module.
- All column names within the schema, including any column names inherited by the current schema from another named schema using the INHERITS clause, must be unique.

See Also

- SCHEMA

Example

The following example creates a schema named trade_schema with two columns named Symbol and Price. The columns have data types of STRING and FLOAT, respectively.

```
CREATE SCHEMA trade_schema (Symbol STRING, Price FLOAT);
```

This example uses a different method to create the same schema for trade_schema as in the previous example. The schema inherits the Symbol column definition from symbol_schema, and then defines an additional Price column.

```
CREATE SCHEMA symbol_schema (Symbol STRING);
CREATE SCHEMA trade_schema INHERITS FROM symbol_schema (Price
FLOAT);
```

CREATE STREAM statement

Creates a data stream attached to the current query module or submodule.

Syntax

```
CREATE [ INPUT | OUTPUT | LOCAL ] STREAM name [schema_clause]
[PROPERTIES prop_def [, ...] ] ;
```

Table 11. Components

name	The name of the stream.
schema_clause	The definition of the schema for the stream. See SCHEMA Clause for syntax and usage. Can be omitted if the schema will be created automatically, as described in INSERT Clause and SELECT Clause.
prop_def	A property definition. Not valid for local streams.

prop_def

```
{ GUARANTEED DELIVERY = {INHERIT | ENABLE | DISABLE} } |
{ GUARANTEED DELIVERY MAXIMUM QUEUE SIZE = size } |
{ GUARANTEED DELIVERY MAXIMUM AGE = age } | { MAXIMUM DELAY =
max_delay } | { OUT OF ORDER DELAY = delay } |
{ SYNCHRONIZATION = { INHERIT | IN ORDER | OUT OF ORDER | USE
SERVER TIMESTAMP } } | { FILTERCOLUMNS = " col_name [, ...]
" }
```

Components

size	The size of the guaranteed delivery queue, in number of rows.
age	The maximum permitted age of rows in the guaranteed delivery queue, as an interval.

max_delay	An interval specifying the maximum time Sybase CEP Engine waits before sequencing incoming rows.
delay	An interval specifying the maximum time rows can be late before Sybase CEP Engine discards them.
col_name	The name of a column to use for filtering.

Usage

If you do not specify a type of stream, Create Stream creates a local stream by default.

For input and output streams, you can assign property values affecting behavior. You can list such specifications in any order. The values for the properties can be specified with any CCL element that is evaluated at compile time, such as functions, operators, and literals, but cannot contain references to stream or window columns or to other elements whose value is determined at run time.

GUARANTEED DELIVERY	This option controls the Guaranteed Delivery feature, which guarantees that every row is received by its destination at least once, as long as the application components are running. In order to truly guarantee delivery, you must also enable persistence on the module to ensure that rows are delivered even after a software failure. The **INHERIT** keyword causes the stream to inherit the Guaranteed Delivery settings of its query module or submodule. This is the default setting of Guaranteed Delivery options for input and output streams. The **ENABLE** keyword enables Guaranteed Delivery for the data stream, even if Guaranteed Delivery is turned off for the query module or submodule as a whole. The **DISABLE** keyword disables Guaranteed Delivery for the data stream, even if the Guaranteed Delivery is turned on for the query module or submodule as a whole.
GUARANTEED DELIVERY MAXIMUM QUEUE SIZE	When Guaranteed Delivery is enabled for the stream this option specifies the maximum number of rows that should be held in the Guaranteed Delivery queue. Rows that exceed the allowed number are expired from the Guaranteed Delivery queue and their delivery is not guaranteed. The default setting for this option is 0, indicating that no limit is placed on the number of rows the Guaranteed Delivery queue can hold.

GUARANTEED DELIVERY MAXIMUM AGE	When Guaranteed Delivery is enabled for the stream this option specifies the maximum permitted age of rows in the Guaranteed Delivery queue. Rows that exceed the allowed age are expired from the Guaranteed Delivery queue and their delivery is not guaranteed. The default setting for this option is 0, indicating that no limit is placed on the age of rows in the Guaranteed Delivery queue.
MAXIMUM DELAY	The interval specified by this option allows Sybase CEP Server to correct synchronization problems among rows arriving in different input streams in a query module or submodule. Sybase retains incoming rows of all the input streams in the query module or submodule for the interval specified by **MAXIMUM DELAY** before sorting the order in which they arrive in their respective streams. Since different streams may have **MAXIMUM DELAY** set to different values, the largest setting of this field among the different streams attached to the query module or submodule is used to determine the overall **MAXIMUM DELAY** interval. The default **MAXIMUM DELAY** setting is one (1) second. **MAXIMUM DELAY** should not be set to 0. If the **SYNCHRONIZATION** property is set to **INHERIT**, the stream uses the **MAXIMUM DELAY** setting of the query module or submodule containing the stream, and any **MAXIMUM DELAY** setting specified directly in the stream properties is ignored.

OUT OF ORDER DELAY	The interval specified by this option allows Sybase CEP Server to correct synchronization problems in a given input stream. Sybase CEP Server retains incoming rows for the input stream for the specified interval before ordering them by their timestamp, and delivering them into the stream. This ensures that any time synchronization problems between Sybase CEP Server and outside data sources encountered in the specified time period are resolved before the information is fed into the stream. If a row with an earlier timestamp arrives in the stream after the interval specified by **OUT OF ORDER DELAY, it is** discarded. The default **OUT OF ORDER DELAY** setting is 0, indicating that rows are not held before being ordered by timestamp. The **OUT OF ORDER DELAY** property works in conjunction with the **SYNCHRONIZATION** property, as explained in the **SYNCHRONIZATION** option entry of this table. If the **SYNCHRONIZATION** property is set to **INHERIT**, the stream uses the **OUT OF ORDER DELAY** setting of the query module containing the stream, and any **OUT OF ORDER** setting specified directly in the stream properties is ignored.

	The **SYNCHRONIZATION** property works with **MAXIMUM DELAY** and **OUT OF ORDER DELAY** to correct synchronization problems in a given input stream. Setting **SYNCHRONIZATION** to **INHERIT** causes the input stream to inherit the **MAXIMUM DELAY**, **OUT OF ORDER DELAY** and **SYNCHRONIZATION** settings of the query module in which the stream is contained. **INHERIT** is the default synchronization setting. Setting **SYNCHRONIZATION** to **IN ORDER** causes Sybase CEP Server to discard any rows arriving in the stream that have an earlier timestamp than the row currently in the stream. When this setting is used, the **OUT OF ORDER DELAY** property must not be set. If **IN ORDER** is set and an **OUT OF ORDER DELAY** value is specified, Sybase CEP Engine issues a warning and ignores the **OUT OF ORDER DELAY** setting.
SYNCHRONIZATION	Setting **SYNCHRONIZATION** to **OUT OF ORDER** causes Sybase CEP Server to retain incoming rows for the input stream for the interval specified by the **OUT OF ORDER DELAY** property, before ordering them by their timestamp, and delivering them into the stream. This ensures that any time synchronization problems between Sybase CEP Server and outside data sources encountered in the specified time period are resolved before the information is fed into the stream. When this setting is used, the **OUT OF ORDER DELAY** property must be set to a value of 1 or higher. If **OUT OF ORDER** is used in conjunction with an **OUT OF ORDER DELAY** setting of 0, the stream behaves as though the **IN ORDER** synchronization setting were selected. Setting **SYNCHRONIZATION** to **USE SERVER TIMESTAMP** causes rows arriving in the stream to be assigned a timestamp by Sybase CEP Engine, based on the current time reflected by Sybase CEP Server. This timestamp overrides and replaces the timestamp set by the input adapter.

FILTERCOLUMNS	The optional FILTERCOLUMNS property, used with output streams, defines a filter based on one or more columns from the stream's schema definition. The filter allows subscribers to the stream to receive only rows that include specified values in the filter column(s) without needing to filter the values using the CCL query filtering clauses. The actual values that the subscriber should receive are passed as GET parameters in the stream URI, using either Sybase CEP Studio, or Sybase CEP SDKs. For example if you define a "FILTERCOLUMNS = 'Exchange, Symbol'" filter, you can then set the stream URI to receive only rows where the value in the Exchange column is NYSE and the value in the Symbol column is IBM. The filtering is performed by Sybase CEP Server before the stream's subscribers receive data from the stream. See the Sybase CEP *Studio Guide* for information on data filtering in Sybase CEP Studio, and the Sybase CEP *Integration Guide* for information about filtering using Sybase CEP SDKs.

Restrictions

- Stream property values cannot refer to stream or window columns, or to other CCL components that are evaluated at run time; however, the col_name used by FILTERCOLUMNS must refer to one of the stream's columns.
- You can only set Guaranteed Delivery properties for input and output streams.
- You can only set the **GUARANTEED DELIVERY MAXIMUM QUEUE SIZE** and **GUARANTEED DELIVERY MAXIMUM AGE** properties if the **GUARANTEED DELIVERY** option is set to **ENABLE**.
- **You can only set the MAXIMUM DELAY** and **OUT OF ORDER DELAY** options for input streams.
- You can only set the FILTERCOLUMNS property for output streams.

See Also

- Create Schema Statement
- SCHEMA

Example

```
CREATE OUTPUT STREAM Alerts
  SCHEMA 'alerts.ccs'
  PROPERTIES
  GUARANTEED DELIVERY = ENABLE,
  GUARANTEED DELIVERY MAXIMUM AGE = 10 MINUTES
  FILTERCOLUMNS="User, Host";
```

CREATE VARIABLE statement

Defines a variable within the scope of the module and, optionally, initializes the variable to a value.

Syntax

```
CREATE VARIABLE  data_type name [= value] ;
```

Table 12. Components

data_type	The CCL data type of the variable.
name	The name of the variable.
value	A literal of the specified type. If omitted, the initial value is Null.

Usage

Another version of this statement is part of the Sybase CEP Function Language, for use is creating user-defined functions. For more information, see Create Variable Statement.

Restrictions

- A variable name must be unique to the query module and has scope only within the module.
- Variables of a data type that does not permit literal constants cannot be initialized with the Create Variable statement.

See Also

- Set Variable Statement

Example

```
CREATE VARIABLE INTEGER message_count = 0;
```

CREATE WINDOW statement

Defines a named window that can be referenced later and used by one or more queries.

Syntax

```
CREATE [ PUBLIC | MASTER ] WINDOW win_name schema_clause
{keep_clause [, keep_clause] } | { MIRROR master_window }
[INSERT REMOVED [ROWS] INTO name ] [ properties_clause ]
```

Table 13. Components

win_name	The name of the window.
schema_clause	A definition of the schema for the window. See SCHEMA Clause for more information.
keep_clause	The policy defining how rows are kept in this window. See KEEP Clause for more information.
master_window	The name of the master window this window mirrors. See Shared Windows for more information.
name	The name of the stream or window where removed rows are published.
properties_clause	Definitions for index columns, filter columns, or filter values. See properties_clause for more information.

properties_clause

```
PROPERTIES [ INDEXCOLUMNS=" col_name [, ...] " ]
[ FILTERCOLUMNS=" col_name [, ...] " | [ FILTER="  value
[, ...] " ] [ FILTEREXPR=" expression " ] ]
```

Table 14. Components

col_name	The name of a column to be indexed (for a public window) or used as a filter (for a master window).
value	A filter value for a mirror window.
expression	A filter expression for a mirror window.

Usage

Use the Create Window statement to create a named window, a public window, or a shared window (either master or mirror).

The Create Window statement:

- Optionally defines the window as a public, master, or mirror window. Note that a public window can also be a master or mirror window, but a master window cannot also be a mirror window.
- Includes either one or two KEEP clauses, which define the conditions under which the window should retain rows (window policies are discussed in more detail in the description of the KEEP clause), or defines the window as a mirror of a master window. Note that a statement defining a mirror window cannot include a KEEP clause.

- Optionally includes a PROPERTIES clause defining index columns for a public window, filter columns for a master window, or filter values or a filter expression for a mirror window.
- Optionally includes an INSERT REMOVED clause that identifies a data stream or named window in the same query module as the current window. Any rows removed from the current window are published to the specified stream or window.

Restrictions

- Named windows are recognized by all CCL statements within a module, but not by statements outside the module.
- The window name must be unique within the query module.
- Because named windows are not permanently attached to any stream, they must be explicitly populated by other statements in order to contain information.
- A master window cannot also be a mirror window.
- A mirror window is read only; you cannot insert, update, or delete rows from it.
- A mirror window must be in a separate project from the master window it mirrors.
- The FIRST and LAST functions do not work with mirror windows.
- The schema defined for a mirror window must exactly match the schema of the master window it is mirroring.
- If you set accelerated playback on the project containing the master window, you must set accelerated playback to the same value on all projects containing windows mirroring that master window.
- If you set persistence but not guaranteed delivery on the project containing the master window, any windows mirroring that master may not be accurate after a restart of the master project.
- The maximum delay setting for the module containing the mirror window must be at least slightly larger than the maximum delay setting for the module containing the master window, assuming you are using message timestamp.

See Also

- Create Schema Statement
- Public Windows
- Shared Windows
- KEEP
- SCHEMA

Examples

```
CREATE WINDOW AllGuestsWindow SCHEMA (Name STRING, NetWorth FLOAT)
KEEP ALL;
```

```
CREATE WINDOW RecentReadingsWindow0 SCHEMA
```

```
   ReadingsSchema
KEEP 10 ROWS;

CREATE WINDOW RecentReadingsWindow1 SCHEMA
   'RfidReadingsSchema.ccs'
KEEP 10.5 MINUTES;

CREATE WINDOW RecentReadingsWindow3 SCHEMA
   'RfidReadingsSchema.ccs'
KEEP 10 MINUTES 30 SECONDS;

CREATE WINDOW RecentReadingsWindow4 SCHEMA
   'RfidReadingsSchema.ccs'
KEEP INTERVAL '00:10:30';

CREATE WINDOW TodayTrades
SCHEMA ( Symbol STRING, Shares FLOAT,
   Duration INTERVAL) KEEP FOR Duration;
```

The following example creates a public window:

```
CREATE SCHEMA Myschema (Symbol STRING, Price FLOAT, Volume INTEGER,
VWAP FLOAT);
CREATE PUBLIC WINDOW MyPubWindow SCHEMA Myschema
KEEP 1 HOUR
PROPERTIES INDEXCOLUMNS = "Symbol, Price";
```

The following example illustrates a basic master and mirror window relationship. ProjectA (in the Default workspace) defines and populates a master window named MyTradesWindow:

```
CREATE MASTER WINDOW MyTradesWindow
SCHEMA MySchema
KEEP 24 hours;

INSERT INTO MyTradesWindow
SELECT * FROM MyTradesStream;
```

ProjectB defines a mirror of MyTradesWindow and sends rows to OutStream based on rows in the mirror window:

```
CREATE WINDOW MyWindow
SCHEMA MySchema
MIRROR "/Stream/Default/ProjectA/MyTradesWindow";

INSERT INTO OutStream
SELECT
  Symbol, Avg(Price), Min(Price), Max(Price)
FROM
  MyWindow
GROUP BY
  Symbol;
```

The following example shows the use of filter columns. The master window definition specifies the filter columns Exchange and Symbol:

```
CREATE MASTER WINDOW MyTradesWindow
SCHEMA MySchema
KEEP 24 hours
PROPERTIES
  FilterColumns='Exchange,Symbol';

INSERT INTO MyTradesWindow
SELECT * FROM MyTradesStream;
```

The mirror window definition specifies filter values "NYSE" and "IBM." The mirror window will only receive rows from the master that have the value "NYSE" in the Exchange column and the value "IBM" in the Symbol column:

```
CREATE WINDOW MyIbmWindow
SCHEMA MySchema
MIRROR "/Stream/Default/ProjectA/MyTradesWindow"
PROPERTIES
  Filter='NYSE,IBM';

INSERT INTO OutIbmStream
SELECT
  Symbol, Avg(Price), Min(Price), Max(Price)
FROM
  MyIbmWindow;
```

Public Windows

Create Window statements is used to define a public named window, which enables you to perform a snapshot query of a public window with SQL queries from several locations.

Usage

You can use the Create Window statement to define a public named window. You can:

- Use a public window in CCL statements in all the same ways you can use a regular named window (see Create Window statement). A public window used in this way must be located in the same project as the CCL queries that use it.
- Perform a snapshot query of a public window with SQL queries, from one of several locations:
  - From Sybase CEP Studio. See "Debug Menu" in the Sybase CEP *Studio Guide* for more information. This method can only query public windows contained in the main module or submodules of the project currently being worked on.
  - From external applications using Sybase CEP SDKs or from the c8_client command-line utility. For more information, see the Sybase CEP *Integration Guide* .
  - From a database subquery inside a CCL FROM clause. This method of querying a public window treats the public window as if it were a table in an external database. The

public window being queried can be located in the same project as the database subquery, or in a different project (however, the Database statement cannot be used to publish data to a public window).

Before querying a public window with a database subquery, you must first create a database service for the project in which the public window is located in the file c8-services.xml. You can then use this service name in place of the database service name within the database subquery. See the Sybase CEP *Administration Guide* for more information about setting up services.

- Using the ReadFromDB and PollFromDB adapters, which are described in the Sybase CEP *Integration Guide* . As with the database subquery, the adapters treat the public window as a table in an external database, and you must therefore set up the window's project as a database service in the file c8-services.xml before using the adapters to connect to the public window.

Indexing

Index a public window in a similar manner to a relational database table. Define an index on a public window on a single column or on multiple columns. Indexing a public window column can significantly improve public window query performance when used with the following SQL operations, either individually or in combination with one another:

- column-name = value
- column-name < value
- column-name > value
- column-name <= value
- column-name >= value
- column-name >= value
- ORDER BY expression ASC

Public window indexes do not apply to:

- OR operator expressions.
- LIKE operator expressions.

Note that, while having a custom index can significantly improve execution time for public window queries, it may also negatively affect performance of your CCL project as a whole. Pay attention to the performance of your project when adding a public window index.

Warning! Sybase CEP Engine supports indexes on BLOB and XML columns, but Sybase does not recommend them since BLOB and XML value comparisons are resource-intensive and will negatively affect performance of your CCL project.

Timestamp Columns in Public Window Queries

All public window queries generate an initial Timestamp column, containing the row timestamp in addition to any columns specified in the query. You can refer to the results of this column in queries of the public window, as in the following example:

```
INSERT INTO OutStream
```

```
SELECT Timestamp
FROM MyPublicWindow;
```

See Also

- Create Schema Statement
- Create Window Statement
- KEEP
- SCHEMA
- SQL

Example

```
CREATE SCHEMA Myschema (Symbol STRING, Price FLOAT, Volume INTEGER,
VWAP FLOAT);
CREATE PUBLIC WINDOW MyPubWindow SCHEMA Myschema
KEEP 1 HOUR
PROPERTIES INDEXCOLUMNS = "Symbol, Price";
```

Shared Windows

Shared windows allow you to perform a snapshot query with live updates, to duplicate contents of one master window in other mirror windows in other projects, and to receive updates to the mirror windows as the master window changes.

Master
A master window is define just the same as any other named window. The optional PROPERTIES clause allows you to define filter columns that allow mirror windows to specify that they will only receive rows from the master with specific values in the filter columns. Only a single set of filter columns can be defined per master window.

Mirror
A mirror window is a read-only window that receives duplicate rows from the master window. When you define a mirror window you specify the master window with a name in the following form, where hostname identifies the computer running Sybase CEP Manager for the project containing the master window, and port is the port number used by that Manager: ccl://hostname:port/Stream/workspace/project_name/master_window_name

If the project containing the master window is running in the same Manager as the project containing the mirror window, you can use a relative name that omits "ccl://hostname:port".

The MIRROR clause replaces the KEEP clause in a mirror window definition; the window retention policy is defined for the master window. The optional PROPERTIES clause allows you to specify values for the filter columns defined in the PROPERTIES clause of the master window, which restricts the rows in the mirror to those with the specified filter values in the filter columns. You specify filter values in CSV format and list them in the same order as the

filter columns definition on the master window. Filter values can be omitted from a mirror window even if the master defines filter columns, but if filter values are included, a value must be specified for every filter column.

Use the FILTEREXPR property to specify an expression for filtering purposes, regardless of whether or not the master window is defined with the FILTERCOLUMNS property. If including this property, the mirror only receives rows for which the expression evaluates to True. The expression is a standard CCL Boolean scalar expression, such as the following:

```
FILTEREXPR="Symbol=IBM AND Price>10"
```

Note that you cannot include any of the following in the filter expression:

- Aggregator functions, such as SUM() or COUNT().
- Stateful operators like PREV().
- FIRST / LAST / INDEX operators.
- CCL variable references.
- GETTIMESTAMP().
- GET__COLUMNBYNAME().
- XMLPATTERNMATCH().
- Functions used within XMLTABLE().

However, you can include user-defined scalar functions and the zero-argument variant of GETTIMESTAMP().

Considerations

When a project that contains a mirror window starts up, the contents of its master window are copied to the mirror. This has a negative performance impact on the project containing the master window. How large an impact depends on the size of the window, the number of projects containing mirror windows starting at the same time, and how much optimization Sybase CEP Server can achieve (if both projects are running on the same server, Sybase CEP Server copies pointers instead of actual row contents). To reduce the performance impact, especially if your master and mirror projects are on separate servers, consider limiting the rows to be copied by using filter columns and also limiting the number of mirror windows that depend on the same master window.

Output from a mirror window does not begin until the entire initial copy has been received from the master window. After the initial copy, the mirror window changes in concert with the master window, receiving the same insertions, deletions, and updates within the constraints of any defined filters.

If the project containing a master window starts or restarts while a project containing a mirror of that master is running, the mirror replaces its contents with a fresh copy of the master window once that project has restarted.

Examples

The following example illustrates a basic master and mirror window relationship. ProjectA (in the Default workspace) defines and populates a master window named MyTradesWindow:

```
CREATE MASTER WINDOW MyTradesWindow
SCHEMA MySchema
KEEP 24 hours;

INSERT INTO MyTradesWindow
SELECT * FROM MyTradesStream;
```

ProjectB defines a mirror of MyTradesWindow and sends rows to OutStream based on rows in the mirror window:

```
CREATE WINDOW MyWindow
SCHEMA MySchema
MIRROR "/Stream/Default/ProjectA/MyTradesWindow";

INSERT INTO OutStream
SELECT
  Symbol, Avg(Price), Min(Price), Max(Price)
FROM
  MyWindow
GROUP BY
  Symbol;
```

The following example shows the use of filter columns. The master window definition specifies the filter columns Exchange and Symbol:

```
CREATE MASTER WINDOW MyTradesWindow
SCHEMA MySchema
KEEP 24 hours
PROPERTIES
  FilterColumns='Exchange,Symbol';

INSERT INTO MyTradesWindow
SELECT * FROM MyTradesStream;
```

The mirror window definition specifies filter values "NYSE" and "IBM." The mirror window will only receive rows from the master that have the value "NYSE" in the Exchange column and the value "IBM" in the Symbol column:

```
CREATE WINDOW MyIbmWindow
SCHEMA MySchema
MIRROR "/Stream/Default/ProjectA/MyTradesWindow"
PROPERTIES
  Filter='NYSE,IBM';

INSERT INTO OutIbmStream
SELECT
  Symbol, Avg(Price), Min(Price), Max(Price)
```

```
FROM
  MyIbmWindow;
```

DATABASE statement

Writes data to an external relational database.

Syntax

exec_clause select_clause from_clause [matching_clause] [on_clause] [where_clause] [group_by_clause] [having_clause] [order_by_clause] [limit_clause] [output_clause] ;

Table 15. Components

exec_clause	The destination for the published rows. See EXECUTE STATEMENT DATABASE Clause for more information. The statements in this clause are executed on the external database whenever this statement generates output.
select_clause	The select list specifying what to publish. See SELECT Clause for more information. All non-asterisk (*) items in this list must include an AS subclause.
from_clause	The data sources. See FROM Clause for more information.
matching_clause	A pattern-matching specification. See MATCHING Clause for more information.
on_clause	A join condition. See ON Clause for more information.
where_clause	A selection condition. See WHERE Clause for more information.
group_by_clause	A partitioning specification. See GROUP BY Clause for more information.
having_clause	A filter definition. See HAVING Clause for more information.
order_by_clause	A sequencing definition. See ORDER BY Clause for more information.
limit_clause	A limit on the number of rows to publish. See LIMIT Clause for more information.

output_clause	A synchronization specification. See OUTPUT Clause for more information.

Usage

The Database statement is a CCL statement that directly modifies data in an external relational database. You must have previously configured the database, as described in the Sybase CEP *Installation Guide* . Enter the Database statement directly into the CCL query module, and it executes SQL statements against the external database without going through an output adapter. In the case of kdb+ databases, the Database statement can also contain statements in Kx System's language q.

This statement is used only to pass SQL or q statements to databases, not to retrieve data from databases. Data retrieval from databases is handled by database subqueries (see Database Subquery for more information).

The Database statement is syntactically similar to the Query statement. It contains a number of clauses that create a query that subscribes to rows from a data stream, processes the incoming rows, and perform actions against the external database based on the results.

In addition to using the Database statement, you can also write to databases using the Write to DB output adapter, which passes SQL statements to an external database. For instructions on using the adapter, see the Sybase CEP *Integration Guide* .

Clause Processing Order

The main clauses within a Database statement are processed in the following order, which affects the query's output:

1. FROM clause (with MATCHING conditions, if any).
2. WHERE clause (with MATCHING conditions, if any).
3. SELECT clause (in conjunction with GROUP BY, if present).
4. HAVING clause.
5. OUTPUT clause.
6. ORDER BY clause.
7. LIMIT clause.
8. INSERT clause.

Restrictions

- Connection with the database listed in the EXECUTE STATEMENT DATABASE clause must first be configured, as described in the Sybase CEP *Installation Guide* , before you can use the Database statement.
- If the SELECT clause of the Database statement uses the "select all" expression (*) and the FROM clause of the statement lists multiple data sources, the data sources cannot share any column names.

- If the SELECT clause contains a list of column names, it must assign an output alias to each one, using the AS subclause.

See Also

- EXECUTE STATEMENT DATABASE
- FROM
- GROUP BY
- HAVING
- LIMIT
- MATCHING
- ON
- ORDER BY
- OUTPUT
- SELECT
- WHERE

Examples

The following simple example of a Database statement deletes rows from database table Table1, based on their match to values in the Column1 column of StreamIn.

```
EXECUTE STATEMENT DATABASE "ExternalDatabase"
  [[DELETE FROM Table1 WHERE Table1.Column1=?Column1]]
SELECT StreamIn.Column1 AS Column1
FROM StreamIn;
```

The following example inserts column values from the ActiveStrategies window and MyTrades streams, as well as a calculated timestamp into TradingDatabase.

```
CREATE WINDOW ActiveStrategies
SCHEMA (Symbol STRING, Strategy STRING)
KEEP LAST PER Symbol;

CREATE INPUT STREAM MyTrades
SCHEMA (Symbol STRING, Quantity INTEGER, Price FLOAT);

EXECUTE STATEMENT DATABASE "TradingDatabase"
   [[INSERT INTO Positions
      VALUES (?Symbol, ?Ts, ?Quantity, ?Price, ?Strategy) ]]
SELECT *, GETTIMESTAMP(MyTrades) as Ts
FROM MyTrades, ActiveStrategies
WHERE MyTrades.Symbol = ActiveStrategies.Symbol;
```

DELETE statement

Explicitly deleted specified rows from a named window.

Syntax

on_clause [when_clause] **DELETE FROM** *window_name*
[where_clause] ;

Table 16. Components

on_clause	A row arriving on the stream or window specified here triggers the deletion. See ON Clause: Trigger Syntax for more information.
when_clause	A condition limiting when the rows are deleted. See WHEN Clause for more information.
window_name	The name of the window from which rows are deleted.
where_clause	A filter limiting which rows are deleted. See WHERE Clause for more information.

See Also

- ON
- WHEN
- WHERE

Example

In this example the deletion is triggered from named window NW1, whenever a row arrives on StreamA. Every time the trigger occurs, all rows where the value in the ID column of the window is the same as the value in StreamA's ID column are deleted.

```
ON StreamA
DELETE FROM NW1
WHERE StreamA.ID = NW1.ID;
```

IMPORT statement

Imports schemas created with a 'Create Schema Statement' and functions created with a 'Created Function Statement' from an external file.

Syntax

IMPORT *file ;*

Table 17. Component

file	The name, including the absolute or relative path, of a file containing CCL statements .

Usage

The Import statement imports all schemas created with a Create Schema statement and functions created with a Create Function statement from the specified file into the current query module. Where the compiler searches for the file specified in this statement is controlled by compiler options. See "Compiler Options" in the Sybase CEP *Studio Guide* and "Importing" in the Sybase CEP *Integration Guide* for more information.

Restrictions

The Import statement does not import in-line schema definitions or other CCL content from the external .ccl file.

See Also

- Create Function Statement
- Create Schema Statement

Example

```
IMPORT "../../mydir/MyModule1.ccl";
```

INSERT VALUES statement

Creates one or more rows at specified times, and inserts them into a stream or window.

Syntax

insert_clause values_clause output_clause ;

Table 18. Components

insert_clause	The destination of the specified values. See INSERT Clause for more information.
values_clause	The values to insert. See VALUES Clause for more information.
output_clause	When the values should be inserted. See OUTPUT Clause for more information.

Usage

The Insert Values statement produces rows of data at specified times and publishes them to its destination data stream or window. This statement executes only at the times specified by the OUTPUT clause.

Important: In order for an Insert Values statement to produce output when you set synchronization to "Use message timestamp," a row must arrive on an input stream.

See Also

- INSERT
- VALUES
- OUTPUT

Example

This example inserts two rows of one column each into the Sites data stream when the project first starts running:

```
INSERT INTO Sites
VALUES ('New York'), ('London')
OUTPUT AT STARTUP;
```

QUERY statement

Subscribes to one or more data sources, processes the incoming rows, and publishes the results into a data stream or named window.

Syntax

insert_clause select_clause from_clause [matching_clause] [on_clause] [where_clause] [group_by_clause] [having_clause] [order_by_clause] [limit_clause] [output_clause] ;

Table 19. Components

insert_clause	The destination for the published rows. See INSERT Clause for more information.
select_clause	The select list specifying what to publish. See SELECT Clause for more information.
from_clause	The data sources. See FROM Clause for more information.
matching_clause	A pattern-matching specification. See MATCHING Clause for more information.

on_clause	A join condition. See ON Clause for more information.
where_clause	A selection condition. See WHERE Clause for more information.
group_by_clause	A partitioning specification. See GROUP BY Clause for more information.
having_clause	A filter definition. See HAVING Clause for more information.
order_by_clause	A sequencing definition. See ORDER BY Clause for more information.
limit_clause	A limit on the number of rows to publish. See LIMIT Clause for more information.
output_clause	A synchronization specification. See OUTPUT Clause for more information.

Clause Processing Order

The main clauses within a Query statement are processed in the following order, which affects the query's output:

1. FROM clause (with MATCHING conditions, if any).
2. WHERE clause (with MATCHING conditions, if any).
3. SELECT clause (in conjunction with GROUP BY, if present).
4. HAVING clause.
5. OUTPUT clause.
6. ORDER BY clause.
7. LIMIT clause.
8. INSERT clause.

See Also

- FROM
- GROUP BY
- HAVING
- INSERT
- LIMIT
- MATCHING
- ON
- ORDER BY
- LIMIT

- OUTPUT
- SELECT
- WHERE

Example

```
INSERT INTO HourlyStockVolume
SELECT Symbol, SUM(Volume)
FROM StockTrades KEEP EVERY 1 HOUR
GROUP BY Symbol,Buyer
OUTPUT EVERY 1 HOUR;
```

REMOTE PROCEDURE statement

Sends Remote Procedure calls (RPCs) to an external destination.

Syntax

```
exec_rem_proc_clause select_clause from_clause
[matching_clause] [on_clause] [where_clause]
[group_by_clause] [having_clause] [order_by_clause]
[limit_clause] [output_clause] ;
```

Table 20. Components

exec_rem_proc_clause	The remote procedure to execute when the other clauses generate output. See EXECUTE REMOTE PROCEDURE Clause for more information.
select_clause	The select list specifying what to publish. See SELECT Clause for more information. All non-asterisk items in this list must include an AS subclause.
from_clause	The data sources. See FROM Clause for more information.
matching_clause	A pattern-matching specification. See MATCHING Clause for more information.
on_clause	A join condition. See ON Clause for more information.
where_clause	A selection condition. See WHERE Clause for more information.

group_by_clause	A partitioning specification. See GROUP BY Clause for more information.
having_clause	A filter definition. See HAVING Clause for more information.
order_by_clause	A sequencing definition. See ORDER BY Clause for more information.
limit_clause	A limit on the number of rows to publish. See LIMIT Clause for more information.
output_clause	A synchronization specification. See OUTPUT Clause for more information.

Usage

The Remote Procedure statement executes a service defined in the file c8-services.xml. You must have previously configured the service as described in "Enabling Remote Procedure Calls" in the Sybase CEP *Installation Guide* . This statement is used only to execute procedures on the remote service, not to retrieve data. Data retrieval from remote services is handled by remote subqueries (see Remote Subquery for more information).

The Remote Procedure statement is syntactically similar to the Query statement. It contains a number of clauses that create a query that subscribes to rows from a data stream and then filters and processes the incoming rows. Whenever this process generates output rows, the specified RPC service is invoked and output is sent to the external destination.

Clause Processing Order

The main clauses within a Remote Procedure statement are processed in the following order, which affects the query's output.

1. FROM clause (with MATCHING conditions, if any).
2. WHERE clause (with MATCHING conditions, if any).
3. SELECT clause (in conjunction with GROUP BY, if present).
4. HAVING clause.
5. OUTPUT clause.
6. ORDER BY clause.
7. LIMIT clause.
8. INSERT clause.

Restrictions

- First configure the service name listed in the EXECUTE REMOTE PROCEDURE clause, as described in "Enabling Remote Procedure Calls" in the *Sybase CEP Installation Guide* , before using this statement.

- If the SELECT clause of the Remote Procedure statement uses the "select all" expression (*) and the FROM clause of the statement lists multiple data sources, the data sources cannot share any column names.
- If the SELECT clause contains a list of column names, it must assign an alias to each selected column.

See Also

- EXECUTE REMOTE PROCEDURE
- FROM
- GROUP BY
- HAVING
- LIMIT
- MATCHING
- ON
- ORDER BY
- OUTPUT
- SELECT
- WHERE

Example

```
EXECUTE REMOTE PROCEDURE "GetManagerApproval"
SELECT
   ManagerName AS Name,
   ManagerTitle AS Title,
   ManagerPhone AS Phone,
   EmployeeName AS EmployeeName,
   EmployeeTitle AS EmployeeTitle,
   TotalSum AS TotalSum
FROM OrdersWithManager
WHERE ManagerName IS NOT NULL AND
     (ManagerApprovalLimit IS NULL OR
      ManagerApprovalLimit < TotalSum);
```

SET VARIABLE statement

Changes the value of one or more variables.

Syntax

on_clause `[when_clause] set_clause ;`

Table 21. Components

on_clause	A row arriving on the data source specified here triggers the variable value change specified in the set_clause. See ON Clause: Trigger Syntax for more information.
when_clause	A condition that limits when the trigger occurs. See WHEN Clause for more information.
set_clause	The variables and values to be set when the trigger occurs. See SET Clause: Set Variable Statement Syntax for more information.

Restrictions

- The Set statement can only be used with variables created with the Create Variable statement.
- The expression used to set each variable's value must evaluate to the data type defined for the variable.

See Also

- Create Variable Statement
- ON
- SET
- WHEN

Example

In this example, the messages in StreamA are counted:

```
ON StreamA
SET message_count = message_count+1;
```

UPDATE WINDOW statement

Updates existing rows in a named window or, optionally, inserts new rows into the window if no matching rows are found to update.

Syntax

```
on_clause [when_clause] update_clause set_clause
[where_clause] [otherwise_insert_clause] ;
```

Table 22. Components

on_clause	A stream or window that acts as the trigger for the update. See ON Clause: Trigger Syntax for more information.
when_clause	A trigger condition. See WHEN Clause for more information.
update_clause	The destination stream or window. See UPDATE Clause for more information.
set_clause	The destination columns and associated values. See SET Clause: Window Syntax for more information.
where_clause	An update condition. See WHERE Clause for more information.
otherwise_insert_clause	A new row to insert into the destination if the condition in the where_clause isn't met. See OTHERWISE INSERT Clause for more information.

Usage

The ON clause specifies a data stream or named window. Arrival of rows in this stream or window, in combination with the WHEN clause condition (if any), triggers the update or insert process. Sybase CEP Engine processes each row arriving on the stream or window individually, performing an update for each row that meets the condition in the WHEN clause or performing an insert, if specified. This row-at-a-time processing means that rows arriving with the same timestamp can each potentially trigger an update to the same row. Thus the updated window is used when the next row is processed, even if that row has the same timestamp as the previous row.

All other affected CCL statements and clauses treat the window update as a row deletion followed by a row insertion, and respond accordingly. For example, named windows that include an INSERT REMOVED clause treat any updated rows (in their pre-update form) as removed rows. In some cases, this results in two published rows, such as when calculating aggregate values: one for the calculation after the updated row is removed and a second for the calculation after the updated row is inserted.

Clause Processing Order

The clauses within an Update Window statement are processed in the following order, which affects the statement's output:

- ON clause.
- WHEN clause.
- WHERE clause.

- SET clause.
- OTHERWISE INSERT clause.
- UPDATE clause.

See Also

- ON
- OTHERWISE INSERT
- SET
- UPDATE
- WHEN
- WHERE

Example

The following example updates the prices of stocks to the most current price. If no stock with the corresponding symbol is found, however, a new row recording the stock symbol and price publishes to the LastTrade window:

```
ON Trades AS T
UPDATE LastTrade AS L
SET T.Price AS Price
WHERE T.Symbol = L.Symbol
OTHERWISE INSERT *;
```

Clauses

The Sybase CEP is capable of utilizing a variety of clause components.

CACHE clause

Clears the cache when a row arrives in a stream, in either database subquery or remote subquery.

Syntax

```
CACHE { { CLEAR [ALL] ON source } | { MAXIMUM AGE age} |
{ MAXIMUM { MEMORY USAGE | SIZE } limit } } [, ...]
```

Table 23. Components

source	The name of a stream or window defined in the same query module as the subquery. A row arriving on this stream or window causes the cache to be cleared as specified.
age	An Interval expression specifying age.
limit	An Integer expression specifying the maximum memory or size.

Usage

CACHE is an optional clause of the database subquery or remote subquery. You can use this clause to clear all or part of the cache whenever a row arrives in the specified stream and/or to set retention preferences for the cache.

In the absence of a CACHE clause Sybase CEP Engine maintains and clears the cache based on the settings of the "CacheMaximumAge," "CacheMaximumMemoryUsage," and "CacheMaximumSize" configuration preferences, defined in the file c8-services.xml. The CACHE clause overrides these preferences and offers an additional level of control over the contents of the cache. This clause can include any of the following subclauses, in any order:

- The CLEAR ALL ON source subclause clears the entire cache whenever a row arrives in the specified stream or window.
- The CLEAR ON source subclause clears matching records in the cache when a row arrives in the specified stream or window. A cache record matches an incoming row if the column names and values of the incoming row match the cache record's. The source must indicate a stream or window with a schema identical to the one in the statement's SCHEMA clause.

The CLEAR ALL ON and CLEAR ON subclauses clear the cache independently of when queries are sent to the external database or remote service.

- The MAXIMUM AGE subclause determines whether caching takes place and specifies the time interval during which data is cached, which can be no longer than 100 years. If age is 0, then Sybase CEP Server does not cache data and all other caching information is ignored. If age is -1, Sybase CEP Server does not limit the cache based on age and you must set another caching preference (either in this clause or in the file c8-services.xml).
- The MAXIMUM MEMORY USAGE limit specifies a maximum number of bytes for the cache.
- The MAXIMUM SIZE limit subclause limits the size of the cache to the specified number of rows.

For more information about setting cache limits, see "Caching Data from an External Database, RPC Server, or Public Window" in *the Sybase CEP Installation Guide* .

See Also

- Database Subquery
- Remote Subquery
- FROM Clause: Database and Remote Subquery Syntax
- SCHEMA Clause

Example

Here is an example of a CACHE clause used inside a database subquery. This clause sets the cache to hold rows for 45 seconds, clears records that match the rows arriving in StreamXxx and clears all records from the cache when a row arrives in StreamYyy.

```
INSERT INTO StreamEnriched
SELECT
   StreamOrig.*,
   Db.*
FROM
   StreamOrig,
   (DATABASE "OracleTestDB" SCHEMA "db-result.cps"
         [[
            SELECT * FROM T WHERE ?StreamOrig.i=T.id AND ?
StreamOrig.s='abc'
         ]]
         CACHE
            MAXIMUM AGE 45 SECONDS
            CLEAR ON StreamXxx
            CLEAR ALL ON StreamYyy
   ) AS Db;
```

EXECUTE REMOTE PROCEDURE clause

Specifies the name of a predefined service that should be invoked by the Remote Procedure Statement and the values of the service parameters.

Syntax

EXECUTE REMOTE PROCEDURE " *service*"

Table 24. Component

service	The name of an RPC service defined in the file c8-services.xml.

ON ERROR extension

Use the ON ERROR extension to handle remote procedure execution errors that may cause the Sybase CEP Engine to stop project execution.

The syntax for the EXECUTE REMOTE PROCEDURE clause with the ON ERROR extension is:

```
EXECUTE REMOTE PROCEDURE "service"
   ON ERROR [error_insert_clause] CONTINUE
   SELECT…
```

Additional syntax related to the error_insert_clause:

```
error_insert_clause: INSERT INTO error_stream_name error_select_list
error_stream_name: Name of an output stream or local stream.
error_select_list: SELECT {error_expression} [,...]
error_expression: A CCL expression.
```

The error_expression can only contain:

- References to columns in the stream specified in the stream_clause.
- Operators, constant literals, and scalar functions.
- CCL parameters in the project. These should be prefaced with $.
- The ERROR_MESSAGE() built in.

The ON ERROR extension is optional. The error_insert_clause for the ON ERROR extension is also optional. If the ON ERROR extension is not provided and errors occur, the behavior of this clause is determined by the setting of the IgnoreErrors property in the c8-services.xml service configuration file. By default, this property is set to 'false', which means that if an error occurs, the query is aborted.

When the ON ERROR extension is provided without the error_insert_clause, errors found in the execution of this clause are ignored and CEP Engine continues to execute the project. Subsequent tuples arriving in any of the input streams of the from_clause continue to trigger execution of this clause. If these executions generate errors, they are also ignored by the CEP Engine.

When the ON ERROR extension is provided with the error_insert_clause, errors found in the execution of this clause are ignored and CEP Engine continues to execute the project. However a new tuple is also inserted into the errorstream (specified by error_stream_name). This tuple contains the fields selected in the error_select_list clause. The timestamp of this tuple is the same as the timestamp of the tuple in the from_clause that triggered the execution of this clause. These tuples are inserted in the errorstream in the same order as that of the original tuples in the from_clause that triggered the execution of this clause.

The ON ERROR extension cannot be used with both an EXECUTE STATEMENT DATABASE/EXECUTE REMOTE PROCEDURE clause and a REMOTE SUBQUERY/ PROCEDURE clause in the same CCL statement. This causes errors. To avoid errors, the CCL statements must be broken into separate statements using an intermediate local stream.

Set the MaxRetries parameter in the service definitions for Database to determine how many times Sybase CEP Engine should retry statement execution after experiencing errors.

The parameter in the service file is written as follows:

```
<Param Name="MaxRetries">some_number</Param>
```

where some_number is an integer between 0 and 255.

If the MaxRetries parameter is set to a value greater than 0, say N, then every time there is an error in an Execute Statement or Subquery statement that subscribes to this service definition, the Sybase CEP Engine automatically retries the execution of that statement up to a maximum of N number of times. If the execution of the statement fails for all the N number of times, an error tuple is inserted into the errorstream.

The default value for MaxRetries is 0, that is, by default there is no retry.

- Remote Procedure Statement
- FROM Clause: Database and Remote Subquery Syntax

Example

This query invokes procedure Temperature and passes values from the TempIn stream to the parameters City, ZipCode, and Time:

```
EXECUTE REMOTE PROCEDURE "Temperature"
SELECT 'CityName' AS City, 'ZipCode' AS ZipCode,
   GetTimestamp() AS Time
FROM TempIn
WHERE TempIn.ID = City;
```

The following example demonstrates how errors are logged when the ON ERROR extension and insert_clause are provided with this clause.

```
EXECUTE STATEMENT DATABASE "StockTradeDB"
[[
    INSERT INTO TradeTable VALUE(?Symbol, ?Price, ?Volume, ?Ts)
]]
ON ERROR
```

```
INSERT INTO DBWriteErrorStream
SELECT Symbol, Ts, ERROR_MESSAGE()
CONTINUE
SELECT
Symbol  as Symbol,
Price as Price,
Volume as Volume,
GETTIMESTAMP(InTrades) as Ts
FROM
InTrades KEEP EVERY 1 minute;
```

where the schema for DBWriteErrorStream is (Symbol String, Ts TimeStamp, ErrMsg String).

EXECUTE STATEMENT DATABASE clause

Specifies the name of an external relational database connection and the SQL or q statements that should be executed against the database.

Syntax

```
EXECUTE STATEMENT DATABASE "service" [[statements]]
   SELECT…
```

Table 25. Components

service	The name of a database service defined in the file c8-services.xml. For more information about configuring Sybase CEP Engine database services, see the *Sybase CEP Installation Guide* .
statements	The statements to be executed by the database software.

Usage

The EXECUTE STATEMENT DATABASE clause is the first clause of the Database statement. It indicates the database that is impacted when the CCL query in the Database statement generates output, and specifies the SQL or q statements that should be executed against the database in that event.

Sybase CEP Engine sends the text you specify (as statements) to the external database for execution after replacing references to CCL columns and parameters with the appropriate values. Since Sybase CEP Engine does not interpret the statements beyond replacing the column and parameter references, you may specify any statement supported by your database. However, Sybase CEP Engine ignores any return values. To retrieve data from an external database, use a database subquery.

To include the value of a CCL column in your database statement, preface the column name with a question mark (**?**stream_or_alias.column). To include the value of a CCL parameter, preface the parameter name by a question mark and a dollar sign (**?$**parameter).

ON ERROR extension

Use the ON ERROR extension to handle database errors that may cause Sybase CEP Engine to stop project execution.

The syntax for the EXECUTE STATEMENT DATABASE clause with the ON ERROR extention is:

```
EXECUTE STATEMENT DATABASE "service" [[statements]]
   ON ERROR [error_insert_clause] CONTINUE
   SELECT…
```

Additional syntax related to the error_insert_clause:

```
error_insert_clause: INSERT INTO error_stream_name error_select_list
error_stream_name: Name of an output stream or local stream.
error_select_list: SELECT {error_expression} [,...]
error_expression: A CCL expression.
```

The error_expression can only contain:

- References to columns in the stream specified in the stream_clause.
- Operators, constant literals, and scalar functions.
- CCL parameters in the project. These should be prefaced with $.
- The ERROR_MESSAGE() built in.

The ON ERROR extension is optional. The error_insert_clause for the ON ERROR extension is also optional. If the ON ERROR extension is not provided and errors occur, the behavior of this clause is determined by the setting of the IgnoreErrors property in the c8-services.xml service configuration file. By default, this property is set to 'false', which means that if an error occurs, the query is aborted.

Only the INSERT INTO statement or the EXECUTE STATEMENT DATABASE clauses are legal within the ON ERROR clause.

When the ON ERROR extension is provided without the error_insert_clause, errors found in the execution of this clause are ignored and CEP Engine continues to execute the project. Subsequent tuples arriving in any of the input streams of the from_clause continue to trigger execution of this clause. If these executions generate errors, they are also ignored by the CEP Engine.

When the ON ERROR extension is provided with the error_insert_clause, errors found in the execution of this clause are ignored and CEP Engine continues to execute the project. However a new tuple is also inserted into the errorstream (specified by error_stream_name). This tuple contains the fields selected in the error_select_list clause. The timestamp of this tuple is the same as the timestamp of the tuple in the from_clause that triggered the execution of this clause. These tuples are inserted in the errorstream in the same order as that of the original tuples in the from_clause that triggered the execution of this clause.

The ON ERROR extension cannot be used with both an EXECUTE STATEMENT DATABASE/EXECUTE REMOTE PROCEDURE clause and a REMOTE SUBQUERY/ PROCEDURE clause in the same CCL statement. This causes errors. To avoid errors, the CCL statements must be broken into separate statements using an intermediate local stream.

Set the MaxRetries parameter in the service definitions for Database to determine how many times Sybase CEP Engine should retry statement execution after experiencing errors. The parameter in the service file is written as follows:

```
<Param Name="MaxRetries">some_number</Param>
```

where some_number is an integer between 0 and 255.

If the MaxRetries parameter is set to a value greater than 0, say N, then every time there is an error in an Execute Statement or Subquery statement that subscribes to this service definition, the Sybase CEP Engine automatically retries the execution of that statement up to a maximum of N number of times. If the execution of the statement fails for all the N number of times, an error tuple is inserted into the errorstream. The default value for MaxRetries is 0, therefore there is no retry.

Kdb+ and q

EXECUTE STATEMENT DATABASE clauses that modify kdb+ databases can contain statements in q language.

Important: The q statement cannot contain newline characters. Everything between the brackets ([[ ]]) must be entered on a single line.

When you pass values from the Database statement's data source to the q statements in the EXECUTE STATEMENT DATABASE clause, you must enter the values as a space-separated list that appears immediately after the parameterized q statement. Sybase CEP Engine passed the values to the final q statement as a single general (multi-type) list. This list takes the place of the right side of the q statement (the part of the statement after the operator). For example, consider the following q statement:

```
5+(1;2;3)
```

To pass column values for the three parameters, you would write it like this:

```
5+ ?MyStream.IntColumn1 ?MyStream.IntColumn2 ?MyStream.IntColumn3
```

If you pass a list of parameters to the q (or Q-SQL) statement, the final statement must be written in the brackets to accommodate the parameters. For example:

```
{[params] select Price, Volume from trades where StockSym = `
$params[0],
   Volume > `$params[1]} ?ParamStream.StockSym ?ParamStream.Volume
```

Sybase CEP Engine automatically converts CCL datatypes to a limited set of q datatypes. To convert a CCL datatype to another kdb+ type, you must explicitly cast the CCL type as the

desired kdb+ type in your q statement. For example, the following Database statement casts the values from four columns to datetime, symbol, real, and short in q:

```
EXECUTE STATEMENT DATABASE "MyKDBService"
[[`trades insert (`datetime ; ` ; `real ; `short) $ ?Ts ?StockSym ?
Price ?Volume]]
SELECT GETTIMESTAMP (QueryIn) AS Ts,
       StockSym AS StockSym,
       Price AS Price,
       Volume AS Volume
FROM QueryIn;
```

Note: Specify all types for all of your parameters, even to cast a single value.

Examples

This query updates a table called MyTable in a database called MyDatabase. The Symbol and Price columns (aliased as CurSymbol and CurPrice) are selected from a Sybase CEP stream called StreamIn. The SQL query matches symbols in the Symbol column of MyTable with those of the Symbol column in StreamIn, and records the corresponding price from the Price column in StreamIn to a column called Price in MyTable.

```
EXECUTE STATEMENT DATABASE "MyDatabase"
  [[UPDATE MyTable
    SET MyTable.Price = ?CurPrice
    WHERE MyTable.Symbol=?CurSymbol]]
SELECT
  StreamIn.Symbol AS CurSymbol, StreamIn.Price AS CurPrice
FROM StreamIn;
```

This example inserts column values from the ActiveStrategies window and MyTrades streams, as well as a calculated timestamp into TradingDatabase:

```
CREATE WINDOW ActiveStrategies
SCHEMA (Symbol STRING, Strategy STRING)
KEEP LAST PER Symbol;

CREATE INPUT STREAM MyTrades
SCHEMA (Symbol STRING, Quantity INTEGER, Price FLOAT);

EXECUTE STATEMENT DATABASE "TradingDatabase"
   [[INSERT INTO Positions
      VALUES (?Symbol, ?Ts, ?Quantity, ?Price, ?Strategy) ]]
SELECT *, GETTIMESTAMP(MyTrades) as Ts
FROM MyTrades, ActiveStrategies
WHERE MyTrades.Symbol = ActiveStrategies.Symbol;
```

This example demonstrates how errors are logged when the ON ERROR extension and insert_clause are provided with this clause:

```
Example:
EXECUTE STATEMENT DATABASE "StockTradeDB"
[[
```

```
    INSERT INTO TradeTable VALUE(?Symbol, ?Price, ?Volume, ?Ts)
]]
ON ERROR
INSERT INTO DBWriteErrorStream
SELECT Symbol, Ts, ERROR_MESSAGE()
CONTINUE
SELECT
Symbol  as Symbol,
Price as Price,
Volume as Volume,
GETTIMESTAMP(InTrades) as Ts
FROM
InTrades KEEP EVERY 1 minute;
```

where the schema for DBWriteErrorStream is (Symbol String, Ts TimeStamp, ErrMsg String).

FROM clause

Used for single-source queries, used for joins that use the JOIN syntax, and it is used for database subqueries and to remote subqueries.

The **FROM** clause, part of a Query Statement, Database Statement, or Remote Procedure Statement, has three syntax variations:

- The FROM Clause: Comma-Separated Syntax: Used for single-source queries, and contains joins that use the comma-separated syntax and queries that use the MATCHING clause to detect patterns.
- The FROM Clause: Join Syntax: Used for joins that use the **JOIN** syntax.
- The FROM Clause: Database and Remote Subquery Syntax: Used for database subqueries and remote subqueries.

FROM clause: Comma-separated syntax

Specifies one or more data sources in a Query statment, Database statement, or Remote Statement.

Syntax

```
FROM { stream [ [AS] alias] | stream [ [AS] alias] keep_clause
[keep_clause] | window_name [ [AS] alias] | ( nested_join
[on_clause] ) |xmltable_exp |subquery } [, ...]
```

Table 26. Components

stream	The name of a data stream.

alias	An alias for the stream or window.
keep_clause	The policy that specifies how rows are maintained in the window. See KEEP Clause for more information.
window_name	The name of a window.
nested_join	A nested join. See FROM Clause: Join Syntax for more information.
on_clause	The join condition for the nested join. See ON Clause: Join Syntax for more information.
xmltable_exp	Produces an XML table. See XMLTABLE Expressions in the FROM Clause for more information.
subquery	A subquery. See CCL Subqueries in the FROM Clause for more information.

Usage

Use this variation of the FROM clause for:

- Single-source queries.
- Inner joins that use the comma-separated syntax. An inner join publishes all possible combinations of rows from the intersection of two or more specified data sources, according to the limitations of the selection condition (if present).
- Queries that use the MATCHING clause to detect patterns.

Any column or data source references in the statement's other clauses must be to one of the data sources named in this clause.

A nested join data source can include an ON subclause that establishes the nested join's selection condition. For more information about when the ON is required and when it is optional, see FROM Clause: Join Syntax.

Windows in the From Clause

Query statement data sources can include named or unnamed windows:

- Create named windows with a separate Create Window statement. These windows exist independently of the query and can simply be specified by name in the FROM clause, with or without an optional alias.
- Create unnamed windows within the FROM clause itself with one or two KEEP clauses, specifying one or two window policies. Window policies define how the window maintains state (retains rows). An unnamed window derives its schema from the stream to which it is linked.

Joins Specified with Comma-Separated Syntax

The comma-separated FROM clause can contain multiple data sources connected with an inner join. The multiple sources are separated by commas. An optional WHERE clause creates the selection condition for the join (the ON clause is not used to create a selection condition for comma-separated joins). In the absence of a WHERE clause, the join selects all values from all selected data source columns. This join syntax can include more than two data sources.

Pattern-Matching Syntax

Use the comma-separated FROM clause in conjunction with the MATCHING clause to specify the data sources that should be monitored for a specified pattern. When used in pattern detection, all data sources specified in the FROM clause must be data streams and must include all streams identified as events in the MATCHING clause. An optional ON clause creates the matching join condition for the query.

Aliases

Data sources specified in the FROM clause can include an alias, which can be used to refer to the data source anywhere in the query where the data source name would otherwise be used.

The use of an alias is required when the same column in the same data source is used more than once by the query's SELECT clause, or when a data stream is listed more than once in the MATCHING clause. In such cases, each use must also be listed as a separate data source in the FROM clause, and a unique alias assigned to each instance.

Restrictions

The following restrictions apply when you use the comma-separated FROM syntax to create a join:

- The list of data sources in the join can include only one data stream expression.
- No ON clause can be used to set a condition for the join, except in the case of nested joins using the JOIN word syntax.

The following restrictions apply when using the comma-separated FROM syntax in conjunction with the MATCHING clause:

- The list of data sources can include only data streams.
- If an ON clause is used, it must establish a relationship between all the data sources included in the FROM clause.
- The list of data sources must include all data sources specified in the MATCHING clause, and cannot include any other data sources.

See Also

- Database Statement
- Query Statement

- Remote Procedure Statement
- KEEP
- MATCHING
- ON

Example

The FROM clause in the following example creates a two-window inner join.

```
INSERT INTO OutStream
SELECT S1.Symbol, S1.Average, S2.Average
FROM Stream1 AS S1 KEEP 10 ROWS, Stream2 AS S2 KEEP 5 MINUTES
WHERE S1.Symbol = S2.Symbol;
```

FROM clause: Database and remote subquery syntax

Retrieves data from a relational database, public window, or another remote source, directly into a CCL query.

Syntax

FROM { { db_sub> | rem_sub>} { , | [**LEFT OUTER**] **JOIN** } *stream* [[**AS**] *alias*] } | { *stream* [[**AS**] *alias*] { , | [**RIGHT OUTER**] **JOIN**} { db_sub> | rem_sub>} }

Table 27. Components

db_sub	A database subquery. See Database Subquery for more information.
rem_sub	A remote subquery. See Remote Subquery for more information.
stream	The name of a data stream.
alias	An alias for the stream.

Usage

Use this specialized form of the FROM clause to send requests for information directly to an external relational database, public window, or other external source. This data is retrieved directly into the CCL query. Two types of specialized subqueries are used for this purpose in the FROM clause.

Note: These subqueries are distinct from CCL subqueries, which are described in CCL Subqueries in the FROM Clause.

A database subquery sends SQL statements to an external relational database or Sybase CEP Engine project containing a public window and retrieves rows from tables in the database, or public windows contained in the project. A remote subquery sends Remote Function Calls

(RFCs) using one of a variety of available protocols to another external non-CCL data source, and retrieves the resulting data into a CCL query.

Before using either the database subquery or the remote subquery, configure Sybase CEP Engine for communication with the external database, project containing the public window, or other data source. For instructions on configuring Sybase CEP Engine for use with external services, see the *Sybase CEP Installation Guide* .

Both the database subquery and the remote subquery perform only information retrieval; neither of these clauses modifies the external data sources. The database and remote subqueries each have a CCL statement counterpart, however, that is used to modify external data sources. The Database statement writes to tables in an external relational database (but not to a CCL public window). The Remote Procedure statement executes procedures on external services.

The database subquery or remote subquery is always joined in the FROM clause to a single data stream. The join between the database or remote subquery and the stream can be an inner join or a right or left outer join, but not a full join.

JOIN or comma (,)	Specifies an inner join	All possible combinations of rows from the intersection of the data stream and the database subquery or remote subquery (limited by the selection condition, if specified) are published.
LEFT OUTER JOIN	Specifies a left outer join (data stream must be on left)	All possible combinations of rows from the intersection of the data stream and the database subquery or remote subquery (limited by the selection condition, if specified) are published. All other rows from the data stream are published as well. Unmatched columns in the database subquery or remote subquery publish a value of Null.
RIGHT OUTER JOIN	Specifies a right outer join (data stream must be on right)	As for LEFT OUTER JOIN

The ON clause is not used with this type of join.

Queries that use the database subquery or remote subquery variety of the FROM clause execute only when a row arrives in the data stream.

For more detailed information about the database subquery and remote subquery usage see Database Subquery and Remote Subquery.

DATABASE subquery

Retrieves data from relational database tables or from CCL public windows into a Query statement, database statement, or remote procedure statement and can also specify cache settings.

Syntax

```
( DATABASE "service"  schema_clause [[statements]]
[cache_clause] ) [AS] alias
```

Important: The double brackets shown here are part of the syntax itself, not an indication of an optional component.

Table 28. Components

service	The name of the database or public window service as configured in the file c8-services.xml. For more information about configuring services, see the *Sybase CEP Installation Guide .*
schema_clause	A schema definition for the retrieved data. See SCHEMA Clause for more information.
statements	Statements to retrieve data from the specified service.
cache_clause	Specifications for clearing the cache. See CACHE Clause for more information.
alias	An alias for the subquery.

Usage

You use a database subquery in a FROM Clause: Database and Remote Subquery Syntax. This specialized subquery contains statements that are passed to an external relational database server or to a CCL public window. The statements select data from the external database tables or public windows contained in the project specified by the service entry, and allow you to access the data in a CCL query. Before using a database subquery, perform the following steps:

- Configure a connection between an external relational database and Sybase CEP Engine, as explained in the *Sybase CEP Installation Guide* , or, in the case of public windows, create the window, using the Create Window statement.
- Set up a service entry for the external database containing the tables, or the project containing the public windows you want to query in the file c8-services.xml. For more

information about configuring Sybase CEP Engine database and public window services, see the *Sybase CEP Installation Guide* .

The schema you specify must provide a column name and data type for every column you want to retrieve from the database or public window. The column names in the schema do not have to match the column names in the database or public window, but the data type for each column must be a CCL data type that is compatible with the data type of the corresponding column in the database or public window. In cases where data types used by the external database server do not perfectly match the data types used by Sybase CEP Server, Sybase CEP Server automatically converts the values. See Conversion Between CCL and ODBC and Oracle and Conversion Between CCL and Q for more information.

Sybase CEP Engine sends the text you specify (as statements) to the external database or project containing the public window for execution after replacing references to CCL columns and parameters with the appropriate values. Sybase CEP Engine does not interpret the statements beyond replacing the column and parameter references, but you cannot include statements that modify the database or public window. For information about writing to or deleting from an external database, see Database Statement. If you are querying a public window, your statements must be made up of SQL statements. See SQL for more information.

To include the value of a column from the data stream in your statement, preface the column name with a question mark (?col_name_or_alias). To include the value of a CCL parameter, preface the parameter name by a question mark and a dollar sign (?$parameter).

Access the retrieved data using the alias and column names previously specified.

Within the FROM clause, always define the database subquery in a join with a single data stream. This join can be either an inner or an outer join. Define inner joins with either a comma-separated syntax, or with the use of the JOIN keyword. When you use the comma-separated syntax, you define the join condition in a WHERE clause. When you use the JOIN keyword syntax, however, you cannot use the CCL ON clause with non-database subquery joins. Instead, define the join condition with an SQL WHERE clause inside the SQL statements.

ON ERROR extension

Use the ON ERROR extension to handle database errors that may cause Sybase CEP Engine to stop subquery execution.

The syntax for the DATABASE subquery clause with the ON ERROR extension is:

```
( DATABASE "service"  schema_clause [[statements]]
[cache_clause] ON ERROR [error_insert_clause] CONTINUE ) [AS] alias
```

Additional syntax related to the error_insert_clause:

```
error_insert_clause: INSERT INTO error_stream_name error_select_list
error_stream_name: Name of an output stream or local stream.
error_select_list: SELECT {error_expression} [,...]
error_expression: A CCL expression.
```

The error_expression can only contain:

- References to columns in the stream specified in the stream_clause.
- Operators, constant literals, and scalar functions.
- CCL parameters in the project. These should be prefaced with $.
- The ERROR_MESSAGE() built in.

The ON ERROR extension is optional. The error_insert_clause for the ON ERROR extension is also optional. If the ON ERROR extension is not provided and errors occur, the behavior of this clause is determined by the setting of the IgnoreErrors property in the c8-services.xml service configuration file. By default, this property is set to 'false', which means that if an error occurs, the query is aborted.

When the ON ERROR extension is provided without the error_insert_clause and there is an error executing the subquery, the error is ignored by the Sybase CEP Engine and the subquery execution is considered as having returned no results. Subsequent tuples arriving in the joining stream continue to trigger execution of the subquery. If these executions generate errors then these errors are also ignored by the Sybase CEP Engine.

When the ON ERROR extension is provided with the error_insert_clause and there is an error executing the subquery, a new tuple is inserted into the errorstream (specified by error_stream_name). This tuple contains values of the fields selected in the error_select_list clause. The timestamp of this tuple is the same as the timestamp of the tuple in the joining stream that triggered the execution of this subquery. These tuples are inserted in the errorstream in the same order as that of the original tuples in the joining stream that triggered the execution of this subquery. When there is no error, no tuples are inserted in the errorstream. The responses containing errors are not cached. If there is a subsequent tuple with the same binding key, it triggers another execution of the database subquery.

The ON ERROR extension cannot be used with both an EXECUTE STATEMENT DATABASE/EXECUTE REMOTE PROCEDURE clause and a REMOTE SUBQUERY/ PROCEDURE clause in the same CCL statement. This causes errors. To avoid errors, the CCL statements must be broken into separate statements using an intermediate local stream.

Set the MaxRetries parameter in the service definitions for Database to determine how many times Sybase CEP Engine should retry statement execution after experiencing errors.

The parameter in the service file is written as follows:

```
<Param Name="MaxRetries">some_number</Param>
```

where some_number is an integer between 0 and 255.

If the MaxRetries parameter is set to a value greater than 0, say N, then every time there is an error in an Execute Statement or Subquery statement that subscribes to this service definition, the CEP Engine automatically retries the execution of that statement up to a maximum of N number of times. If the execution of the statement fails for all the N number of times, an error tuple is inserted into the errorstream.

The default value for MaxRetries is 0, that is, by default there is no retry.

CCL References within SQL Statements

The statement within the database subquery can refer to column names in the data stream to which the database subquery is joined, as well as to CCL module parameters. CCL data stream column references within the SQL query use the following syntax to distinguish them from internal database or public window columns:

```
?
data-stream-name
.
column-name
```

OR

```
?
data-stream-alias
.
column-name
```

CCL module parameters can be referenced within the SQL query using the following syntax:

```
?$
parameter-name
```

When the SQL statements inside your database subquery refer to a public window, the references must use the following syntax:

Restrictions

Queries that include database subqueries are subject to the following restrictions:

- A query that includes a database subquery must have exactly one other data source.
- The non-database, non-public window data source must not be windowed.
- In the statement, all names of stream columns must be prefixed with the data stream name or alias.
- A left outer join that joins a database subquery to a data stream must list the data stream on the left; a right outer join that joins a database subquery to a data stream must list the data stream on the right.
- No full outer joins are allowed with database subqueries.
- For a discussion restrictions specific to kdb+, see kdb+_and_q.
- The use of the ON clause is not allowed with database subqueries.
- If you are querying a public window contained in a submodule, and the public window reference is the last component of your SQL query, you must separate the closing bracket (]) of the window reference from the closing double bracket (]]) of the statements by one or more spaces. A triple bracket (]]]) generates an error.

- If you are querying an external database, the exact syntax allowed between the double brackets is controlled by the target database. In particular, be aware that the target database may not support comments.
- Sybase CEP does not support stored procedures in database subqueries that connect to an external database using the Oracle native driver.

See Also

- Create Schema Statement
- Database Statement
- CACHE
- SCHEMA
- SQL

Examples

This example executes a database subquery on the MyDB database server. The file valuation.ccs contains the schema to be used with the imported data. The data retrieved from the database is then references as the Shares_outstanding column.

```
INSERT INTO Valuation
SELECT T.Symbol, T.Price, S.Shares_outstanding,
       T.Price * S.Shares_outstanding
FROM TradeStream T,
    (DATABASE "MyDB" SCHEMA 'valuation.ccs'
    [[SELECT * FROM Stocks WHERE ?T.Symbol=Stocks.Symbol ]]) AS S
```

This example queries a public window named GroupsWindow in the Groups submodule of the project's main module.

```
INSERT INTO WinOut
SELECT pw.GroupID, pw.GroupName, pw.SeverityAssessmentLevel,
       pw.PrincipleAffiliation
FROM MyStream,
     (DATABASE "SecurityThreatGroups"
      SCHEMA (GroupID STRING, GroupName STRING,
              SeverityAssessmentLevel STRING, PrincipleAffiliation
STRING)
      [[SELECT GroupID, GroupName, SeverityAssessmentLevel,
               PrincipleAffiliation
        FROM [Groups/GroupsWindow] ]]) AS pw ;
```

The following example demonstrates how errors are logged when the ON ERROR extension and insert_clause are provided with this clause.

```
INSERT INTO OutTradesWithVwap
SELECT    InTrades.*,    Vwap.*
FROM
    InTrades,
    (DATABASE "ExamplesDB"
         SCHEMA (Vwap FLOAT)
         [[
```

```
             SELECT Vwap
            FROM Vwap
            WHERE Symbol = ?InTrades.Symbol
            ORDER BY Ts DESC
            LIMIT 1
          ]]
            CACHE
            MAXIMUM AGE 1 minute,
        ON ERROR
        INSERT INTO errorstream
        SELECT InTrades.*, ERROR_MESSAGE()
        CONTINUE;
) as Vwap
```

REMOTE subquery

Sends Remote Function Calls (RFCs) to an external service, and retrieves the resulting data into a Query statement, Database statement, or Remote Procedure statement.

Syntax

```
( REMOTE QUERY "service"  schema_clause
( [ {value [AS] param} [, ...] ] )    [cache_clause] ) [AS] alias
```

Table 29. Components

service	The name of the remote service as configured in the file c8-services.xml. For more information about configuring services, see the *Sybase CEP Installation Guide* .
schema_clause	A schema definition. See SCHEMA Clause for more information.
value	An expression specifying the value of a parameter for the remote service.
param	The name of a parameter for the remote service as configured in the file c8-services.xml. For more information about configuring services, see the *Sybase CEP Installation Guide* .
cache_clause	Specifications for clearing the cache. See CACHE Clause for more information.
alias	An alias for the subquery.

Usage

A remote subquery is used in the FROM Clause: Database and Remote Subquery Syntax variation of the FROM clause. This specialized subquery invokes a Remote Function Call

(RFC) that has been previously defined in the file c8-services.xml. The remote call is passed to one of a variety of remote services and returns data that is used within the CCL query. Remote calls can include a variety of protocols and interfaces, depending on your configuration and requirements. Examples of supported remote subquery configuration include SOAP and RMI.

The service parameter specifies the name of the service defined in c8-services.xml. This service definition contains all the information necessary to initialize, execute and shut down the service. The remote subquery simply executes the service whenever a row arrives in the stream to which the remote subquery is joined. Each time the service is executed it returns zero or more rows. For more information about configuring services in c8-services.xml, see the *Sybase CEP Installation Guide* .

The remote subquery is only used for function calls that retrieve data into Sybase CEP Engine. Calls that write data to external services must use the Remote Procedure statement.

A SCHEMA clause provides a CCL schema definition for the retrieved data and allows it to be used in CCL queries. CCL clauses within the CCL statement that refer to messages retrieved via the remote subquery must do so using the column names defined by the SCHEMA clause. Since the remote service cannot be preconfigured to send data using a specific data type, the CCL schema must be configured to receive any input, although it is allowed to generate errors when unexpected input is received.

The internal set of parentheses of a remote subquery contains a comma-separated sequence of one or more value expressions and parameter references. The parameter references are the parameters associated with the service, while the CCL values on the left specify the parameter value passed to the service. values can contain literals, column names from the data stream that is joined to the remote subquery, operators, scalar and other (but not aggregate) functions, and parentheses. If the external service takes no values, the remote subquery includes an empty set of parentheses.

An optional CACHE clause sets the criteria for caching rows and clearing the cache. See CACHE Clause for more information.

The entire remote subquery must also be aliased with an AS clause. This alias is used by any clauses in the CCL statement that refer to the remote subquery.

Within the FROM clause the remote subquery is always defined in a join with a single data stream. This join can be either an inner or outer join. Inner joins can be defined using either a comma-separated syntax, or with the use of the JOIN keyword. When the comma-separated syntax is used, the join condition s defined in a WHERE clause. When the JOIN keyword syntax is used, the use of the CCL ON clause with non-remote-subquery joins is not permitted.

Use the ON ERROR extension to handle errors that may cause Sybase CEP Engine to stop subquery execution.

The syntax for the REMOTE subquery clause with the ON ERROR extension is:

```
( REMOTE QUERY "service"  schema_clause
( [ {value [AS] param} [, ...] ] )  [cache_clause] ON ERROR
```

```
[error_insert_clause] CONTINUE )
[AS] alias
```

Additional syntax related to the error_insert_clause:

```
error_insert_clause: INSERT INTO error_stream_name error_select_list
error_stream_name: Name of an output stream or local stream.
error_select_list: SELECT {error_expression} [,...]
error_expression: A CCL expression.
```

The error_expression can only contain:

- References to columns in the stream specified in the stream_clause.
- Operators, constant literals, and scalar functions.
- CCL parameters in the project. These should be prefaced with $.
- The ERROR_MESSAGE() built in.

The ON ERROR extension is optional. The error_insert_clause for the ON ERROR extension is also optional. If the ON ERROR extension is not provided and errors occur, the behavior of this clause is determined by the setting of the IgnoreErrors property in the c8-services.xml service configuration file. By default, this property is set to 'false', which means that if an error occurs, the query is aborted.

When the ON ERROR extension is provided without the error_insert_clause and there is an error executing the subquery, the error is ignored by the Sybase CEP Engine and the subquery execution is considered as having returned no results. Subsequent tuples arriving in the joining stream continue to trigger execution of the subquery. If these executions generate errors then these errors are also ignored by the Sybase CEP Engine.

When the ON ERROR extension is provided with the error_insert_clause and there is an error executing the subquery, a new tuple is inserted into the errorstream (specified by error_stream_name). This tuple contains values of the fields selected in the error_select_list clause. The timestamp of this tuple is the same as the timestamp of the tuple in the joining stream that triggered the execution of this subquery. These tuples are inserted in the errorstream in the same order as that of the original tuples in the joining stream that triggered the execution of this subquery. When there is no error, no tuples are inserted in the errorstream. The responses containing errors are not cached. If there is a subsequent tuple with the same binding key, it triggers another execution of the subquery.

The ON ERROR extension cannot be used with both an EXECUTE STATEMENT DATABASE/EXECUTE REMOTE PROCEDURE clause and a REMOTE SUBQUERY/ PROCEDURE clause in the same CCL statement. This causes errors. To avoid errors, the CCL statements must be broken into separate statements using an intermediate local stream.

Set the MaxRetries parameter in the service definitions for RemoteService to determine how many times Sybase CEP Engine should retry statement execution after experiencing errors.

The parameter in the service file is written as follows:

```
<Param Name="MaxRetries">some_number</Param>
```

where some_number is an integer between 0 and 255.

If the MaxRetries parameter is set to a value greater than 0, say N, then every time there is an error in an Execute Statement or Subquery statement that subscribes to this service definition, the CEP Engine automatically retries the execution of that statement up to a maximum of N number of times. If the execution of the statement fails for all the N number of times, an error tuple is inserted into the errorstream.

The default value for MaxRetries is 0, that is, by default there is no retry.

Restrictions

Queries that include remote subqueries are subject to the following restrictions:

- A query that includes a remote subquery must have exactly one other data source.
- The non-remote-call data source must not be windowed.
- A left outer join that joins a remote subquery to a data stream must list the data stream on the left; a right outer join that joins a remote subquery to a data stream must list the data stream on the right.
- No full outer joins are allowed with remote subqueries.
- The use of the ON clause is not allowed with remote subqueries.
- CCL value expressions within the remote subquery cannot contain aggregate functions.

See Also

- Create Schema Statement
- Remote Procedure Statement
- CACHE
- SCHEMA

Example

This example executes a service called GetManager. Column values from the InOrders stream are passed to the EmployeeID and EmployeeName parameters of the service. Once executed, the joined results of the remote subquery and the data stream are passed to the OrdersWithManager stream.

```
INSERT INTO OrdersWithManager
SELECT *
FROM InOrders LEFT OUTER JOIN
     (REMOTE QUERY "GetManager" SCHEMA GetManagerRes
         (InOrders.EmployeeID AS EmployeeID,
          InOrders.EmployeeName AS EmployeeName)
     ) AS Manager;
```

The following example demonstrates how errors are logged when the ON ERROR extension and insert_clause are provided with this clause.

```
INSERT INTO OutTradesWithVwap
SELECT    InTrades.*,    Vwap.*
FROM
    InTrades,
    (REMOTE QUERY "ExamplesDB"
```

```
         SCHEMA (Vwap FLOAT)
         [[
             SELECT Vwap
            FROM Vwap
            WHERE Symbol = ?InTrades.Symbol
            ORDER BY Ts DESC
            LIMIT 1
         ]]
            CACHE
            MAXIMUM AGE 1 minute,
        ON ERROR
        INSERT INTO errorstream
        SELECT InTrades.*, ERROR_MESSAGE()
        CONTINUE;
) as Vwap
```

FROM clause: Join syntax

Specifies two data sources in a Query Statement, Database statement, or Remote Procedure statement for inner and outer joins created in the JOIN keyword syntax.

Syntax

FROM { *stream* [[**AS**] *alias*] | *stream* [[**AS**] *alias*] keep_clause [keep_clause] | *window_name* [[**AS**] *alias*] |nested_join| xmltable_exp |subquery } [**RIGHT | LEFT | FULL] [OUTER**] **JOIN** { *stream* [[**AS**] *alias*] | *stream* [[**AS**] *alias*] keep_clause [keep_clause] | *window_name* [[**AS**] *alias*] |nested_join | xmltable_exp |subquery }

Table 30. Components

stream	The name of a data stream.
alias	An alias for the stream or window.
keep_clause	The policy that specifies how rows are maintained in the window. See KEEP Clause for more information.
window_name	The name of a window.
nested_join	A nested join. See nested_join for more information.
xmltable_exp	Produces an XML table. See XMLTABLE Expressions in the FROM Clause for more information.

subquery	A subquery. See CCL Subqueries in the FROM Clause for more information.

nested_join

```
FROM { stream [ [AS] alias] | stream [ [AS] alias] keep_clause
[keep_clause] |window_name [ [AS] alias] |nested_join |
xmltable_exp |subquery} [ RIGHT | LEFT | FULL ] [OUTER] JOIN {
stream [ [AS] alias] | stream [ [AS] alias] keep_clause
[keep_clause] | window_name [ [AS] alias] |nested_join |
xmltable_exp |subquery } [on_clause]
```

Table 31. Components

stream	The name of a data stream.
alias	An alias for the stream or window.
keep_clause	The policy that specifies how rows are maintained in the window. See KEEP Clause for more information.
window_name	The name of a window.
nested_join	Another nested join.
xmltable_exp	Produces an XML table. See XMLTABLE Expressions in the FROM Clause for more information.
subquery	A subquery. See CCL Subqueries in the FROM Clause for more information.
on_clause	The join condition. See ON Clause: Join Syntax for more information.

Usage

This variation of the FROM clause is used for creating inner and outer joins that use the JOIN keyword syntax. Any column or data source references in the query's other clauses must be to one of the data sources named in this clause.

For outer joins, the use of an ON clause defining the join selection condition is required. The ON clause is optional for inner joins using the FROM data source JOIN data source syntax. Other clauses, including the WHERE clause, can be used for further data filtering.

This variation of FROM can be used to create inner, left outer, right outer, and full outer joins as follows:

JOIN	All possible combinations of rows from the intersection of both data sources (limited by the selection condition, if one is specified) are published.
RIGHT OUTER JOIN	All possible combinations of rows from the intersection of both data sources (limited by the selection condition, if one is specified) are published. All the other rows from the right data source are also published. Unmatched columns in the left data source publish a value of `NULL` .
LEFT OUTER JOIN	All possible combinations of rows from the intersection of both data sources (limited by the selection condition, if one is specified) are published. All the other rows from the left data source are also published. Unmatched columns in the right data source publish a value of `NULL` .
FULL OUTER JOIN	All possible combinations of rows from the intersection of both data sources (limited by the selection condition, if one is specified) are published. All other rows from both data sources are published as well. Unmatched columns in either data source publish a value of `NULL` .

The data sources used with this syntax can include data stream expressions, named and unnamed window expressions, and subqueries. Data stream and window expressions can also be combined with XMLTABLE expressions. XMLTABLE expressions and CCL subqueries are discussed in XMLTABLE Expressions in the FROM Clause and CCL Subqueries in the FROM Clause. For more information about named and unnamed windows in the FROM clause, see Windows in the From Clause.

Data sources in this variation of the FROM clause can be aliased. For more information on aliasing, see Aliases.

The JOIN variation of the FROM clause is limited to two data sources. Additional data sources can be accommodated by using a nested join as one of the data sources, or by using CCL subqueries, described in CCL Subqueries in the FROM Clause. If a nested join is used, it must be enclosed in parentheses, and can include its own ON subclause. The rules for the use of the ON subclause with a nested join are the same as the rules that govern the use of the ON clause in the join containing the nested join.

Restrictions

- The list of data sources cannot contain more than one data stream expression.
- A full outer join cannot join a window to a data stream.
- A left outer join that joins a window to a data stream must list the data stream on the left; a right outer join that joins a window to a data stream must list the data stream on the right.

See Also

- Database Statement
- Query Statement
- Remote Procedure Statement
- KEEP
- ON

Example

The FROM clause in the following example creates a two-window full outer join.

```
INSERT INTO OutStream
SELECT S1.Symbol, S1.Average, S2.Average
FROM Stream1 AS S1 KEEP 10 ROWS
   FULL OUTER JOIN Stream2 AS S2 KEEP 5 MINUTES
ON S1.Symbol = S2.Symbol
WHERE S1.Symbol = 'IBM';
```

CCL Subqueries in the FROM Clause

Creates a CCL subquery inside the FROM clause of a Query statement, Database statement, or remote preocedure statement.

Syntax

```
( select_clause from_clause [matching_clause] [on_clause]
[where_clause] [group_by_clause] [having_clause]
[order_by_clause] [limit_clause] [output_clause] ) [AS] alias
[keep_clause] [keep_clause]
```

Table 32. Components

select_clause	The select list specifying what to publish. See SELECT Clause for more information.
from_clause	The data sources. See FROM Clause for more information.
matching_clause	A pattern-matching specification. See MATCHING Clause for more information.
on_clause	A join condition. See ON Clause for more information.
where_clause	A selection condition. See WHERE Clause for more information.

group_by_clause	A partitioning specification. See GROUP BY Clause for more information.
having_clause	A filter definition. See HAVING Clause for more information.
order_by_clause	A sequencing definition. See ORDER BY Clause for more information.
limit_clause	A limit on the number of rows to publish. See LIMIT Clause for more information.
output_clause	A synchronization specification. See OUTPUT Clause for more information.
alias	An alias for the subquery.
keep_clause	A window policy. See KEEP Clause for more information.

Usage

The list of data sources in the FROM Clause: Comma-Separated Syntax and FROM Clause: Join Syntax variations of the FROM clause can include CCL subqueries. The main body of the subquery is enclosed in parentheses and follows the same syntax as the Query Statement, but without an INSERT clause and without the final semicolon (;). Subqueries in the FROM clause produce aggregate output and can be nested one inside another (subqueries can contain other subqueries).

The body of the subquery must be aliased with an AS clause. This clause serves two purposes: it creates an implicit destination, into which the results of the subquery are published, and it is used by clauses in the outer query to refer to the subquery output. An implicit schema for the destination defined by the alias is created based on the column names of the data sources specified within the subquery. By default, the destination created by the alias behaves as a data stream and does not keep state. However, one or two KEEP clauses can be added to the alias to create a window. In this case, the subquery destination behaves as a window and keeps state.

Restrictions

- A subquery cannot directly reference a data stream or window in the parent or outer query and vice versa.
- A subquery produces an implicit schema based on the column names of the data sources specified within the subquery. To prevent this implicit schema from having duplicate or ambiguous names, the select list of the subquery's SELECT clause must observe *one* of the following restrictions:
  - The select list must be limited to column references (with or without data source names and/or aliases) and all the column names must be unique.

    OR

- The expression of every select list item must be assigned a unique alias.

Example

The following example contains a subquery that creates a full outer join between two windows based on StreamA and StreamB. The results of the subquery are then joined with a third unnamed window, based on StreamC.

```
INSERT INTO OutStream
SELECT Sq.Subcol1, Sq.Subcol2, Sq.Subcol3, C.Column1, C.Column2
FROM
     (SELECT A.Column1 AS Subcol1, A.Column2 AS Subcol2,
             B.Column2 AS Subcol3
      FROM
         StreamA AS A  KEEP 100 ROWS
         FULL OUTER JOIN
         StreamB AS B KEEP 100 ROWS
      ON A.Column1 = B.Column2) AS Sq KEEP 100 ROWS
   FULL OUTER JOIN StreamC AS C KEEP 100 ROWS
   ON Sq.Subcol3 = C.Column2;
```

XMLTABLE expressions

Produces an XML table output from one of the data sources in a Query statement, Database statement, or Remote Procedure statement.

Syntax

```
{ name [AS] alias } | { stream [AS] alias keep_clause
[keep_clause] } xmltable_func
```

Components

name	The name of a stream or window.
alias	An alias for the stream or window.
stream	The name of a stream.
keep_clause	A window policy. See KEEP Clause for more information.
xmltable_func	An XMLTABLE function.

xmltable_func

```
XMLTABLE ( column ROWS xpath COLUMNS { expression AS
out_column } [, ...] )
```

Components

column	The name of an XML column from the specified stream or window.
xpath	A constant string. Sybase CEP Engine applies this string to column using XPATH.
expression	An expression that contains no aggregators and evaluates to an unambiguous data type.
out_column	An output column.

Usage

Data sources listed in the FROM Clause: Comma-Separated Syntax and FROM Clause: Join Syntax variations of the FROM clause can include an XML table expression. This expression consists of an aliased data stream or aliased window and an XMLTABLE() function. The XMLTABLE() function receives rows from the data stream or window, and produces zero or more rows from every row received from the data stream or window.

The schema of the output from XMLTABLE() usually differs from the schema of the inbound rows. All other clauses in the query that reference the columns of the XMLTABLE() expression data source must reference the new schema produced by XMLTABLE(), not the schema of the associated data stream or window. Where an explicit reference to the data source is required, this data source must be referenced by the alias specified with the stream or window.

The XMLTABLE() function can reference the column names in the data stream or window to which it is linked. All unqualified column names within the XMLTABLE() are assumed to be to this data stream or window.

For every inbound row from the linked stream or window, the contents of the specified column are passed to XMLTABLE(). The constant string specified in XPATH string is then applied via XPATH once successively to each of the elements in the sequence passed to XMLTABLE(). The result of the XPATH operation on column is a sequence of elements, with each element generating one row, here referred to as CurrentXML. Every CurrentXML is treated as an XPATH context node, within the element tree provided by the associated data stream or window: therefore, the path . points to the CurrentXML node.

The list of expressions and output column names under COLUMNS specifies how each CurrentXML should be generated. Every CurrentXML is processed by the list of expressions and outputs into the corresponding column names. The output schema is generated from this list of expressions and column names, therefore, the data type of the column must be clearly identifiable from the expression. The following should be kept in mind in creating unambiguously typed output:

- Constants output as the default data type for the constant.

- The XML data type is permissible in the output.
- The XMLEXTRACT() function implicitly returns an XML data type (or NULL).
- The XMLEXTRACTVALUE() function implicitly returns a STRING data type (or NULL).
- If the desired column data type is other than XML or STRING, an explicit data conversion function must be used.

Expression can contain column references from the linked data stream or window and can contain the XMLEXISTS(), XMLEXTRACT(), and XMLEXTRACTVALUE() functions, which operate on CurrentXML.

The variants of XMLEXISTS(), XMLEXTRACT(), and XMLEXTRACTVALUE() used with XMLTABLE() take only one argument: XPATH string. The implicit first argument in each of these functions is provided by CurrentXML and the functions operate exactly as their two-argument counterparts. When XMLEXTRACT() or XMLEXTRACTVALUE() are used with XMLTABLE() the functions copy only the context node contained in CurrentXML out of the element tree provided by the associated data stream or window.

Example

In the following example, stream Inventory has an XML column called inv, which contains invItem elements. These elements are first found with XPATH, using '//invItem' and a row is constructed for each invItem. The first column, called SKU, has a data type of XML. The second column, called Num, requires STRING to INTEGER conversion:

```
INSERT INTO ItemStream
SELECT L.SKU, L.Num
FROM Inventory AS L XMLTABLE(inv
   ROWS '//invItem'
   COLUMNS
      XMLEXTRACT('SKU') AS SKU,
      TO_INTEGER(XMLEXTRACTVALUE ('NUM')) AS Num);
```

GROUP BY clause

Affects the behaviour of aggergate functions and the OUTPUT clause in the Query statement, database statement, or Remote procedure statement.

Syntax

GROUP BY { *column* | gettimestamp } [, ...]

Table 33. Components

columns	The name of a column in one of the data sources, to partition by value in this column.

gettimestamp	A GETTIMESTAMP function, to partition by the row timestamp.

Usage

The optional GROUP BY clause specifies a list of one or more column references. The list can also contain a GETTIMESTAMP (Scalar Function) function. This clause has the following effect on the behavior of aggregators, window contents, and the OUTPUT clause, where combination refers to a unique combination of values in the list of columns and, optionally, the timestamp referenced by the GROUP BY clause.

- If a GROUP BY clause is present, aggregate functions are executed separately for every combination. In the absence of a GROUP BY clause, aggregation is performed once for each aggregate function for every group of rows received by the clause that contains the aggregate function.
- If a GROUP BY clause is present in a query that includes one or more unnamed window policies (as defined by the KEEP subclause in the FROM clause) all the window policies are implemented separately for every combination, effectively resulting in the creation of a separate window for every combination. (The GROUP BY clause actually adds one or more implicit PER subclauses to each KEEP clause.) This default behavior can be overridden by adding one or more explicit PER clauses or an UNGROUPED clause to the window policies in question. See the description of the KEEP clause for more information.

  In the absence of a GROUP BY clause, a single policy is implemented for every KEEP clause.
- The presence of a GROUP BY clause affects the behavior of the OUTPUT clause, causing it to output separate rows for every combination. See OUTPUT for more details.

See Also

- Database Statement
- Query Statement
- Remote Procedure Statement
- HAVING
- KEEP
- OUTPUT
- GETTIMESTAMP (Scalar Function)

Examples

The following example calculates the average price separately for each symbol in the NamedWindow.Symbol column.

```
INSERT INTO OutStream
SELECT Symbol AS Symbol, AVG(Price) AS Average
FROM NamedWindow
GROUP BY Symbol;
```

This example calculates the average prices as in the previous example, but uses an unnamed window that retains ten rows for each value in the Symbol column.

```
INSERT INTO OutStream
SELECT Symbol AS Symbol, AVG(Price) AS Average
FROM InStream KEEP 10 ROWS
GROUP BY Symbol;
```

HAVING clause

Specifies an optional post-aggregation selction condition in a Query statement, Database statement, or remote procedure statement.

Syntax

HAVING *boolean*

Table 34. Component

boolean	A Boolean expression.

Usage

The HAVING clause is syntactically nearly identical to the selection condition version of the WHERE clause. Unlike WHERE, however, the HAVING clause filters rows after they have been processed by the GROUP BY clause (if one is present) and can itself contain aggregate functions. The selection condition in this clause can also include literals, column references from the query's data sources listed in the FROM clause, operators, scalar and miscellaneous functions, and parentheses. The HAVING clause cannot include subqueries or references to streams that are specified as non-events in the MATCHING clause. Column references within the selection condition must refer to columns in one of the query's data sources. The HAVING clause is generally used in conjunction with the GROUP BY clause (however, a GROUP BY clause can be used without a HAVING clause) or to create selection conditions that require the use of aggregate functions. The HAVING clause can also be used in conjunction with the WHERE clause, where the WHERE clause filters rows before aggregators are executed on them, and the HAVING clause then filters the results.

Restrictions

- A HAVING clause cannot include subqueries.
- A HAVING clause cannot include references to non-events (data streams specified with ! event-name syntax in the MATCHING clause).

See Also

- Database Statement
- Query Statement

- Remote Procedure Statement
- GROUP BY
- WHERE

Example

In the following example, the HAVING clause filters out groups where the average value of the Salary column is less than 3000:

```
INSERT INTO OutStream
SELECT I.Dept, AVG(I.Salary)
FROM InStream AS I KEEP ALL
GROUP BY I.Dept
HAVING AVG(I.Salary) >= 3000.00;
```

INSERT clause

Indentifies destinations for the results of a Query statement or Insert Values statement.

The following sections describe two variants of the **INSERT** clause. The Query statement and Insert Values statement must begin with this clause. You can use either variant to begin a Query statement. However, an Insert Values statement must use the INSERT INTO form of the INSERT clause.

- INSERT INTO form: Identifies a single destination for the results of a Query statement or Insert Values statement
- INSERT WHEN form: Identifies multiple destinations for the results of a single Query statement, creating a branch. Rows are inserted into one of the specified destinations based on one or more conditions applied against the query's SELECT list.

INSERT INTO clause

Indentifies a single destination for the results of Query statement or insert values statement.

Syntax

INSERT INTO *name* [**(** *column* [, ...]**)**]

Table 35. Components

name	The name of a window or a local or output stream in the current query module.
column	The name of a column in the specified stream or window.

Usage

This form of the **INSERT** clause is used as the first clause of a Query statement whenever a query has a single destination. When branching into separate destinations, use the INSERT WHEN form of the **INSERT** clause. INSERT INTO must also appear as the first clause in an Insert Values statement. This clause indicates the destination for the statement. Rows are published to the specified destination any time the statement executes and produces output. The destination must be a named window or a local or output data stream.

INSERT INTO can also optionally specify one or more column names referring to the schema of the destination stream or window. This list explicitly indicates the columns to which data should be published, from left to right. The number and data types of the columns specified in the **INSERT INTO** clause must correspond to the number of items and data types of the select list in the statement's SELECT clause or column expressions in the VALUES clause. The left-most item in the select list or column expression list is published to the left-most column specified in the **INSERT INTO** clause; the next item is published to the next column indicated in the **INSERTINTO**, and so on, in order of select list or values list items and **INSERTINTO** columns.

Alternately, in the case of a Query statement you can explicitly match select list items with the destination's column names by using the AS output column reference syntax in the SELECT clause. See SELECT Clause for more information regarding this syntax.

When neither the **INSERT INTO** clause nor the SELECT clause (in the case of a Query statement) explicitly lists columns of the destination, the schema of the destination stream or window must entirely match the select list or column expression list in its number of columns and data types. Select or value list item output is then published to destination columns from left to right, as before, but based on the destination's entire schema, not the column specifications in the **INSERTINTO**.

If the destination stream or window includes more columns than you specify in the INSERT INTO, Sybase CEP Engine sets the unlisted columns to NULL when rows are published to the destination. If no columns are explicitly stated in the INSERT INTO, Sybase CEP Engine attempts to publish information to all columns of the destination.

Automatic Schema Creation

Under certain circumstances, you can use the **INSERTINTO** clause in a Query statement to create a name and schema for a previously undefined data stream. The data stream can then be used in subsequent CCL statements in the current module. Automatic schema creation can only be performed on a data stream that does not receive data from outside the current project.

The first Query statement that invokes the data stream must state the stream's name in the query's INSERT clause and list the stream's columns either in the INSERT clause, or in the SELECT clause. When the INSERT INTO form is used for this purpose, the clause must use the following syntax:

```
INSERT INTO stream ( column [, ...] )
```

Sybase CEP Engine then automatically creates the stream schema and data types for the stream. The data types for the columns of the automatic schema are determined by the data types of the data sources in the first Query statement that uses the stream as its destination.

Restrictions

- Input data streams cannot be designated as statement destinations. (An input stream is a type of Sybase CEP Engine stream that receives data from outside the current query module, either by way of bindings, or through a connection to an input adapter.)
- You cannot use duplicate column references in the **INSERTINTO** list of columns (you can, however, use duplicate column references in the SELECT clause of a Query statement).
- A Query statement that defines an automatic schema for a stream must appear in the query module before any CCL statement that uses the stream, either as a data source or destination.

See Also

- Insert Values Statement
- Query Statement

Example

The following example publishes the quantity and price of rows from InvoiceStream to OutStream, if the InvoiceStream rows have value in the Pretax column greater than 50. Total price is calculated by multiplying the Pretax column of InvoiceStream by 1.08 (simulating the amount of tax on the item). The quantity is published to the Quantity column of OutStream, and the calculated total price is published to OutStream's Price column.

```
INSERT INTO OutStream (Quantity, Price)
SELECT S.Quantity, S.Pretax * 1.08
FROM InvoiceStream AS S
WHERE S.Pretax > 50;
```

INSERT WHEN clause

Identifies a single destination for the results of a continuous query created by a Query statement.

Syntax

```
INSERT { WHEN condition THEN name [ ( column [, ...]) ] }
[, ...] [ ELSE name [ ( column [, ...]) ] ]
```

Table 36. Components

condition	A Boolean expression.

name	The name of a window or a local or output stream in the current query module.
column	The name of a column in the window or stream.

Usage

This form of the INSERT clause is used as the first clause of a Query statement to create a branch, in which rows from the query are inserted into one of several destinations, depending on the specified conditions. (For queries where no branching is required, use the INSERT INTO form of the INSERT clause instead.) Rows are published to one of the specified destinations any time the query executes and produces output. All destinations must be named windows, or local or output data streams.

INSERT WHEN contains one or more WHEN subclauses, each of which specifies a condition tested against the query's select list for every incoming row. The specified conditions are similar to CASE expressions, and are subject to similar restrictions.

The select list of every incoming row is tested against the condition of the first (left-most) WHEN subclause. If the condition is not met, and other WHEN subclauses are present, the row is tested against the conditions of the subsequent WHEN subclauses from left to right, until a condition is found to be true. Once a condition is met, the row is inserted into the destination specified by the corresponding WHEN subclause. The conditions of any subsequent WHEN subclauses are not considered for the row in question. If the incoming row meets none of the specified conditions, and an ELSE subclause is present, the row is published to the destination specified by the ELSE subclause. If no ELSE subclause is specified, the row is discarded. As with the INSERT INTO form, INSERT WHEN publishes results only if the query as a whole generates output.

A destination is specified by every WHEN subclause and by the ELSE subclause (if one is present) of the INSERT WHEN. The same destination can be associated with multiple WHEN subclauses, and/or with the ELSE subclause.

Every destination specified by the INSERT WHEN can also optionally include one or more column names referring to the schema of a destination stream or window. As in the case of INSERT INTO, these specify the columns to which data should be published from left to right. Alternately, as in the case of INSERT INTO, select list items can be explicitly matched with the destination's column names by using the AS output column reference syntax in the SELECT clause.

The usage and restrictions on the use of destination schemas and specified column names in the WHEN and ELSE subclauses of the INSERT WHEN is the same as for INSERT INTO, except for the following differences:

- Every item in the select list of the query's SELECT clause must have a corresponding column in the schema of at least one (but not necessarily all) of the destinations specified by the INSERT WHEN.
- The schema corresponding to a destination specified by a given WHEN or ELSE subclause, must have columns corresponding to some, but not necessarily all, items in the

query's select list. Any select list items that do not correspond to a column in the destination are discarded.

- If a given WHEN or ELSE subclause in an INSERT WHEN includes a list of column names referring to its destination, the number of specified columns cannot exceed the number of items in the query's select list, but can contain fewer columns than items in the select list. Any select list items that do not correspond to a column in the destination are discarded.

Automatic Schema Creation

As with INSERT INTO, the WHEN and ELSE subclauses of the INSERT WHEN can be used to create names and schemas for previously undefined data streams. See the AUTOMATIC SCHEMA CREATION USING INSERT INTO section under INSERT INTO for more information about the usage and restrictions of automatic schema creation with the INSERT clause.

The automatic schema feature can be used in either the WHEN subclauses or the ELSE subclause of INSERT WHEN. When used in a WHEN subclause, the automatic schema feature uses the following syntax:

WHEN *condition* **THEN** *stream* **(** *column* [, ...] **)**

In the ELSE subclause, the following syntax is used:

ELSE *stream* **(** *column* [, ...] **)**

Restrictions

- Input data streams cannot be designated as query destinations. An input stream is a type of data stream that receives data from outside the current query module, either by way of bindings, or through a connection to an input adapter.
- No duplicate column references are permitted inside a single WHEN or ELSE subclause of the INSERT WHEN.
- The query that defines the automatic schema for a stream must appear in the query module before any CCL statement that uses the stream, either as a data source or destination.

See Also

- Query Statement

Example

The following example separates the Dept stream into three destinations, based on the values in the Loc column.

```
INSERT
  WHEN Loc IN ('New York', 'Boston') THEN
    Dep_east
  WHEN Loc IN ('Chicago') THEN
    Dep_mid
  ELSE
```

```
    Dep_west
SELECT Deptno, Dname, Loc
FROM Dept;
```

KEEP clause

Specifies a window policy in a Create window statement or in the From clause of a Query Statement, Database statement, or Remote Procedure statement.

Syntax

```
time_policy | count_policy
```

Table 37. Components

time_policy	Specify that the window maintain rows based on time. See time_policy for more information.
count_policy	Specify that the window maintain rows based on count. See count_policy for more information.

time_policy

```
KEEP { { [EVERY] interval [OFFSET BY interval] } | FOR
interval_col | UNTIL times_list } [ PER column [...] |
UNGROUPED ]
```

Table 38. Components

interval	An Interval literal specifying the maximum age of rows in the window or the amount of time to shift the starting point for the time calculation.
interval_col	The name of a column of type Interval.
times_list	A list of times indicating when the window should be emptied. See times_list for more information.
column	The name of a column.

times_list

```
time_spec | (time_spec [, ...])
```

Table 39. Component

time_spec	A time specification. See time_spec for more information.

time_spec

```
' [ SUN | MON | TUE | WED | THU | FRI | SAT ] hour : minute [ :
second [. fraction ]] [timezone]'
```

Table 40. Components

hour	A value from 0 to 23 indicating the hour of the day. Must be preceded by at least one space.
minute	A value from 0 to 59 indicating the minute.
second	A value from 0 to 59 indicating the second.
fraction	A value from 0 to 999999 indicating the fraction of a second.
timezone	A string representing the time zone. If omitted, assumes the local time zone. See "Sybase CEP Time Zone Database" in the Sybase CEP Integration Guide for more information about valid time zone strings. Must be preceded by at least one space.

count_policy

```
KEEP { [EVERY] count BUCKETS { BY column } [...] } |
{ { [EVERY] count  ROW[S] } | { LAST [ROW] } | { count
{ LARGEST | SMALLEST } [DISTINCT] ROW[S] { BY column } [...] }
| { ALL [ROW[S]] } [ { PER column } [...] | UNGROUPED ] }
```

Table 41. Components

count	The number of rows or buckets to maintain.
column	The name of a column.

Usage

The KEEP clause defines a policy for a named or unnamed window. Named windows are created with a Create Window statement, while unnamed windows are defined in the FROM clause.

The window definition determines the manner in which the window keeps state. Rows are retained and removed from the window based on the window definition. Note that, when a window serves as a data source for a query in a query chain, the window's policy has no effect on which rows are propagated down the chain, or on how quickly the rows are propagated. Window policies keep information about the state of rows, but do not delay them, or prevent them from traveling down the chain. Window policies also do not prevent removed or discarded rows from propagating down the chain if the rows are not filtered out by other clauses in the query.

Window policies include time-based policies and count-based policies:

- A time-based policy specifies the time duration for which a window retains rows.

- A count-based policy specifies the maximum number of rows (or groups of rows called buckets) that the window can retain.

In most cases, a window definition can include a single count-based policy, a single time-based policy, or one of each. However, the count-based BUCKETS window policy cannot be combined with a time-based policy.

Both count-based and time-based policies can be sliding or jumping:

- A sliding policy removes rows one at a time.
- A jumping policy clears the window of rows every time the maximum row count is reached or the specified time interval passes.

The following table describes the different types of window policies:

Window Type	Basic Syntax	Behavior
Sliding Time-Based	KEEP interval or KEEP FOR interval_col	Each new row arriving in the window is retained for a specific length of time and is then removed, regardless of the number of other rows contained in the window, or of the other rows' arrival time. In the case of KEEP interval, every arriving row is retained for the same interval, specified by constant-interval-expression, for example 2 MINUTES. In the case of KEEP FOR interval-col, every arriving row is retained for the interval specified in its interval-col column. Since the intervals specified in this column can be different for different rows, rows can be retained for different time intervals.

Window Type	Basic Syntax	Behavior
Jumping Time-Based	KEEP EVERY interval or KEEP UNTIL times-list	In the case of KEEP EVERY interval, time is subdivided into interval segments. During each interval, the window accumulates whatever rows arrive. At the end of the interval, all rows held in the window are removed and the process starts over. In the case of KEEP UNTIL times-list, a comma-separated list of one or more times is specified in the window policy. The window accumulates whatever rows arrive until the first applicable time in times-list, then removes all rows and starts over. Components of times-list can include STRING literals and STRING type parameters. Times listed in times-list can include the day of the week. If no day of the week is specified, rows are removed at the indicated time daily. The listed times can also specify a time zone. If no time zone is specified, rows are removed according to local time. **Note:** Time changes, such as the change between standard and daylight savings time, may cause undesired results in the behavior of KEEP UNTIL windows when the time change occurs. To avoid this problem, Sybase recommends omitting times of the day or week that coincide with the time change from your time list.
Sliding Count-Based	KEEP count ROW or KEEP count ROWS or KEEP LAST ROW A variation of sliding count-based windows can also be defined to keep buckets of rows, instead of individual rows See WINDOW BUCKETS for more information.	The window accumulates arriving rows until it reaches the maximum specified by count or the word LAST (KEEP LAST ROW is the equivalent of KEEP 1 ROW). Once the maximum is reached, every new row arriving in the window displaces one of the other rows already held in the window. When no additional syntax is used, the displaced row is the oldest row held by the window. This default behavior of the sliding count-based window using the count (but not the KEEP LAST) syntax can be changed with the use of the LARGEST or SMALLEST keyword, as described later.

Window Type	Basic Syntax	Behavior
Jumping Count-Based	KEEP EVERY count ROW or KEEP EVERY count ROWS A variation of jumping count-based windows can also be defined to keep buckets of rows instead of individual rows. See WINDOW BUCKETS for more information.	The window accumulates arriving rows until it reaches the maximum specified by integer-expression. When the next row arrives, all older rows held in the window are removed, and the window begins accumulating rows again until it reaches the maximum.
Multi-Policy	One count-based policy and one time-based policy, as described above.	The window retains and removes rows according to both policies.
Keep All	KEEP ALL or KEEP ALL ROWS	All rows arriving in the window are retained. No rows are removed from the window. **Warning!** Because a Keep All window retains all rows that arrive in the stream, it is potentially very costly in terms of resources, and may eventually use up all available memory on your system. Sybase recommends using this type of window only when you are sure that doing so will not exceed your allocated resources, or when you have provided a means of explicitly deleting rows from the window with a Delete statement.

Window Buckets

Count-based windows can retain a specified number of groups of rows (called buckets), instead of the specified number of individual rows. These types of windows are defined using the BUCKETS keyword, and include one or more BY subclauses, each of which includes a column reference corresponding to the window's schema. Rows that contain the same values in all of the columns identified by the BY subclauses are grouped into a single bucket. The window policy determines how many buckets the window can hold.

Every arriving row is placed into a bucket, based on the values it contains in the columns named in the BY subclauses. If a bucket for the row does not already exist, a new bucket is created.

Syntax	Behavior
KEEP count BUCKETS BY column [BY column] [...]	The window accumulates buckets up to the maximum specified by the count. Once the maximum number of buckets is reached, every new bucket created in the window displaces the oldest existing bucket.
KEEP EVERY count BUCKETS BY column [BY column] [...]	The window creates buckets up to the maximum specified by the count. Once the maximum number of buckets is reached and a subsequent new bucket is created, all existing buckets held in the window are removed, and the window begins accumulating buckets again until it reaches the maximum.

A window policy that uses buckets cannot be used in the same window with a time-based window policy.

Keeping Largest or Smallest Rows

By default, sliding count-based windows retain the newest rows, and remove the oldest. For windows that do not use buckets, this behavior can be modified with the addition of the LARGEST or SMALLEST keyword, with or without the DISTINCT keyword, which causes the window to keep only rows with the largest or smallest available values, as described below. At least one BY subclause is required with this syntax, and multiple BY subclauses can be used.

Syntax	Behavior
... count LARGEST ROWS...	Initially the window accumulates rows until it reaches the maximum specified by count. A new row arriving in the window after the window is full is evaluated for the value of col1, where col1 is the column specified in the first BY subclause. If this value is larger than at least one other row already in the window, the new row is retained, and the row with the smallest col1 value is removed. If the incoming row's col1 is smaller than col1 of all the current rows, the incoming row is expired. If col1 is the same for both the new row and all the other rows in the window, and a second BY subclause is specified, the same comparison is performed on the basis of the column specified in the second BY subclause. If the values in the second column are also the same for all rows in the window and the incoming row, the same comparison is performed for the next BY subclause, if one exists, and so on in order of BY subclauses from left to right, until either a value differential is found, or no more BY subclauses remain. If, at any point, the value in the appropriate column of the incoming row is determined to be smaller than the corresponding value of all the rows in the window, the incoming row is discarded. If no differences in value are found at the end of this procedure, however, the oldest row is removed.

Syntax	Behavior
... count LARGEST DISTINCT ROWS...	A new row arriving in the window is checked against the existing rows to determine if the new row is distinct from all the others. Distinction is determined by comparing the values of all columns specified by all BY subclauses in the window policy for all the current rows in the window and the incoming row. If two rows contain the same values in all these columns, the rows are not distinct. If the incoming row is not distinct from another row in the window, the new row is retained, and the older row is removed. Initially the window accumulates distinct rows until it reaches the maximum specified by count. If a new row is distinct, the value of its first BY subclause column is compared to all the other rows in the window. If any rows have a smaller value in this column than the incoming row, the new row is retained, and the row with the smallest value is removed. If the incoming row has a smaller value than any of the existing rows, it is discarded. If the window policy has multiple BY subclauses and the incoming row and all the current rows have the same value in the column indicated by the first BY subclause, the column specified by the second BY subclause is evaluated for the row with the smallest value, and so on through any other subsequent BY subclauses, until a row with a different and smaller value is found.

Syntax	Behavior
... count SMALLEST ROWS...	Initially the window accumulates rows until it reaches the maximum specified by count. A new row arriving in the window after the window is full is evaluated for the value of col1, where col1 is the column specified in the first BY subclause. If this value is smaller than at least one other row already in the window, the new row is retained, and the row with the largest col1 value is removed. If the incoming row's col1 is larger than the value of col1 in all the current rows, the incoming row is discarded. If col1 is the same for both the new row and all the other rows in the window, and a second BY subclause is specified, the same comparison is performed on the basis of the column specified in the second BY subclause. If the values in the second column are also the same for all rows in the window and the incoming row, the same comparison is performed for the next BY subclause, if one exists, and so on in order of BY subclauses from left to right, until either a value differential is found, or no more BY subclauses remain. If, at any point, the value in the appropriate column of the incoming row is determined to be larger than the corresponding value of all the rows in the window, the incoming row is discarded. If no differences in value are found at the end of this procedure, however, the oldest row is removed.

Syntax	Behavior
... count SMALLEST DISTINCT ROWS...	A new row arriving in the window is checked against the existing rows to determine if the new row is distinct from all the others. Distinction is determined by comparing the values of all columns specified by all BY subclauses in the window policy for all the current rows in the window and the incoming row. If two rows contain the same values in all these columns, the rows are not distinct. If the incoming row is not distinct from another row in the window, the new row is retained, and the older row is removed. Initially the window accumulates distinct rows until it reaches the maximum specified by count. If a new row is distinct, the value of its first BY subclause column is compared to all the other rows in the window. If any rows have a larger value in this column than the incoming row, the new row is retained, and the row with the smallest value is removed. If the incoming row has a larger value than any of the existing rows, it is discarded. If the window policy has multiple BY subclauses and the incoming row and all the current rows have the same value in the column indicated by the first BY subclause, the column specified by the second BY subclause is evaluated for the row with the largest value, and so on through any other subsequent BY subclauses, until a row with a different and larger value is found.

Effect of PER, GROUP BY, and UNGROUPED

If a window policy includes one or more PER subclauses, a separate window is effectively created for every unique combination of values found in the combination of column references listed by the PER subclauses. For example, the following window policy keeps ten rows for every unique combination of values found in Col1 and Col2:

```
KEEP 10 ROWS PER Col1 PER Col2
```

Additionally, when a GROUP BY clause is used in the same query as an unnamed window definition, the GROUP BY clause creates implicit PER clauses on the window policy, based on the column references contained in the GROUP BY. For example, the following two code snippets create windows that behave the same:

```
GROUP BY Symbol, Broker, Price
FROM Stream1 KEEP 10 ROWS
```

```
FROM Stream1 KEEP 10 ROWS PER Symbol PER Broker PER Price
```

When the query includes both a GROUP BY clause and an explicit PER clause, however, the explicit PER clause supersedes the implicit PER created by the GROUP BY. For example, the following query causes the window policy to retain ten rows per each unique value in the Broker column, not per the values in the Symbol column:

```
INSERT INTO OutStream
SELECT MAX(Price)
FROM Stream1 KEEP 10 ROWS PER Broker
GROUP BY Symbol;
```

The effect of the GROUP BY clause on an unnamed window can also be overridden by the addition of the UNGROUPED keyword, which causes the window policy to retain rows according to the specified number or rows or interval, disregarding the columns specified in the GROUP BY clause. For example, the following query keeps only one set of 50 rows, regardless of the values in the Symbol column:

```
INSERT INTO OutStream
SELECT AVG(Price)
FROM Stream1 KEEP 50 ROWS UNGROUPED
GROUP BY Symbol;
```

In most cases, the use of the PER, GROUP BY, or UNGROUPED clauses has no effect on the retention behavior of time-based window policies, and on the KEEP ALL window, since time-based policies retain rows for a specified period of time or until a specified time, regardless of their grouping, and KEEP ALL retains all rows regardless of grouping. However, the use of PER, GROUP BY, or UNGROUPED does affect the output of the FIRST and LAST functions, which retrieve the first or last row of the current group (instead of the contents of the window) when a PER or GROUP BY clause is present.

See Also

- Database Statement
- Query Statement
- Remote Procedure Statement
- FROM
- GROUP BY

Examples

The following jumping count-based named window definition keeps every 15 rows:

```
CREATE WINDOW Window1
SCHEMA (Col1 INTEGER, Col2 STRING)
KEEP EVERY 15 ROWS;
```

The following time-based jumping window offsets the beginning of its interval calculation by six minutes:

```
INSERT INTO OutStream
SELECT *
FROM Stream1 KEEP EVERY 60 MINUTES OFFSET BY 6 MINUTES;
```

This example keeps the three distinct rows with the largest value in the Price column for every stock symbol listed in the StockSymbol column:

```
INSERT INTO OutStream
SELECT *
FROM Trades KEEP 3 LARGEST DISTINCT ROWS BY Trades.Price
     PER Trades.StockSymbol;
```

This window creates a separate bucket for every arriving row with a distinct stock symbol. Rows with stock symbols for which buckets have already been created are placed into those buckets. After the third bucket is created, any new bucket displaces the oldest bucket in the window:

```
CREATE WINDOW NewestSymbols SCHEMA StockSchema
KEEP 3 BUCKETS BY StockSymbol;
```

This window expires rows at 5:00pm every weekday:

```
INSERT INTO OutStream
SELECT *
FROM Stream2
KEEP UNTIL ('MON 17:00:00', 'TUE 17:00:00', 'WED 17:00:00', 'THU
17:00:00', 'FRI 17:00:00');
```

MATCHING clause

Detects a pattern of events with specified temporal relationships.

Syntax

MATCHING [**ONCE** [**(** *event* [, ...] **)**] [*interval* : pattern] [on_clause]

Table 42. Components

event	The name or alias of a stream listed as one of the query's data sources.
interval	An Interval expression.
pattern	A pattern. See pattern for complete syntax.
on_clause	An ON clause. See ON Clause: Pattern Matching Syntax for more information.

pattern

```
[!] { event | ( pattern ) | [ interval : pattern ] } [ {&& | ||
| ,} [!] { event | ( pattern ) | [ interval : pattern ] } ]
[...]
```

Table 43. Components

event	The name or alias of a stream listed as one of the query's data sources.
pattern	Another, nested pattern.
interval	An Interval expression.

Interval Expression: A valid interval literal, or an expression that evaluates to an interval.

Event Name: The name or alias of a data stream that is also listed as one of the query's data sources. If a stream is referenced more than once in a pattern definition, each reference must use its own alias.

Usage

The optional MATCHING clause follows the FROM clause in a Query statement, Database statement, or Remote Procedure statement. When used, the MATCHING clause must immediately follow the FROM data source list, which must include all data sources specified in the MATCHING clause, and cannot include any other data sources.

The MATCHING clause allows a query to monitor several data sources for a specified pattern of rows that arrive, or don't arrive, in the query's data sources within a specified time interval. If the specified pattern is detected, and other query selection conditions are met, the query publishes output to its destination.

Sybase CEP pattern matching is inclusive; a pattern is considered matched, as long as the specified events occur (or do not occur) as indicated, even if additional unspecified events also occur during the designated interval.

When used without the ONCE subclause, the MATCHING clause produces one output row for each unique combination of input events that matches the pattern (subject to other query selection conditions). Data streams can also contain overlapping occurrences of patterns, in which case all occurrences are published. This rule can be modified with the ONCE subclause.

If the MATCHING clause is used with a ONCE subclause, the query outputs no more than one pattern match for every event specified by the ONCE subclause. In this case, the published pattern match will be the first pattern match involving the specified event to be detected and to fall within the WHERE clause selection condition, if such a condition is indicated. If the ONCE subclause does not specify a list of events, the once-only rule applies to all events in the pattern.

The MATCHING clause of a query that includes multiple data sources can contain an optional ON subclause, which defines one or more multi-equalities that limit the join condition.

Queries containing a MATCHING clause can also include other selection criteria, such as a WHERE clause, that further refine the matching conditions for the pattern. However, only the WHERE clause can place selection conditions on non-events. See the WHERE clause description for more information.

The outer level of a pattern definition consists of a bounded pattern definition. Bounded pattern definitions are enclosed in square brackets ([ ]) and contain:

- An interval expression specifying the interval within which the pattern must be detected. The interval expression can include any CCL elements that are evaluated at compile time, such as functions, operators and literals, but cannot contain references to stream or window columns or to other elements whose value is determined at run time.
- A pattern specification, indicating the events or groups of events that must occur, or not occur, within the specified interval to meet the pattern matching criteria.

  The pattern specification includes one or more of the following:
  - Event specifications: An event is specified by indicating the name or alias of a data stream to which the query subscribes. The event occurs every time a row arrives in the specified data stream.
  - Nested bounded patterns: Syntactically identical to the outer bounded pattern. Nested bounded patterns define a smaller subinterval within the larger interval, and a series of events or groups of events, that must occur or fail to occur, within the smaller interval to produce a pattern match.

  Where a pattern specification consists of more than one event, the events or groups of events must be connected with the operators listed in the following table. A special ! operator signifies that the event or group of events must not occur to produce a pattern match. Non-occurring events are sometimes called non-events.

All the components of a bounded pattern, including nested bounded patterns, must occur within the interval specified for the bounded pattern to produce a pattern match.

Table 44. Event Operators

Operator	Operator Name	Description
!	Not operator	Specifies a negative condition for a pattern component. (Pattern conditions are met when the pattern component does *not* occur).
&&	Conjunction (And) operator	Both pattern components linked by the conjunction operator must occur for the match condition to be met, but they need not necessarily occur in the order listed.

Operator	Operator Name	Description
||	Disjunction (Or) operator	One or both pattern components linked by the Disjunction operator must occur to meet the conditions of the match. Each output row produced by a Disjunction match shows the match for one of the members of the Disjunction, and NULL values for the other members. This is true even when several members of the disjunction produce events.
,	Sequence operator	Pattern components linked by the Sequence operator must both occur in the order listed, to meet the conditions of the match. The listed order refers to the rows' timestamp, not necessarily the order in which the rows appear in a stream viewer. A sequence pattern matches a given set of events only if the events associated with the sequence occur in the specified order, *and the intervals associated with the events do not overlap.* Pattern interval rules are discussed below.

The default order of precedence in which pattern components are analyzed for a possible pattern match follows the order of operators, as they are listed in the table. The tightest binding between an operator and a pattern component is that of the Not operator. The bindings then get progressively looser, for events linked with a conjunction, disjunction, and sequence operators, respectively. This default order of precedence can be overridden by enclosing a pattern component in parentheses.

Pattern Definition Validity Rules

A valid pattern definition and all of its components must be properly anchored. Anchoring refers to whether or not the beginning and end of the time interval corresponding to the pattern component can be determined. Anchoring can be fixed by the row timestamp of an event within the pattern component, or can be deduced from the context of the component. Pattern components are referred to as being left-anchored (a pattern component with a fixed interval start time), right-anchored (a pattern component with a fixed interval end time), fully anchored (a pattern component that is both left-anchored and right-anchored), or unanchored (a pattern component that is neither left-anchored nor right-anchored). The following rules must be satisfied for a pattern definition to be considered valid:

1. Every pattern component within any bounded pattern must be either left-anchored, right-anchored, or both.
2. For any two consecutive pattern components within a sequence, the left component must be right-anchored, and/or the right component must be left-anchored.

The following table specifies the anchoring status of all pattern components.

Pattern Component Type	Anchoring Status
A single event A	Fully anchored. The event can be anchored in time by its row timestamp.
Non-event !A, where A is a single event or other pattern component	Unanchored. Since the event definition specifies an event or group of events that does not occur, no specific start or end time can be associated with it.
A conjunction A && B, where A and B are single events or other pattern components	Left-anchored if, and only if, both A and B are left-anchored. Right-anchored if, and only if, both A and B are right-anchored.
A disjunction A || B, where A and B are single events or other pattern components	Left-anchored if, and only if, both A and B are left-anchored. Right-anchored if, and only if, both A and B are right-anchored.
A sequence A, ..., B where A and B are single events or other pattern components	Left-anchored if, and only if, A is left-anchored. Right-anchored if, and only if, B is right-anchored.
A bounded-pattern [interval-expression: A] where A is a single event or other pattern component	Always fully anchored. The pattern's start and end times are determined by the specified interval-expression and A's timestamps.

Here are some examples illustrating pattern definition validity rules:

```
[10 SECONDS: A, B, !C]
```

This is a valid pattern definition. The sequence A, B, !C is left-anchored (because A is left-anchored), so the first anchoring rule is satisfied. A is right-anchored and B is left-anchored, so the second anchoring rule is satisfied for A and B. B is right-anchored, so the second anchoring rule is satisfied for B and !C.

```
[1 SECOND: !A, B, !C]
```

This pattern definition is invalid. The sequence !A, B, !C fails the first anchoring rule, because it is unanchored.

```
[3 SECONDS: A, !B, !C]
```

This pattern definition is invalid. !B is not right-anchored and !C is not left-anchored, so the pattern definition fails the second anchoring rule.

```
[5 MINUTES: A && (!B, C)]
```

This pattern definition is valid. The conjunction is right-anchored, since both A and (!B, C) are right-anchored.

```
[20 SECONDS: [10 SECONDS: !A, B], !C]
```

This pattern definition is valid. The nested bounded pattern is fully anchored, so the sequence [10 SECONDS: !A, B], !C is valid and left-anchored.

Pattern Interval Rules

A pattern component definition, together with the actual incoming events scanned for pattern matches, determine a time interval, within which the A pattern component can be said to occur. Specifically, the pattern component's interval is determined by subtracting the pattern component's starting time from its ending time. Where the starting time and ending time are the same, the interval is considered to be NULL.

The following rules specify how the starting time and ending time of each component are determined:

A single event A

- The starting time and ending time are determined by the row timestamp of A.

Non-event !A, where A is a single event or other pattern component

- Although a non-event does not contain a start or end time, the context in which the non-event appears (conjunction, disjunction, sequence) exactly specifies the interval to which it applies, as determine by the other rules in this list.

A conjunction A && B, where A and B are single events or other pattern-components

- The starting time is the earlier of the starting times of A and B.
- The ending time is the later of the ending times of A and B.
- A left-unanchored operand of a conjunction takes its starting time from the starting time of the conjunction.
- A right-unanchored operand of a conjunction takes its ending time from the ending time of the conjunction.

A disjunction A || B, where A and B are single events or other pattern components

- The starting and ending times are those associated with A or B: whichever of the two occurs.
- A left-unanchored operand of a disjunction takes its starting time from the starting time of the disjunction.

- A right-unanchored operand of a disjunction takes its ending time from the ending time of the disjunction.

A sequence A, ..., B where A and B are single events or other pattern components

- The starting time is the starting time of A.
- The ending time is the ending time of B.
- If the first pattern component of a sequence is left-unanchored, it takes its starting time from the starting time of the sequence.
- If a pattern component is left-unanchored, and is not the first operand of a sequence, its starting time is calculated by adding one microsecond to the ending time of the previous operand in the sequence.
- If a pattern component is right-unanchored, and is not the last operand of a sequence, its ending time is calculated by subtracting one microsecond from the starting time of the next operand in the sequence.
- If the last pattern component of a sequence is right-unanchored, it takes its ending time from the ending time of the sequence.

A bounded pattern [interval-expression: A] where A is a single event, or other pattern-component

- If A is fully anchored, the starting and ending times are those of A.
- If A is left-unanchored, the ending time is the ending time of A, and the starting time is determined by the ending time of A minus the result of the specified interval-expression.
- If A is right-unanchored, the starting time is the starting time of A, and the ending time is determined by the starting time of A plus the result of the specified interval-expression.
- A left-unanchored operand of a bounded pattern takes its starting time from the starting time of the bounded pattern.
- A right-unanchored operand of a bounded pattern takes its ending time from the ending time of the bounded pattern.

Here are some illustrations of pattern interval rules.

```
[10 SECONDS: A, B, !C]
```

The interval for the whole pattern starts when a row arrives in stream A and ends ten seconds later. The interval associated with B starts one microsecond after the arrival of the row in A (that is, a row in stream B must arrive after the arrival of the row in A, and must not overlap A to produce a pattern match). Likewise, the interval associated with C starts one microsecond after the arrival of the row in B and extends until ten seconds have elapsed since A's arrival.

```
[5 MINUTES: A && (!B, C)]
```

The interval for the whole pattern ends when a row arrives in either stream A or C - whichever arrives later - and starts five minutes before this time. The interval associated with B starts at the same time as the interval for the whole pattern, and ends when the row arrives in C.

```
[20 SECONDS: [10 SECONDS: !A, B], !C]
```

The interval for [10 SECONDS: !A, B] ends when a row arrives in B, and starts ten seconds earlier. The interval for the whole pattern starts ten seconds before the row arrives in B and ends 20 seconds later (that is, ten seconds after the row arrives in B.) The interval associated with A starts ten seconds before the arrival of a row in B minus one microsecond. The interval associated with C starts one microsecond after a row arrives in B and ends ten seconds after the row arrives in B.

Capturing Matching Row Data

The XMLPATTERNMATCH() function can be used in the SELECT clause of the query that contains the MATCHING clause to generate an XML tree that contains data from all the rows (except non-events) that resulted in successful pattern matches. The XML tree can then be published to the query's destination.

Restrictions

The following restrictions apply to the use of pattern matching:

- The MATCHING clause supports pattern matches only within data streams and subqueries, not database subqueries or windows.
- A stream name or alias name can appear only once within a pattern definition. In the case of multiple references to the same stream, each reference must use a different alias.
- Events that are directly or indirectly specified as non-events cannot be referred to in the query's select list, since a non-existent row cannot be published.
- Events that are directly or indirectly specified as non-events can be referenced with certain restrictions in the query's WHERE clause. See WHERE for details.
- The interval expression specifying the interval within which the pattern must be detected can include any CCL elements that are evaluated at compile time, such as functions, operators, and literals, but cannot contain references to stream or window columns or to other elements whose value is determined at run time.

See Also

- Database Statement
- Query Statement
- Remote Procedure Statement
- ON
- XMLPATTERNMATCH()

Examples

The following example monitors for a row arriving in the HighTemperature stream, followed by a row arriving in the HighSmoke stream, and no row in the SprinklersOn stream, all within a 120-second interval, where the RoomNumber column in the HighTemperature and HighSmoke streams contain the same value:

```
INSERT INTO OutStream
SELECT HighTemperature.RoomNumber
FROM HighTemperature, HighSmoke, SprinklersOn
   MATCHING [120 SECONDS: HighTemperature, HighSmoke, !SprinklersOn]
ON HighTemperature.RoomNumber = HighSmoke.RoomNumber
   = SprinklersOn.RoomNumber;
```

The pattern defined in the following example is similar to the previous one, but allows the rows in HighTemperature and HighSmoke to occur in either order. The pattern interval in this example is shorter (20 seconds) than in the previous one, and each stream has an associated alias. A condition selection requires the Building column in all three streams to contain the same value:

```
INSERT INTO BackupAlarmSystem
SELECT 'SprinklerFailure'
FROM HighSmoke AS HS, HighTemperature AS HT, SprinklersOn AS S
   MATCHING [20 SECONDS: HS && HT, !S]
ON HS.Building = HT.Building = S.Building;
```

Here is another example of a pattern definition:

```
INSERT INTO OutStream
SELECT A.x, B.x, C.x
FROM A, B, C, D, E, F, G, H, I
   MATCHING [20 SECONDS: A && B, C && D, E, !F, G || H, !I];
```

The following example contains a parentheses-enclosed subpattern that overrides the default event operator order of precedence:

```
INSERT INTO OutStream
SELECT A.x, C.x, D.x
FROM A, C, D
   MATCHING [1 SECOND: (A || B) && C, D];
```

The following example contains a nested bounded pattern definition that monitors for events with a five-second interval, nested within the larger ten-second interval of the outer pattern:

```
INSERT INTO OutStream
SELECT A.x, D.x
FROM A, D
   MATCHING [10 SECONDS: A, !B, [5 SECONDS: !C, D]];
```

The following example shows a pattern-matching specification that contains a ONCE subclause, which limits the output of matches involving stream A, causing the query to output only the first pattern match involving A, where the value of B.x is greater than 100.

```
INSERT INTO OutStream
SELECT A.x, B.x
FROM A, B
   MATCHING ONCE (A) [10 SECONDS: A, B]
WHERE B.x > 100;
```

ON clause

Specifies join conditions for joins preformed using the JOIN keyword, creates a pattern join condition, or used to identify the trigger which causes rows to be deleted, updated, or inserted into a named window, or which causes variables to change state.

The following subsections describe three syntax variations of the ON clause. The first syntax variation is used to specify join conditions for joins performed using the JOIN keyword. The second syntax variation is used in pattern definition, with the MATCHING clause, to create a pattern join condition. The third syntax variation is used with a Delete statement, Set Variable statement, or Update Window statement, or to identify the trigger which causes rows to be deleted, updated, or inserted into a named window, or which causes variables to change state.

- ON Clause: Join Syntax
- ON Clause: Pattern Matching Syntax
- ON Clause: Trigger Syntax

ON clause: Trigger syntax

Specifies the stream or named window in which the arrival of a row triggers the deletion of rows from a named window or the resetting of a variable state.

Syntax

```
ON name [ [AS] alias ]
```

Table 45. Components

name	The name of a data stream or window.
alias	An alias for the data source.

Usage

This form of the ON clause appears as the first clause in a Delete statement, Set Variable statement, or Update Window statement. The ON clause specifies the name of a data stream or named window in which the arrival of a row is a trigger. When the clause is used in a Delete statement, the trigger prompts the deletion of rows from a specified named window. In a Set Variable statement, the trigger causes specified variables to be reset. In an Update Window statement, the trigger prompts existing rows in a named window to be updated or new rows to be inserted.

The optional WHEN clause can be used in conjunction with the ON clause to limit the conditions for the trigger.

The data stream or window specified in this form of the ON clause can include an alias specified with the syntax stream-or-window-name [AS] alias.

Use this alias to refer to the trigger data stream or window in the WHEN clause of a Delete statement or Update Window statement, as well as in the SET, WHERE, and OTHERWISE INSERT clauses of an Update Window statement.

See Also

- Delete Statement
- Set Variable Statement
- Update Window Statement
- OTHERWISE INSERT
- SET
- WHEN
- WHERE

Examples

The following example monitors for the arrival of a row in InStream. When the row arrives, window rows in MyWindow1 that have the same symbol as the arriving InStream row are deleted.

```
ON InStream
DELETE FROM MyWindow1
WHERE InStream.Symbol = MyWindow1.Symbol;
```

In the following example, the rows in StreamB are counted.

```
ON StreamB
SET message_count = message_count+1;
```

ON clause: Join syntax

Specifies a join condition for data sources joined with a JOIN keyword.

Syntax

ON *inner_condition* | *source.column* = *source.column* [**AND** ...]

Table 46. Components

inner_condition	A Boolean expression.
source	The name or alias of a data source.
column	The name of a column.

Usage

This form of the ON clause is required with outer joins, but is optional with inner joins specified with the JOIN keyword syntax. The ON clause defines a condition for the join.

- When used with outer joins, the condition must consist of one or more simple equality comparisons, each of which compare a column in one data source with a column in the other data source. When multiple column comparisons are specified, they are separated by the AND keyword.
- When used with inner joins, the condition can be any valid Boolean expression that evaluates to true or false.

All column references in the ON clause must refer to data sources specified in the FROM clause. This form of the ON clause is used in a Query statement, Database statement, or Remote Procedure statement.

Restrictions

- Outer join conditions are limited to comparisons between two or more columns in the two data sources of the join. The comparison cannot specify a literal value, or compare two columns in the same data source.
- The ON clause cannot be used with joins specified using the FROM Clause: Database and Remote Subquery Syntax or FROM Clause: Comma-Separated Syntax. Join conditions for such joins are specified using the WHERE clause.

See Also

- Database Statement
- Query Statement
- Remote Procedure Statement
- FROM
- WHERE

Example

The following example publishes information about the symbol and price of a stock when an inquiry for the stock arrives on the Inquiry stream:

```
INSERT INTO OutStream
SELECT Trades.Symbol, Trades.Price
FROM Inquiry JOIN Trades
ON Inquiry.Symbol = Trades.Symbol;
```

ON clause: Pattern matching syntax

Optional clause that specifies a pattern join condition. This form of ON syntax is used in conjunction with a MATCHING clause in a Query statement, database statement, or remote procedure statement.

Syntax

```
ON {source.column = source.column [= ...] } [AND ...]
```

Table 47. Components

source	The name or alias of a data source.
column	The name of a column.

Usage

A query that includes a MATCHING clause pattern definition can also include an optional ON clause that defines a join condition for the pattern. The ON clause consists of one or more multi-equalities. Each multi-equality must refer to one column from every data source listed in the query's FROM clause. The multi-equality condition causes Sybase CEP Engine to search for the specified pattern only among rows where the listed columns of the various data sources contain equal values. All the data sources specified in the ON clause must also be listed as data sources for the query.

A separate pattern search is performed for every unique value contained in the specified columns. This form of the ON clause can be used in a Query statement, Database statement, or Remote Procedure statement.

SEE ALSO

- Database Statement
- Query Statement
- Remote Procedure Statement
- MATCHING

Example

The following example monitors for three consecutive login failures, within three minutes of one another, from the same terminal, and associated with the same user ID:

```
INSERT INTO OutStream
SELECT L3.TerminalID, L3.UserID
FROM LoginFailure AS L1, LoginFailure AS L2, LoginFailure AS L3
   MATCHING [3 MINUTES: L1, L2, L3]
ON L1.TerminalID = L2.TerminalID = L3.TerminalID
   AND L1.UserID = L2.UserID = L3.UserID;
```

ORDER BY clause

Orders multiple rows produced simultaneously by a join operation in a Query Statement, Database statement, or Remote Procedure statement.

Syntax

```
ORDER BY { column [ ASC[ENDING] | DESC[ENDING] ] } [, ...]
[ UNGROUPED | PER column [ ... ] ]
```

Table 48. Component

column	The name of an input column.

Usage

The optional ORDER BY clause orders multiple rows that are produced during one execution of a query. This clause is typically used with joins involving two or more data sources, in which a single row arriving in one of the query's data sources often results in multiple rows of output from the query. In cases where multiple rows are produced, the rows are first ordered according to the values of the first column reference specified in the clause. Rows with duplicate values in this column are further ordered by the second and subsequent column references, if any are specified. Each additional column reference further refines the ordering process.

You can specify DESCENDING (or DESC) with each column reference. These control whether ordering proceeds in ascending (the default) or descending order of values for each specified column.

When specifying a grouping with PER, ordering is performed for each unique combination of values in the specified columns. Specifying UNGROUPED instead of a PER clause to make the default behavior explicit has no effect. Note that any GROUP BY clause in the statement does not apply to the ORDER BY clause.

Restrictions

- The ORDER BY clause reorders rows only within sets of rows produced simultaneously by a join operation. It does not reorder rows that have the same timestamp for other reasons and it never reorders rows with different timestamps.

Example

The following example orders the output from the join by ascending values in the Temp.Location column. If multiple rows have the same value in Location, they are further ordered by the Wind.Windspeed column, in descending order:

```
INSERT INTO OutStream
SELECT Temp.Location, Temp.Temperature, Wind.WindSpeed,
       AVG(Temp.Temperature)
FROM Temp KEEP 1 DAY, Wind KEEP 1 DAY
WHERE Temp.Location = Wind.Location
GROUP BY Temp.Location
ORDER BY Temp.Location, Wind.Windspeed DESC;
```

OTHERWISE INSERT clause

Defines a list of values and column associations, which generates a new row and publishes it to the destination named window specified in an Update Window statemen if a trigger occurs but no current rows in the window match the statement's update condition.

Syntax

```
OTHERWISE INSERT value [AS column] [, ...]
```

Table 49. Components

value	An expression that evaluates to a value of the same data type as the destination column.
column	The name of a column in the statement's destination, specified by the UPDATE clause, to which value is published.

Usage

This optional clause of an Update Window statement takes effect only when a row arrives in the triggering stream or window specified in the ON clause and meets the criteria of the WHEN condition if one is set but no rows in the statement's destination window, defined in the UPDATE clause, match the update condition set by the WHERE clause. In other words, when a trigger occurs but the destination window doesn't contain any rows that can be updated.

The OTHERWISE INSERT clause contains an insert list of one or more items, separated by commas. Each expression in the insert list can contain literals, column references from the triggering stream or window specified in the Update Window statement's ON clause, operators, scalar and miscellaneous functions, and parentheses, or can contain the "select all" asterisk (*) character, which is equivalent to a list of all column values from the triggering stream or window in the ON clause, listed in order from left to right.

Items in the insert list that are not an asterisk can include an AS subclause, indicating a column in the destination window's schema to which the results of the expression are published. You must use the AS subclause either for all of the non-asterisk items or for none of the items in the insert list.

If the insert list includes AS subclauses, the output column reference associated with each expression must uniquely match a column name in the destination's named window schema, and the expression must evaluate to the column's data type. In the absence of an AS subclause, items are published in order from left to right.

The insert list must contain a value for every column in the destination window's schema.

RESTRICTIONS

- OTHERWISE INSERT clause expressions cannot include column values from the named window specified in the UPDATE clause, since OTHERWISE INSERT executes only when no corresponding value is found in this named window.

See Also

- Update Window Statement
- ON
- UPDATE
- WHEN
- WHERE

Examples

```
ON ShippedOrders AS S
WHEN Code = "Ship Notice"
UPDATE Orders AS O
SET TRUE AS Shipped, S.ShippedTime AS ShippedTime
WHERE S.OrderID = O.OrderID
OTHERWISE INSERT S.OrderID AS OrderID, TRUE AS Shipped, S.ShippedTime
AS ShippedTime;
```

OUTPUT clause

Sychronizes, limits, or delays output from a Query statement, database statement, or Remote Procedure statement, or schedules the generation of rows from an Insert Values statement.

Syntax

OUTPUT { [**ALL**] **EVERY** { *count* **ROW**[**S**] | *interval* [**OFFSET BY** *interval*] } } | { {**AFTER** | **FIRST WITHIN**} {*count* **ROW**[**S**] | *interval*} } | { [**ALL**] **AT** times_list } [**UNGROUPED** | **PER** *column* [...]]

Table 50. Components

count	The number of rows to publish.
interval	An Interval literal specifying when to publish rows or the amount of time to shift the starting point for the time calculation.
times_list	A list of times indicating when the window should be emptied. See times_list for more information.

column	The name of a column in the data source, for grouping purposes.

times_list

```
time_spec | (time_spec [, ...])
```

Table 51. Component

time_spec	A time specification. See time_spec for more information.

time_spec

```
{ ' [ SUN | MON | TUE | WED | THU | FRI | SAT ] hour : minute
[ : second [. fraction ]] [timezone]' } |STARTUP
```

Table 52. Components

hour	A value from 0 to 23 indicating the hour of the day. Must be preceded by at least one space.
minute	A value from 0 to 59 indicating the minute.
second	A value from 0 to 59 indicating the second.
fraction	A value from 0 to 999999 indicating the fraction of a second.
timezone	A string representing the time zone. If omitted, assumes the local time zone. See "Sybase CEP Time Zone Database" in the Sybase CEP Integration Guide for more information about valid time zone strings. Must be preceded by at least one space.

Usage

The **OUTPUT** clause can be used as the last clause of a Query statement, Database statement, or Remote Procedure statement, and is required in an Insert Values statement. OUTPUT is the last clause to be executed, after the statement performs all other data filtering and processing, and before it generates output. The **OUTPUT** clause comes in four variations, each of which has a distinct syntax and usage: **OUTPUT EVERY** and **OUTPUT ALL EVERY**, **OUTPUT AFTER**, **OUTPUT FIRST WITHIN,** and **OUTPUT AT**.

In a Query statement, Database statement, or Remote Procedure statement, all variations of the OUTPUT clause control the statement's output based on the number of rows arriving in the statement's data sources, or the rows' time of arrival. The **OUTPUT** clause processes these rows regardless of window definitions, and is unaffected by the statement's KEEP clause conditions, if any are present. An Insert Values statement uses only the OUTPUT AT variation

of the OUTPUT clause, which determines when values are generated and inserted into the specified stream or window.

OUTPUT EVERY and OUTPUT ALL EVERY

Limits the statement's output to specific time intervals, or number of rows received. The output specification can be indicated as a number of rows or as a time interval.

- A time-based OUTPUT EVERY clause publishes one or no rows every interval expression, where interval expression is the interval specified in the clause. If output from the statement was generated within the last interval expression, the most recently generated row is published. If no rows were generated within the last interval expression, no rows are published.
- A time-based OUTPUT ALL EVERY clause publishes the row most recently generated by the statement every interval expression, as specified in the clause. The row is published whether or not it was generated during the most recent interval expression or during a previous interval expression. If the statement generated no rows since it started running, no rows are published.

By default, interval expression calculation for the OUTPUT EVERY and OUTPUT ALL EVERY clause is calculated from midnight of January 1, 1970 GMT/UTC. This starting time can be offset by the interval expression specified in the OFFSET BY subclause.

- A count-based OUTPUT EVERY clause publishes every Nth row of output from the statement, where N is the number of rows specified as integer expression in the clause.
- A count-based OUTPUT ALL EVERY clause behaves identically to the count-based OUTPUT EVERY clause. (Since the OUTPUT EVERY clause publishes the Nth row whenever it is generated by the statement, and does not publish a row until the Nth row is generated, the concept of publishing rows from a previous interval does not apply).

The OUTPUT EVERY clause can be combined with a GROUP BY clause. In the following descriptions, combination refers to a unique combination of values in the list of columns and, optionally, the timestamp referenced in the GROUP BY clause:

- A time-based OUTPUT EVERY clause, in conjunction with GROUP BY, publishes rows every interval expression, as specified in the clause. OUTPUT EVERY publishes the most recent row for every combination defined in the GROUP BY clause, for which rows were generated in the most recent interval expression. No rows from previous interval expressions are published.
- A time-based OUTPUT ALL EVERY clause, in conjunction with GROUP BY, publishes rows every interval expression, as specified in the clause. OUTPUT ALL EVERY publishes the most recent row for every combination defined by the GROUP BY clause encountered from the time the statement started executing. If no row was generated by the statement for a specific combination within the most recent interval expression, the most recent row from a previous interval expression is published. If the statement generated no rows since it started running, no rows are published.

- A count-based OUTPUT EVERY clause, in conjunction with GROUP BY, publishes every Nth row of output from the statement for every combination defined by the GROUP BY clause, where N is the number of rows specified as integer expression in the clause.
- A count-based **OUTPUTALLEVERY** clause, in conjunction with GROUP BY, behaves identically to the count-based **OUTPUTEVERY** clause, in conjunction with GROUP BY.

OUTPUT AFTER

Delays the publication of all rows generated by the statement, either by the specified interval, or until such time as the specified number of subsequent rows have been generated. However, no rows are filtered out with this clause; all rows that would otherwise have been published are published eventually.

OUTPUT FIRST WITHIN

Publishes only the first row generated by the statement, within a specified time interval, or within a specified number of rows, using the following rules:

1. The first row generated by the statement is published.
2. For count-based clauses, the N-1 subsequent rows generated by the statement are ignored, where N is the number of rows specified as integer expression in the OUTPUT FIRST WITHIN clause. For time-based clauses, all rows generated by the statement within the next interval expression are ignored, as specified in the OUTPUT FIRST WITHIN clause.
3. The first row to be generated by the statement after the specified number of rows are ignored, or the specified interval elapses, is published.
4. The interval or row count restarts and all subsequent rows are ignored until the count expires.

When the **OUTPUT FIRST WITHIN** clause is combined with a GROUP BY clause, these steps are performed separately for every combination defined in the GROUP BY.

OUTPUT AT

In a Query statement, Database statement, or Remote Procedure statement, an OUTPUT AT clause limits statement output to one or more daily or weekly times, as specified by a times list. The times list can include STRING literals and STRING type parameters. Times listed in the times list can include the day of the week. If no day of the week is specified, output occurs at the indicated time daily. The listed times can also specify a time zone. If no time zone is specified, rows are removed according to local time. The STARTUP keyword cannot be used in these statements. An OUTPUT AT clause is also required in an Insert Values statement, where it specifies the times at which rows are generated and published to the destination window or stream. The Insert Values statement OUTPUT AT can use the STARTUP keyword.

For more information, see OUTPUT AT BEHAVIOR IN THE QUERY STATEMENT, DATABASE STATEMENT AND REMOTE PROCEDURE STATEMENT and OUTPUT AT BEHAVIOR IN THE INSERT VALUES STATEMENT.

Note: Time changes, such as the change between standard and daylight savings time, may cause undesired results in the behavior of the **OUTPUT AT** and **OUTPUT ALL AT** clauses when the time change occurs.

To avoid this problem, Sybase recommends omitting times of the day or week that coincide with the time change from your time list.

OUTPUT AT in the QUERY Statement, DATABASE Statement and REMOTE PROCEDURE Statement

- An **OUTPUT AT** clause publishes one or no rows at the specified times. If output from the statement was generated since the last specified time, the most recently generated row is published. If no rows were generated since the last output time, no rows are published.
- An **OUTPUT ALL AT** clause publishes the row most recently generated by the statement at the specified times. The row is published whether or not it was generated since the last output time. If the statement generated no rows since it started running, no rows are published.

The OUTPUT AT clause can be combined with a GROUP BY clause. In the following descriptions, combination refers to a unique combination of values in the list of columns and, optionally, the timestamp referenced in the GROUP BY clause:

- An OUTPUT AT clause, in conjunction with GROUP BY, publishes rows at the specified times. OUTPUT AT publishes the most recent row for every combination defined in the GROUP BY clause, for which rows were generated since the last output time. No rows generated before the last output time are published.
- An OUTPUT ALL AT clause, in conjunction with GROUP BY, publishes rows at the specified times. OUTPUT ALL AT publishes the most recent row for every combination defined by the GROUP BY clause encountered from the time the statement started executing. If no row was generated by the statement for a specific combination since the last output time, the most recent row from before the last output time is published. If the statement generated no rows since it started running, no rows are published.

OUTPUT AT in the INSERT VALUES Statement

Both the OUTPUT AT and the OUTPUT ALL AT clauses produce the rows specified by the VALUES clause and publish them to the statement's destination either at the specified times of the week or day, or on project startup (which you specify with the STARTUP keyword). In a query module where persistence is not turned on, STARTUP refers to any time when the project containing the module starts. In a query module where persistence is turned on, "startup" refers to the time when the project containing the module first starts or starts with a clean slate.

PER Clause

If you specify explicit grouping with a PER subclause, your output specifications apply to each unique combination of values in the indicated input columns. If you omit the PER subclause, any GROUP BY clause in the statement applies to the output clause. If you specify

UNGROUPED, no grouping applies. Note that, while you can use grouping with OUTPUT AFTER interval, the grouping has no effect and generates a warning message.

Restrictions

- The Insert Values statement uses only the OUTPUT AT variation of the **OUTPUT** clause. **OUTPUT EVERY, OUTPUT ALL EVERY**, **OUTPUT AFTER**, and **OUTPUT FIRST WITHIN** cannot be used with this statement.
- Only the Insert Values statement can use the STARTUP keyword in an OUTPUT AT clause.

See Also

- Database Statement
- Insert Values Statement
- Query Statement
- Remote Procedure Statement
- GROUP BY
- VALUES

Examples

The following example publishes the most recent row for every value in the Symbol column within a thirty-second period, where the value in the Price column exceeds fifty:

```
INSERT INTO OutStream
SELECT Symbol, Price
FROM Trades
WHERE Price > 50
GROUP BY Symbol
OUTPUT EVERY 30 SECONDS;
```

This query publishes the value of the Temp column of the TempIn stream for rows with "New York" in the Location column. All output is delayed by ten seconds:

```
INSERT INTO InStream
SELECT Temp
FROM TempIn
WHERE TempIn.Location = 'New York'
OUTPUT AFTER 10 SECONDS;
```

This example publishes every third row from the SFTemp stream:

```
INSERT INTO OutStream
SELECT Temp, MAX(Temp)
FROM SFTemp KEEP 5 ROWS
OUTPUT FIRST WITHIN 3 ROWS;
```

This example publishes daily at 6:00 o'clock in the evening, any rows with a value greater than 100 in the Wind column that were generated in the last 24 hours:

```
INSERT INTO OutStream
SELECT Wind, Time
FROM Weather
WHERE Wind > 100
OUTPUT AT '18:00:00';
```

SCHEMA clause

Provides a schema definition for a named window inside a Create Windowstatement, or for a database subquery or remote subquery inside a Query statement, Database statement, or Remote Procedure statement.

Syntax

SCHEMA *name* | *'file'* | **(** *column type* [, ...] **)**

Table 53. Components

name	The name of a schema created with a Create Window statement.
file	The name of a schema file.
column	The unique name of a column.
type	The data type of the specified column.

Usage

Use the SCHEMA clause as part of a Create Window statement and in a database subquery or remote subquery of the FROM clause to define a schema for the named window, database subquery, or remote subquery. The SCHEMA clause can actually define the schema's column names and data types (called an inline schema), or can refer to a named schema, created with a Create Schema statement, or to an external .ccs schema file that contains the column name and data type definitions. You can create an external schema file in Sybase CEP Studio (see the Sybase CEP *Studio Guide* for more information). If your SCHEMA clause refers to an external file, you must specify the file's absolute path, or its path relative to the module .ccl file that contains the SCHEMA clause.

See Also

- Create Schema Statement
- Create Window Statement
- Database Subquery
- Remote Subquery
- FROM

Examples

Here is an example of a SCHEMA clause used in a Create Window statement:

```
CREATE WINDOW LastDayTrades
SCHEMA (Symbol STRING, Price FLoAT)
KEEP 1 DAY;
```

Here is an example of a SCHEMA clause used within a database subquery:

```
INSERT INTO OutStream
SELECT S.Location, S.Name, D.Age
FROM RfidStream AS S,
  (DATABASE "MyDB" SCHEMA 'mydb.ccs'
     [[SELECT * FROM People WHERE ?S.Name=People.Name ]]) AS D;
```

SELECT clause

Specifies a select list for a query in a Query Statement or Database statement or passes values to the parameters of an external service in a Remote procedure statement.

Syntax

SELECT { *expression* [**AS** *column* | *alias* | *parameter*] }
[, ...]

Table 54. Components

expression	An expression that evaluates to a value of the same data type as the corresponding destination column.
column	The name of a column in the query destination, as specified with the INSERT clause.
alias	An alias for a column as used in the EXECUTE STATEMENT DATABASE clause.
parameter	The name of a parameter used by the remote service.

Usage

The SELECT clause inside a Query statement, Database statement, or Remote Procedure statement specifies a select list of one or more items. Rows from the data sources listed in the FROM clause are passed to the SELECT clause after being filtered by the WHERE clause, if one is specified. The results of the expressions in the list are processed by other clauses (if any). If, after this processing, the CCL statement generates results, the results are handled differently, depending on the type of CCL statement in which they are used:

- In a Query statement, the results are published to the destinations specified in the query's INSERT clause.
- In a Database statement, the SQL statements specified in the EXECUTE STATEMENT DATABASE clause are executed against the external database. The SQL statements usually use the processed select list results as their input.
- In a Remote Procedure statement, the results are passed to the specified parameters (as specified in the AS clause of each expression) in the external service indicated by the EXECUTE REMOTE PROCEDURE clause.

The following rules apply to the select list:

- The expression within each select list item can contain literals, column names from one of the statement's data sources listed in the FROM clause, operators, scalar and miscellaneous functions, and parentheses. Query statement and Database statement select list expressions can also include aggregate functions, but Remote Procedure statement select lists cannot.

  Alternately, expression can be specified with a "select all" asterisk (*) character, which is equivalent to a list of all column values from all data sources listed in the statement's FROM clause, listed in order from left to right, or as data-source.*, which is equivalent to a list of all column values from the specified data source (where data-source is the name or alias of one of the data sources listed in the FROM clause). Note that expressions using the asterisk involve certain limitations when used with the Database statement or Remote Procedure statement, as discussed in the RESTRICTIONS section.
- The following rules apply to all expressions that do not include an asterisk:
  - When used in a Query statement, each list item can specify an AS output column reference subclause indicating the column within the destination, to which the select list item should be published. (The results of the AS subclause here are the same as the usage of explicit column specification in the INSERT clause.) The AS subclause must be used either for all or for none of the non-asterisk items in the select list. In the absence of an AS subclause, or a list of column names in the INSERT clause, items are published from left to right.
  - When used in a Database statement, each non-asterisk list item must specify an AS output-alias subclause. When the SQL query in the EXECUTE STATEMENT DATABASE clause refers to a column in the CCL SELECT clause, the SQL reference must use the column's alias, in the form ?output-alias.
  - When used in a Remote Procedure statement, each non-asterisk list item must specify an AS parameter subclause. The result of each expression is then passed to the specified parameter in the external service.

Automatic Schema Creation

Under certain circumstances the SELECT clause of a Query statement can be used to create a schema for a previously undefined data stream. The data stream can then be used in subsequent CCL statements in the current module. The automatic stream and schema creation can only be performed on a data stream that does not receive data from outside the current project.

The first Query statement that invokes the data stream must list the desired stream name in its INSERT clause, and then list the desired columns for the stream either in the INSERT clause, or in the SELECT clause. When the SELECT clause is used to define the stream's columns the following syntax is used:

```
SELECT { expression AS column } [, ...]
```

The use of the "select all" asterisk (*) is an exception to this syntactical requirement. When the first Query statement that included the data stream uses the "select all" asterisk (*), the column names for the destination stream are automatically inferred from the query's data source.

The stream schema and data types are then automatically created by Sybase CEP Engine. The data types for the columns of the automatic schema are determined by the data types of the data sources in the first Query statement that uses the stream as its destination.

Restrictions

- If the SELECT clause of a Database Statement or Remote Procedure Statement uses the "select all" expression (*) and the FROM clause of the statement lists multiple data sources, the data sources cannot share any column names.
- When the select list of a Database statement or Remote Procedure statement contains a list of expressions separated by commas, all non-asterisk expressions must include an AS subclause. In a Query statement containing a comma-separated list of expressions, either all or none of the non-asterisk expressions must contain an AS subclause.
- CCL expressions within the select list of a Remote Procedure statement cannot contain aggregate functions.

See Also

- Query Statement
- Database Statement
- Remote Procedure Statement
- FROM Clause: Comma-Separated Syntax
- FROM Clause: Database and Remote Subquery Syntax
- FROM Clause: Join Syntax
- INSERT

Examples

Here is an example of the simplest SELECT clause, used by a Query statement that selects all the columns from its data source:

```
INSERT INTO OutStream
SELECT *
FROM Trades
WHERE Trades.Price = 100;
```

The following example includes a more complex Query statement SELECT clause:

```
INSERT INTO StepCompletionCounts
SELECT
    COUNT(C.ScenarioID) AS TheCount,
    C.ScenarioID AS ScenarioID,
    C.StepID AS StepID
FROM
    CompletedSteps AS C KEEP 10 SECONDS
GROUP BY
    C.ScenarioID, C.StepID
OUTPUT
    EVERY 10 SECONDS;
```

Here is an example of a SELECT clause in a Database statement:

```
EXECUTE STATEMENT DATABASE "MyDB"
  [[UPDATE Inventory
    SET Inventory.Quantity = ?Quantity
    WHERE Inventory.ItemID =?ID]]
SELECT
  StreamIn.ItemID AS ID, StreamIn.Quantity AS Quantity
FROM StreamIn;
```

Here is an example of a SELECT clause in a Remote Procedure statement:

```
EXECUTE REMOTE PROCEDURE "Search"
SELECT Name AS SearchString
FROM Clients;
```

SET clause

Sets variables and updates rows in a window.

CCL includes two variations of the SET clause, one used to set variables, the other to update rows in a window. These variants are described in the following subsections:

- SET Clause: Variable Syntax
- SET Clause: Window Syntax

SET clause: Set variable statement syntax

Sets the values of one or more variables in a Set Variable statement.

Syntax

SET *name* = *value* [, ...]

Table 55. Components

name	The name of a variable, previously created with a Create Variable statement.
value	An expression that evaluates to a value of the appropriate data type.

Usage

The SET clause sets the values of listed variables to the output of the specified expression, if the trigger, specified by the Set Variable statement's ON and WHEN clauses occurs.

See Also

- Set Variable statement
- ON
- WHEN

EXAMPLES

```
ON StreamX
WHEN StreamX.Price > 100
SET price_alert = TRUE;
```

SET clause: Window syntax

Indicates which columns of a named window should be updated by the Update Window statement and specifies values for those columns.

Syntax

SET { *value* **AS** *column* [, ...] } | { *column* = *value* [, ...] }

Table 56. Components

value	An expression that evaluates to a value of the same data type as the specified column.
column	The name of a destination column to which value is published.

Usage

The SET clause is used in an Update Window statement, where it indicates which columns in the destination named window (specified by the statement's UPDATE clause) should be updated when the trigger condition is met, and specifies values for those columns. If the statement includes a WHERE clause, the SET clause updates only the rows that meet the

WHERE clause update condition. In the absence of a WHERE clause, the SET clause updates all the rows in the destination window.

The SET clause contains a comma-separated update list, which is specified in either of two syntax forms. In both forms, the update list is comprised of one or more items, each of which includes an expression, and an update-column-reference (note that the order of expression and update-column-reference is reversed in the two forms). Update list expressions can contain literals, column references from the streams or windows specified in the Update Window statement's ON and UPDATE clauses, operators, scalar and miscellaneous functions, and parentheses.

The update-column-reference associated with each expression must uniquely match a column name in the destination's named window schema, and the expression must evaluate to the column's data type. However, the update list does not need to contain a match for every column in the destination window's schema. Any columns not specified in the update list are left unchanged in the destination window. All rows affected by the SET clause receive a new row timestamp.

See Also

- Update Window Statement
- ON
- UPDATE
- WHERE

Examples

```
ON ReturnedOrders AS R
UPDATE Orders AS O
SET Shipped = FALSE, ShippedTime = NULL
WHERE R.OrderID = O.OrderID;
```

UPDATE clause

Specifies the named window in an Update Window statement on which the update or insert should be performed.

Syntax

UPDATE *window*[[**AS**] *alias*]

Table 57. Components

window	A named window.
alias	An alias for the window.

Usage

The UPDATE clause specifies the named window to which the Update Window statement publishes row updates or new rows. Optionally, you can also use this clause to set an alias for the named window, which can be used in references to the window inside the SET and WHERE clauses of the Update Window statement.

See Also

- Update Window Statement
- SET
- WHERE

Example

```
ON StreamA
UPDATE MyWindow AS W
SET W.Column1 = TRUE;
```

VALUES clause

Specifies a list of values in an Insert Values statement to be generated at the specified time and inserted into the statement's destination.

Syntax

VALUES (*column_expression* [, ...] **)** [, ...]

Table 58. Component

column_expression	An expression that evaluates to the same data type as the corresponding destination column. column_expression can contain constants, variables, operators, parentheses, scalar functions, and the COALESCE function.

Usage

VALUES is the second clause inside an Insert Values statement. This clause produces one or more rows, which the statement passes to its destination in the INSERT clause, at the times you specify in the OUTPUT clause. VALUES contains one or more row expressions, each of which is enclosed in parentheses and each of which generates a row that is published to the destination at the specified time. Lists of two or more row expressions are separated by commas.

Every row expression consists of one or more column expressions, each of which generates a value that corresponds to the appropriate column of the destination specified in the INSERT clause. Lists of two or more column expressions are separated by commas. The results of

column expressions within a row expression are matched positionally with the stream schema or column name list in the INSERT clause (see INSERT Clause for more information).

Restrictions

- Expressions in a list of values cannot refer to data stream or window column names.
- Column expressions must produce values that match the data types of the corresponding stream or window columns specified in the INSERT clause.

See Also

- Insert Values Statement.
- INSERT
- OUTPUT

Example

The following example produces two rows when the project first starts and at 18:00:00.1 every day. The first row consists of two columns: one containing the zero and the other the current timestamp, as generated by the NOW function. The second row consists of the value 1 in one column and the current timestamp in the second column. Both rows are published to the Flag and TimeMarker columns of .InitStream.

```
INSERT INTO InitStream (Flag, TimeMarker)
VALUES (0, Now()), (1, Now())
OUTPUT AT ( STARTUP, '18:00:00.1' );
```

WHEN clause

Defines a condition that narrows the cases in which the trigger occurs in a Delete statement, Set Variable statement, or Update Window statement.

Syntax

WHEN *trigger*

Table 59. Component

trigger	A Boolean expression.

Usage

The WHEN clause is syntactically almost identical to the selection condition version of the WHERE clause. It is optionally used in a Delete statement, Set Variable statement, or Update Window statement to create a trigger condition that evaluates rows arriving in the stream or named window specified by the ON clause. When a WHEN clause is present, the trigger that initiates the deletion, update, or insertion of rows from the named window, or the setting of a variable occurs only when the incoming row in the triggering stream or named window meets

the trigger condition. In the absence of the WHEN clause, the deletion or variable setting is triggered whenever a row arrives in the stream or window specified by the ON clause. The trigger condition in this clause can include literals, column references from the data stream or window specified by the ON clause, operators, scalar and miscellaneous functions, and parentheses. The WHEN clause cannot include subqueries or aggregate functions.

Restrictions

- A WHEN clause cannot include subqueries.
- A WHEN clause cannot include aggregate functions.
- In a Delete statement, column references in the WHEN clause cannot refer to the columns of the named window specified in the DELETE FROM clause.

See Also

- Delete Statement
- Set Variable Statement
- Update Window Statement
- ON
- WHERE

Examples

The WHEN clause in the following example initiates a trigger for deletion from a named window only when the Price column in Win1 contains a value greater than 50.00:

```
ON Win1
WHEN Price > 50.00
DELETE FROM Win2
WHERE Win2.Price < 50.00;
```

The WHEN clause in this example initiates a trigger for the setting of the Delay_Length variable only when the Delay column in StreamIn is smaller than 10:

```
ON StreamIn
WHEN Delay < 10
SET Delay_Length = 'Short';
```

WHERE clause

Specifies a selection condition for filtering input from data sources in a Query Statement, Database statement, or Remote Procedure statement, provides join conditions in the FROM Clause: Comma-separated syntax, update conditions in an Update Window statement, and delete conditions in a Delete statement.

Syntax

WHERE *condition*

Table 60. Component

condition	A Boolean expression representing a selection, update, delete, or join condition, depending on the context.

Usage

WHERE is a versatile clause, used for filtering rows in several CCL statements, with similar syntax, but slightly different usage and context in each case.

AS SELECTION CONDITION

The first syntactical form of this clause is used optionally, in conjunction with single-source FROM clauses, as well as the FROM Clause: Join Syntax and FROM Clause: Database and Remote Subquery Syntax versions of the FROM clause. The Boolean expression specified in this clause creates a selection that filters rows arriving in the query's data sources before passing them on to the SELECT clause.

- The selection condition can include literals, column references from the query's data sources listed in the FROM clause, operators, scalar and miscellaneous functions, subqueries, parameters, and parentheses. Subqueries in the WHERE clause are discussed later.
- In a Query statement, Database statement, or Remote Procedure statement, column references within the selection condition must refer to columns in one of the query's data sources. In a Delete statement, column references must refer either to columns in a named window or stream referenced in the ON clause, and/or a named window in the DELETE FROM clause.
- When used in conjunction with a MATCHING clause, the WHERE clause can place selection conditions on streams in which events do not occur, (streams that are specified as non-events using the !stream-name syntax in the MATCHING clause).

  Note that, because non-events track the failure of a row to arrive in a stream, the WHERE clause qualifier on a non-event specification expands rather than limits pattern sequences that are considered to have met the matching criteria. For example, a pattern specification of !A, B requires that no row at all arrive in stream A prior to the arrival of a row in stream B within the appropriate interval, whereas the same pattern specification with a WHERE clause of WHERE A >= 10 permits rows with values smaller than 10 to arrive in stream A.
- WHERE clause filtering is performed before the GROUP BY clause and before aggregation (if any), so cannot include aggregate functions or the filtering of results based on the results of aggregates. Use the HAVING clause for post-aggregate filtering.

AS UPDATE CONDITION

When used in an Update Window statement, the expression in the WHERE clause specifies the criteria for updating existing rows in the named window specified in the UPDATE clause, with the values specified in the SET clause, when the update trigger occurs. If none of the rows

in the window match the update criteria, new rows are inserted into the window, as indicated with the OTHERWISE INSERT clause, if one is present.

The WHERE clause is optional. In the absence of a WHERE clause, all rows in the window are updated when the trigger occurs. If no trigger for an update occurs, no rows are updated or inserted, whether or not a WHERE clause is specified.

- The update condition expression can include literals, column references, operators, scalar and miscellaneous functions, parameters, and parentheses.
- Column references within the update condition must refer to one of the columns in the ON clause, or to the named window in the UPDATE clause.
- In the absence of an OTHERWISE INSERT clause, the update condition can be any valid Boolean expression. However, if you use an OTHERWISE INSERT clause in your Update Window statement, the update condition must consist of one or more simple equality comparisons, each of which compare a column in the ON clause stream or window to the named window in the UPDATE clause. When multiple column comparisons are specified, they are separated by the AND keyword.

AS DELETE CONDITION

When used in a Delete statement, the expression in the WHERE clause specifies the criteria for deleting rows from the named window indicated in the statement, once the deletion trigger occurs. This clause is optional. In the absence of a WHERE clause, all rows are deleted from the window when the trigger occurs. If no trigger for deletion occurs, no rows are deleted from the window, whether or not a WHERE clause is specified.

- The delete condition expression can include literals, column references, operators, scalar and miscellaneous functions, subqueries, parameters, and parentheses.
- Column references within the delete condition must refer to one of the columns in the window's schema.

AS JOIN CONDITIONS

When used in conjunction with the FROM Clause: Comma-Separated Syntax form of the FROM clause, the WHERE clause creates one or more join condition for the comma-separated join. The use of a WHERE clause is optional in a comma-separated join. In the absence of a join condition, all rows from all data sources are selected. When a WHERE clause is present, its syntax resembles the ON Clause: Join Syntax (which is not used with comma-separated joins). The join condition can be any valid Boolean expression that specifies the condition for the join. All column references in this form of the WHERE clause must refer to data sources specified with the FROM clause.

CCL Subqueries

The selection condition or delete condition expression (but not the update condition or join condition) of the WHERE clause can include scalar CCL subqueries using the following syntax: `( select_clause from_clause [matching_clause] [on_clause] [where_clause] [group_by_clause] [having_clause] [output_clause] )`

Table 61. Components

select_clause	The select list specifying what to publish. See SELECT Clause for more information.
from_clause	The data sources. See FROM Clause for more information.
matching_clause	A pattern-matching specification. See MATCHING Clause for more information.
on_clause	A join condition. See ON Clause for more information.
where_clause	A selection condition. See WHERE Clause for more information.
group_by_clause	A partitioning specification. See GROUP BY Clause for more information.
having_clause	A filter definition. See HAVING Clause for more information.
output_clause	A synchronization specification. See OUTPUT Clause for more information.

The subquery is enclosed in parentheses and follows the same syntax as the Query statement, but without an INSERT clause and without the final semicolon. Subqueries in the WHERE clause are only permitted when they produce scalar output (one row per execution of the query). Aggregate output is not permitted, though aggregate functions can be used in the SELECT and HAVING clauses of the subquery to produce a single line of output from multiple lines. A subquery in the WHERE clause cannot directly reference a data stream or window in the parent or outer query and vice versa. WHERE clause subqueries can be nested.

Restrictions

- A WHERE clause cannot use aggregate functions.
- Subqueries in a WHERE clause expression must produce scalar output.
- Joins using the JOIN keyword do not use the WHERE clause to specify join conditions (though they can use the clause in its selection condition form). See ON Clause: Join Syntax for more information.
- The following restrictions apply to a WHERE clause that specifies a relationship between two or more streams contained in a MATCHING clause pattern specification, when one of the members of the relationship appears in the MATCHING clause as a non-event (using the !stream-name syntax).
  - All the members of the relationship in the WHERE clause must appear in the MATCHING clause in a sequence (comma-separated) list.
  - None of the members of the relationship in the WHERE clause can appear in the MATCHING clause in a sub-pattern or nested bounded pattern relationship to the other

members. For example, a where clause of WHERE A = B is allowed with a MATCHING clause pattern specification of MATCHING [5 MINUTES: A, !B, C], but not with a pattern specification of MATCHING [5 MINUTES: A, (!B, C) && D].

See Also

- Query Statement
- Database Statement
- Delete Statement
- Remote Procedure Statement
- FROM Clause: Comma-Separated Syntax
- HAVING
- MATCHING
- ON
- OTHERWISE INSERT
- SET
- WHEN

Examples

The following example uses a subquery in the WHERE clause. The selection condition is used to produce a continuous stream of Boolean values that indicate whether or not a stock price is at least $1.00 above the moving average:

```
INSERT INTO OutStream
SELECT *
FROM Trades
WHERE Trades.Price > 1.00 +
   (SELECT AVG(Price)
    FROM Trades KEEP 100 ROWS);
```

The following WHERE clause defines a selection condition for a join:

```
INSERT INTO OutStream
SELECT Trades.Symbol, Trades.Price, Broker.Name
FROM Trades KEEP 1 DAY, Broker KEEP 1 DAY
WHERE Trades.Name = Broker.Name;
```

The following example updates the Status column of MyWindow for all rows that contain the same symbol as the symbol arriving in the triggering InStream:

```
ON InStream
UPDATE MyWindow
SET MyWindow.Status = "SOLD"
WHERE InStream.Symbol = MyWindow.Symbol;
```

The following example specifies that whenever a row with the message "Clear Window" arrives in the stream InStream, rows with the same value in the Symbol column as the arriving row should be deleted from the NamedWin window:

```
ON InStream
WHEN InStream.Message = 'Clear Window'
DELETE FROM NamedWin
WHERE InStream.Symbol = NamedWin.Symbol;
```

Sybase CEP SQL

The Sybase CEP implementation of Structured Query Language (SQL) makes available a subset of SQL-92 commands for querying the contents of CCL public windows.

Though SQL clauses resemble their CCL counterparts, they are not interchangeable. The following table summarizes the differences between CCL and CEP SQL.

CCL	SQL
Used to develop all Sybase CEP applications.	Used to query the current contents of public window, which is created using the Create Window statement and populated in the same way as all other CCL windows.
Used inside Sybase CEP Engine query modules.	Used in the Query Public Window dialog box of Sybase CEP Studio, in one of Sybase CEP's SDKs, in the c8_client command-line utility, or in a CCL database subquery.
Executes continuously when a project is running, once for every time a row arrives in one of the statement's data sources.	Executes on demand, once each time the query is run.

This section describes the syntax of the SQL Select statement, and its various clauses. For more detailed descriptions of SQL syntax, refer to SQL-92 documentation.

SELECT statement

Queries the contents of a public window.

Syntax

```
simple [ { {UNION [ALL] | INTERSECT | EXCEPT } simple } [...] ]
[ORDER BY column [ ASC[ENDING] | DESC[ENDING] ] [, ...]]
[LIMIT {limit [OFFSET offset | {offset, limit} ]
```

Components

simple	A simple select statement. See simple for more information.
column	The name of a column by which to sort the output.

limit	The maximum number of rows to publish. If negative or omitted, place no limit on the number of rows published.
offset	The number of rows of the result to skip before publishing.

simple

```
select_clause from_clause [WHERE expression] [GROUP BY
expression [, ...]] [HAVING expression]
```

Components

select_clause	The select list. See select_clause for more information.
from_clause	The data sources. See from_clause for more information.
expression	An expression specifying selection, grouping, or filtering conditions. See Usage for more information.

select_clause

```
SELECT [ ALL | DISTINCT ] { expression [[AS] alias] }
[, ...]
```

Components

expression	A SQL-92 expression.
alias	An alias for the output.

from_clause

```
FROM single | inner | outer
```

Components

single	A single data source expression. See single for more information.
inner	An inner join. See inner for more information.
outer	An outer join. See outer for more information.

single

```
{ pub_window [AS alias] } | { ( sub_select ) [AS alias] }
```

Components

pub_window	The name of a public window. If the window is in a submodule, you must identify the submodule by specifying path/submodule/window, where path includes the names of any additional parent submodules, separated by slashes, and submodule is the name of the module containing the window.
alias	An alias for the data source.
sub_select	A nested select statement (a subquery).

inner

```
single [, ...] |single [INNER] JOIN single [ ON expression |
USING ( ids ) ] |single NATURAL [INNER] JOIN single |single
NATURAL [INNER] JOIN single
```

Components

single	A single data source expression. See single for more information.
expression	A SQL-92 expression specifying the join condition.
ids	A list of column IDs specifying the join condition.

outer

```
single { RIGHT |LEFT |FULL} OUTER JOIN single [ ON expression
| USING ( ids ) ]
```

Components

single	A single data source expression. See single for more information.
expression	A SQL-92 expression specifying the join condition.
ids	A list of column IDs specifying the join condition.

Usage

The SQL Select statement queries the current contents of the public windows listed in its FROM clause and generates a result of zero or more rows of data, each of which has a fixed number of columns. In addition to any columns explicitly queried in the statement, the query result includes an initial TIMESTAMP column, containing the row timestamp.

The Select statement includes one or more simple-select components, each of which can contain the following clauses.

- The SELECT clause contains a select list of one or more items, each of which is an expression. When the expressions contain column names, the columns must be from one of the public windows listed in the query's FROM clause. The number of columns in the result of the Select statement is determined by the number of items contained in the select list of the SELECT clause.
- The FROM clause lists one or more public windows or subqueries against which the query is executed. Public windows are created with the CCL Create Window Statement, and are populated in the same manner as other CCL windows. The SQL Select statement is executed against the current output of the specified public windows or subqueries.
- The optional WHERE clause can be used to specify selection conditions and certain join conditions.
- The optional GROUP BY clause causes one or more rows of the result to be combined into a single row of output.
- The optional HAVING clause performs filtering after grouping has occurred.

A compound-select is formed from two or more simple-selects connected by one of the operators described in the following table. All simple-selects in a compound-select must specify the same number of result columns. The following operators can be used to create a compound-select:

UNION	The results to the left and to the right of the UNION operator are combined into a single table. All results are distinct, as duplicate rows are eliminated from the results. NULL values are not treated as being distinct from one another.
UNION ALL	The results to the left and to the right of the UNION operator are combined into a single table. Any duplicate rows are retained in the results.
INTERSECT	Only the intersection between the parts of the compound-select to the left and to the right of the INTERSECT operator are included in the results.
EXCEPT	The results to the left of the EXCEPT operator are published, after removing from them the results to the right of the EXCEPT operator.

When three or more simple-selects are connected into a compound-select, they group from left to right.

The Select statement can also contain no more than one of each of the following clauses:

- The optional ORDER BY clause causes output rows to be sorted.

- The optional LIMIT places an upper limit on the number of rows that can be returned in the result.

Select Clause

Every simple select of the Select statement must include one SELECT clause. This clause specifies a select-list, which is used to generate query results. The number of columns in the result is determined by the number of items in the query's select-lists. In compound selects, the connect operator also partially determines the number of columns in the query result.

A select-list consists of one or more comma-separated items, each of which is an arbitrary SQL-92 expression:

- A select-list expression can refer to column names of the public windows specified in the Select statement's FROM clause, as determined by the window schemas.
- If the FROM clause refers to multiple public windows, any ambiguous column reference in the select-list (usually a reference to a column name defined for more than one of the windows) must be specified in the format public-window-name.column-name.
- The asterisk (*) character can be used to select all columns from all public windows listed in the FROM clause. The asterisk can also be used in combination with a public window name, within a select-list item to indicate that all columns from the specific public window should be selected public-window-name.*.
- Rows from the public window listed in the FROM clause are passed to the SELECT clause after being filtered by the query's WHERE clause, if one is specified.
- The results of the expressions in the list are processed by other clauses (if any) in the query.
- Each select-list expression item can specify an AS output-alias subclause, indicating an alias by which the column results should be published. The item's alias can also be listed directly after the item, omitting the AS keyword.

The select-list can be prefaced by a DISTINCT keyword. If two or more rows contain the same values in all queried columns, the DISTINCT keyword causes only one of the rows to be included in the result. In the absence of the DISTINCT keyword, result rows are returned regardless of whether or not they are distinct. This default behavior can be specified explicitly by using the ALL keyword.

From Clause

The FROM clause specifies the data sources (public windows or subqueries) against which the Select statement is executed. Rows from these sources are filtered by the WHERE clause, if one is specified, and are then passed to the query's SELECT clause.

The FROM clause can refer to a single data source, or to multiple joined data sources. Each data source must be one of the following:

- A public window (created with the CCL Create Window statement).
- A subquery-nested Select statement, enclosed in parentheses.

When your SQL FROM clause refers to a public window that is located in the project's main module, refer to the window by the name you defined for it in the Create Window statement.

When referring to a public window in one of the main module's submodules, use the following syntax in your name reference:

[[parent/ [...]] submodule/window]

Data sources specified in the FROM clause can include an SQL alias, specified with the syntax data-source AS alias. This alias can be used to refer to the data source anywhere in the query where the data source name would otherwise be used. This clause can also be used without the AS keyword.

When the FROM clause refers to joined data sources, the joins can be specified using either the comma-separated format, or the JOIN keyword format. SQL supports standard SQL-92 syntax, which is summarized below.

An inner join publishes all possible combinations of rows from the intersection of two or more specified data sources, according to the limitations of the selection condition (if one is present). Joined data sources can refer only to public windows in the same module or in the module's submodules. SQL provides four options for specifying an inner join:

Comma-separated join	Multiple data sources can be listed one after the other, separated by commas. An optional WHERE clause in the query creates the join condition for this type of join (the ON and USING subclauses of the FROM clause are not used to create a selection condition for comma-separated joins). In the absence of a WHERE clause, the join selects all values from all selected data source columns. This join syntax can include more than two data sources.
JOIN or **INNER JOIN**	Two data sources can be joined with the JOIN keyword, which can also be explicitly qualified as an INNER JOIN. A join condition for this join can be specified by an expression in the ON subclause, or a column ID list in the USING subclause. A WHERE clause selection condition for rows can also be included in the query, but is not used for the join condition specification.
NATURAL JOIN or **NATURAL INNER JOIN**	A join created with the keywords JOIN or INNER JOIN can be further defined as being NATURAL. A natural join is created on any and all columns that have the same names and data types in both data sources. Natural joins do not include a further join condition, though the query can include a WHERE clause selection condition for row filtering.

CROSS JOIN or **CROSS INNER JOIN**	A join created with the keywords JOIN or INNER JOIN can be further defined as being a CROSS join. A cross join returns the Cartesian product from its two data sources. No join condition is used with a cross join, though the WHERE clause selection condition for rows can be included in the query. This join type can be thought of as functioning identically to a comma-separated join with no selection condition and limited to only two data sources.

SQL also supports the following standard syntax for outer joins. An ON or USING subclause is required with outer joins to establish the join condition.

RIGHT OUTER JOIN	All possible combinations of rows from the intersection of both data sources (limited by the selection condition, if one is specified) are published. All the other rows from the right data source are also published. Unmatched columns in the left data source publish a value of NULL.
LEFT OUTER JOIN	All possible combinations of rows from the intersection of both data sources (limited by the selection condition, if one is specified) are published. All the other rows from the left data source are also published. Unmatched columns in the right data source publish a value of NULL.
FULL OUTER JOIN	All possible combinations of rows from the intersection of both data sources (limited by the selection condition, if one is specified) are published. All other rows from both data sources are published as well. Unmatched columns in either data source publish a value of NULL.

Where Clause

The WHERE clause can be used to specify an expression, which is used for one of two purposes:

- As a selection condition, the WHERE clause expression filters rows from the query's data sources, before they are passed to the SELECT clause.
- As a join condition, the WHERE clause can be used with comma-separated inner joins to establish a relationship between the columns in the query's data sources, thus limiting the cross-product output of the join.

Expressions specified inside the WHERE clause cannot refer to aggregate functions.

Group By Clause

The expressions specified in the **GROUP BY** clause cause one or more rows of the result to be combined into a single row of output. **GROUP BY** is often used when the query result contains aggregate functions. **GROUP BY** clause expressions do not have to appear in the result, however.

Having Clause

The HAVING clause is similar to the WHERE clause, but is applied after grouping has occurred. The HAVING expression can refer to aggregate functions and to values that are not included in the result.

Order Clause

The ORDER BY clause sorts the query's output rows according to the specified list of expressions. These expressions do not have to be part of the result when used with a simple-select. When used with a compound-select, however, each sort expression must exactly match one of the result columns. Each expression can optionally be followed by either or both of the following:

- A COLLATE subclause, which specifies the name of a collating function used for ordering text.
- The ASC or DESC keywords, which are used to specify the sort order.

Only one ORDER BY clause can appear inside a SELECT statement.

Limit Clause

The LIMIT clause places an upper limit on the number of rows returned by the query. A negative LIMIT indicates no upper bound. Optionally, this clause can also indicate how many rows should be skipped at the beginning of the result set. Three syntax forms are permitted for the LIMIT clause:

LIMIT *limit-number*	The clause used without an offset specifier simply indicates an integer limiting the number of rows returned by the query.
LIMIT *limit-number* OFFSET *offset-number*	An additional OFFSET subclause specifies the number of rows that should be skipped (offset) at the beginning of the result set.
LIMIT *offset-number, limit-number*	An offset can also be specified with a comma. Notice, that, when this syntax is used, the first integer is the offset number, and the second integer is the limit number. This opposite order from the OFFSET subclause form is an intentional feature of SQL-92, provided to maximize compatibility with legacy SQL database systems.

When used in conjunction with a compound select the LIMIT clause can be used only once, but is applied to all the simple selects in the query.

Supported SQL-92 Expressions

Syntax that are used to refer SQL reference to expression.

SQL references to expression refer to the following syntax, unless indicated otherwise:

```
{ expression  binary expression } |{ expression [NOT] LIKE
expression [ESCAPE expression] } |{ unary expression } |
(expression) |column |pub.column |  literal |parameter
|{ function ( expression | * ) } |{ expression { ISNULL |
NOTNULL } } |{ expression [NOT] BETWEEN expression AND
expression } |{ expression [NOT] IN ( values | select ) } |
{ [EXISTS] ( select ) } |{ CASE [expression] { WHEN expression
THEN expression } [...] [ELSE expression] END } |{ CAST
( expression AS type ) }
```

binary

```
|| | * | / | % | + | - | << | >> | & | | | < | <= | >= | = | ==
| != | <> | IN | AND | OR
```

unary

```
- | + | ! | ~ | NOT
```

Functions

The Sybase CEP is capable of utilizing a variety of function components.

Scalar Functions

Scalar Functions take a list of scalar arugments and return a single scalar value.

Scalar functions can appear in the select list of the SELECT clause as well as in the WHERE and HAVING clauses.

Most scalar functions return Null when called with a Null argument. Exceptions include COALESCE() and many of the XML functions.

Aggregate Functions

Aggregate functions return a single result row based on input consisting of groups of rows.

Aggregate functions can be used in the select lists of a SELECT clause, typically in conjunction with a GROUP BY clause, and in a HAVING clause.

Aggregate functions can be applied to columns or to expressions, but cannot be used in an expression that contains another aggregate expression. For example, the expression SUM(AVG(pi * r * r)) is invalid.

All aggregate functions except COUNT() ignore NULL values when performing their aggregate calculations, but return a value of NULL when all input passed to the function consists of NULLs.

Aggregate Functions and DISTINCT

When you use DISTINCT with aggregate functions, input with duplicate values is considered only once when performing calculations. For example, the values 1, 1, 1, 2, 3 submitted to the function SELECT MEDIAN(DISTINCT col1) are considered by the function as 1, 2, 3.

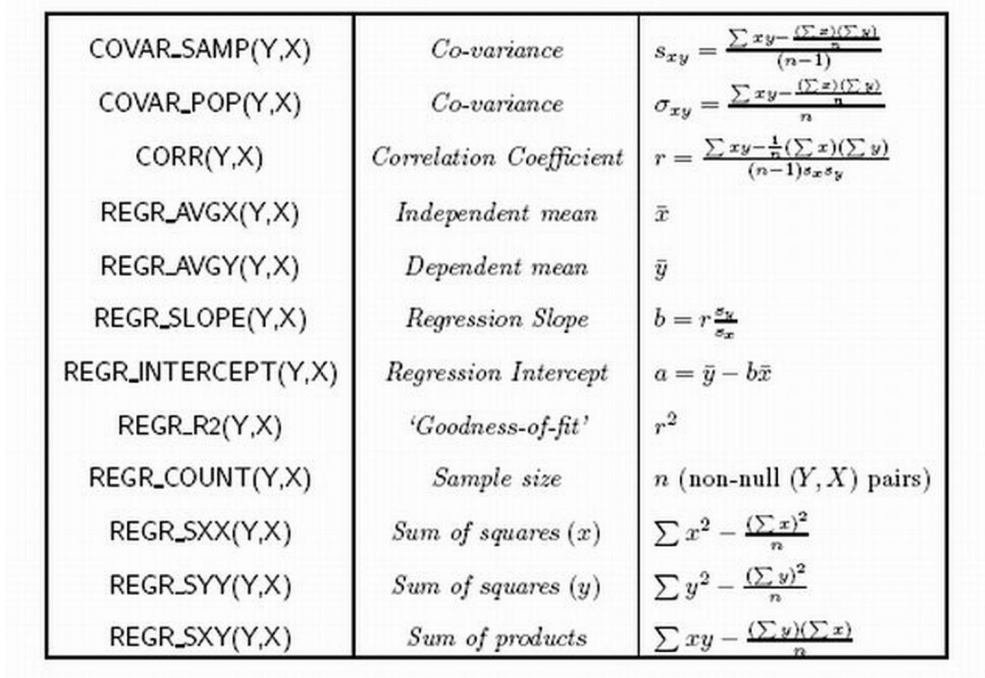
Mathematical Formulas for Aggregate Functions

Two tables provide the equivalent mathematical formulas for several of the aggregate functions supported in Sybase CEP.

Figure 1: Simple aggregate functions

Function	Symbol	Formula
SUM(X)		$\sum_{i=1}^{n} x_i$
MAX(X)		$x_i : x_i \geq x_j, i \neq j \; \forall i,j \in n$
MIN(X)		$x_i : x_i \leq x_j, i \neq j \; \forall i,j \in n$
AVG(X)	$\bar{x}$	$\frac{\sum x_i}{n}$
COUNT(*)		n
VAR_SAMP(X)	s_x^2	$\frac{\sum (x_i - \bar{x})^2}{(n-1)}$
VAR_POP(X)	σ_x^2	$\frac{\sum (x_i - \bar{x})^2}{n}$
VARIANCE(X)		identical to VAR_SAMP(X)
STDDEV_SAMP(X)	s_x	$\sqrt{\frac{\sum (x_i - \bar{x})^2}{(n-1)}}$
STDDEV_POP(X)	σ_x	$\sqrt{\frac{\sum (x_i - \bar{x})^2}{n}}$
STDDEV(X)		identical to STDDEV_SAMP(X)

Figure 2: Statistical aggregate functions



COVAR_SAMP(Y,X)	*Co-variance*	$s_{xy} = \frac{\sum xy - \frac{(\sum x)(\sum y)}{n}}{(n-1)}$
COVAR_POP(Y,X)	*Co-variance*	$\sigma_{xy} = \frac{\sum xy - \frac{(\sum x)(\sum y)}{n}}{n}$
CORR(Y,X)	*Correlation Coefficient*	$r = \frac{\sum xy - \frac{1}{n}(\sum x)(\sum y)}{(n-1)s_x s_y}$
REGR_AVGX(Y,X)	*Independent mean*	$\bar{x}$
REGR_AVGY(Y,X)	*Dependent mean*	$\bar{y}$
REGR_SLOPE(Y,X)	*Regression Slope*	$b = r\frac{s_y}{s_x}$
REGR_INTERCEPT(Y,X)	*Regression Intercept*	$a = \bar{y} - b\bar{x}$
REGR_R2(Y,X)	*'Goodness-of-fit'*	r^2
REGR_COUNT(Y,X)	*Sample size*	n (non-null (Y, X) pairs)
REGR_SXX(Y,X)	*Sum of squares* (x)	$\sum x^2 - \frac{(\sum x)^2}{n}$
REGR_SYY(Y,X)	*Sum of squares* (y)	$\sum y^2 - \frac{(\sum y)^2}{n}$
REGR_SXY(Y,X)	*Sum of products*	$\sum xy - \frac{(\sum y)(\sum x)}{n}$

ABS()

Scalar. Returns the absolute value of an expression.

Syntax
ABS(`expression`**)**

Table 62. Data Types

Return	*expression*
Integer	Integer
Long	Long
Float	Float
Interval	Interval

Example
The following example calculates the absolute value of a column from the stream named Devices:

```
INSERT INTO OutStream
SELECT ABS(Devices.Value)
FROM Devices;
```

ACOS()

Scalar. Returns the arccosine of a value.

Syntax
ACOS(*value* **)**

Table 63. Parameter

value	A float between -1 and 1. A value outside this range returns NULL.

Table 64. Data Types

Return	*value*
Float	Float

Example

```
INSERT INTO OutStream
SELECT ACOS(InStream.FloatColumn)
FROM InStream;
```

ASCII()

Scalar. Returns the Unicode code point for a particular character.

Syntax
ASCII(*character* **)**

Table 65. Parameter

character	A character string. The function returns the code point for the first character of this string. If empty or NULL, the function returns NULL.

Table 66. Data Types

Return	*character*
Integer	String

Example

The following example determines the Unicode code point of the first character of a column in the stream InStream. Because only the first character is converted, column contents of both D and Dog are converted to 68.

```
INSERT INTO OutStream
SELECT ASCII(InStream.StringColumn1)
FROM InStream;
```

ASIN()

Scalar. Returns the arcsine of a value.

Syntax

ASIN(*value* **)**

Table 67. Parameter

value	A float between -1 and 1. A value outside this range returns NULL.

Table 68. Data Types

Return	*value*
Float	Float

Example

```
INSERT INTO OutStream
SELECT ASIN(InStream.FloatColumn)
FROM InStream;
```

ATAN()

Scalar. Returns the arctangent of a value.

Syntax

ATAN(*value* **)**

Table 69. Data Types

Return	*value*
Float	Float

Example

```
INSERT INTO OutStream
SELECT ATAN(InStream.FloatColumn)
FROM InStream;
```

ATAN2()

Scalar. Returns the arctangent of the quotient of two values.

Syntax

ATAN2(*dividend, divisor* **)**

Table 70. Data Types

Return	*dividend*	*divisor*
Float	Float	Float

Example

```
INSERT INTO OutStream
SELECT ATAN2(InStream.FloatColumn, 2.1)
FROM InStream;
```

AVG()

Aggregate. Returns the average value of an expression, calculated over the rows in a window

Syntax

AVG([DISTINCT] *expression* **)**

Table 71. Parameter

expression

Table 72. Data Types

Return	*expression*
Float	Integer
	Long
	Float
Interval	Interval
Timestamp	Timestamp

Example
The following example calculates the average of the Price column from the Trades stream over the last 5 minutes:

```
INSERT INTO OutStream
SELECT AVG(Price)
FROM Trades KEEP 5 MINUTES;
```

AVG()

Scalar. Returns the average value of multiple expressions.

Syntax
AVG(*expression, expression* [, ...] **)**

Table 73. Parameters

expression	You must pass at least two parameters, all of the same type. If any parameter is NULL, the function returns NULL.

Table 74. Data Types

Return	*expression*
Float	Integer
	Long
	Float
Interval	Interval
Timestamp	Timestamp

Example
The following example calculates the average of three columns from the stream Orders:

```
INSERT INTO OutStream
SELECT AVG(Orders.line1price, Orders.line2price, Orders.line3Price)
FROM Orders;
```

BITAND()

Scalar. Returns the results of performing a bitwise AND operation on two expressions.

Syntax

BITAND(*expression1, expression2* **)**

Table 75. Data Types

Return	*expression1*	*expression2*
Integer	Integer	Integer
Long	Long	Long

Example

```
INSERT INTO OutStream
SELECT BITAND(InStream.IntColumn, 1010)
FROM InStream;
```

BITCLEAR()

Scalar. Returns the value of an expression after setting a specific bit to zero.

Syntax

BITCLEAR(*expression, bit* **)**

Table 76. Parameters

expression	The initial value.
bit	Which bit to clear, starting from 0 as the least-significant bit.

Table 77. Data Types

Return	*expression*	*bit*
Integer	Integer	Integer
Long	Long	Integer

Examples

```
INSERT INTO OutStream
SELECT BITCLEAR(InStream.IntColumn,2)
FROM InStream;
```

BITFLAG() function

Scalar. Returns a value with all bits set to zero, except for specified bits.

Syntax

BITFLAG(*bit* **) BITFLAGLONG(** *bit* **)**

Table 78. Parameter

bit	Which bit to set, starting from 0 as the least-significant bit.

Table 79. Data Types

Return	*bit*
Integer (BITFLAG)	Integer
Long (BITFLAGLONG)	Integer

Examples

```
INSERT INTO OutStream
SELECT BITMASK(0)
FROM InStream;
```

BITMASK()

Scalar. Returns a value with all bits set to zero except for a specified range of bits.

Syntax

BITMASK(*first, last* **)BITMASKLONG(** *first, last* **)**

Table 80. Parameters

first	The first bit to set, starting from 0 as the least-significant bit.
last	The last bit to set, starting from 0 as the least-significant bit.

Table 81. Data Types

Return	*first*	*last*
Integer (BITMASK)	Integer	Integer
Long (BITMASK-LONG)	Integer	Integer

Example

```
INSERT INTO OutStream
SELECT BITMASK(1, 6)
FROM InStream;
```

BITNOT()

Scalar. Returns the value of an expression with all bits inverted (zeroes set to one and vice-versa).

Syntax
BITNOT(*expression* **)**

Table 82. Data Types

Return	*expression*
Integer	Integer
Long	Long

Example

```
INSERT INTO OutStream
SELECT BITNOT(InStream.IntColumn) AS Negated
FROM InStream;
```

BITOR()

Scalar. Returns the results of performing a bitwise OR operation on two expressions.

Syntax
BITOR(*expression1, expression2* **)**

Table 83. Data Types

Return	*expression1*	*expression2*
Integer	Integer	Integer
Long	Long	Long

Example

```
INSERT INTO OutStream
SELECT BITOR(InStream1.IntColumn, InStream2.IntColumn)
FROM InStream1, InStream2;
```

BITSET()

Scalar. Returns the value of an expression after setting a specific bit to one.

Syntax

BITSET(*expression,bit* **)**

Table 84. Parameters

expression	The initial value.
bit	Which bit to set, starting from 0 as the least-significant bit.

Table 85. Data Types

Return	*expression*	*bit*
Integer	Integer	Integer
Long	Long	Integer

Example

```
INSERT INTO OutStream
SELECT BITSET(InStream.IntColumn,4)
FROM InStream;
```

BITSHIFTLEFT()

Scalar. Returns the value of an expression after shifting the bits left a specific number of positions.

Syntax

BITSHIFTLEFT(*expression, count* **)**

Table 86. Parameters

expression	The initial value.
count	How many positions to shift. The same number of right-most bits are set to zero.

Table 87. Data Types

Return	*expression*	*count*
Integer	Integer	Integer
Long	Long	Integer

Examples

```
INSERT INTO OutStream
SELECT BITSHIFTLEFT(InStream.IntColumn, 3)
FROM InStream;
```

BITSHIFTRIGHT()

Scalar. Returns the value of an expression after shifting the bits right a specific number of positions.

Syntax

BITSHIFTRIGHT(*expression, count* **)**

Table 88. Parameters

expression	The initial value.
count	How many positions to shift. The same number of left-most bits are set to zero.

Table 89. Data Types

Return	*expression*	*count*
Integer	Integer	Integer
Long	Long	Integer

Example

```
INSERT INTO OutStream
SELECT BITSHIFTLEFT(InStream.IntColumn, 5)
FROM InStream;
```

BITTEST()

Scalar. Returns the value of a specific bit in a binary value.

Syntax

BITTEST(*expression, bit* **)**

Table 90. Parameters

expression	The initial value.
bit	Which bit to return. All other bits are set to zero.

Table 91. Data Types

Return	*expression*	*bit*
Integer	Integer	Integer
Long	Long	Integer

Example

```
INSERT INTO OutStream
SELECT BITTEST(InStream.IntColumn,1)
FROM InStream;
```

BITTOGGLE()

Scalar. Returns the value of an expression after inverting the value of a specific bit.

Syntax

BITTOGGLE(*expression,bit* **)**

Table 92. Parameters

expression	The initial value.
bit	Which bit to toggle.

Table 93. Data Types

Return	*expression*	*bit*
Integer	Integer	Integer

Return	*expression*	*bit*
Long	Long	Integer

Example

```
INSERT INTO OutStream
SELECT BITTOGGLE(InStream.IntColumn,0)
FROM InStream;
```

BITXOR()

Scalar. Returns the results of performing a bitwise exclusive or operation on two expressions.

Syntax
BITXOR(*expression1, expression2* **)**

Table 94. Data Types

Return	*expression1*	*expression2*
Integer	Integer	Integer
Long	Long	Long

Example

```
INSERT INTO OutStream
SELECT BITXOR(InStream1.IntColumn, InStream2.IntColumn)
FROM InStream1, InStream2;
```

CEIL()

Scalar. Returns the smallest integer value greater than or equal to a specific value.

Syntax
CEIL(*expression* **)**

Table 95. Data Types

Return	*expression*
Float	Integer
Long	

Return	*expression*
Float	

Example
The following example returns the smallest integer greater than or equal to 3.45:

```
INSERT INTO OutStream
SELECT CEIL(3.45)
FROM Devices;
```

CHR() or CHAR()

Scalar. Returns the characters responding to one or more Unicode code points.

Syntax
CHR(*expression* [, ...] **)** **CHAR(** *expression* [, ...] **)**

Table 96. Parameters

expression	A Unicode code point. An invalid code point, 0, or NULL returns NULL.

Table 97. Data Types

Return	*expression*
String	Integer

Example
The following example inserts a tab into the second column of OutStream.

```
INSERT INTO OutStream
SELECT InStream.Column1, CHR(9)
FROM InStream;
```

COALESCE()

Other. Returns the first non-NULL expression from a list of expressions.

Syntax
COALESCE(*expression* [, ...] **)**

Table 98. Parameters

expression	All parameters must be the same data type. If all parameters are NULL, the function returns NULL.

Table 99. Data Types

Return	*expression*
Blob	Blob
Boolean	Boolean
Float	Float
Integer	Integer
Interval	Interval
Long	Long
String	String
Timestamp	Timestamp
XML	XML

Example

The following example writes the result of the COALESCE function into the PrimAddress column of the Address stream. If the row from the Employee stream contains a non-NULL value in the Home column, that value is written into PrimAddress. If Home is NULL, but Work is non-NULL, the Work value is written into PrimAddress. Otherwise the word 'None' is written into PrimAddress.

```
INSERT INTO Address (PrimAddress)
SELECT COALESCE(Home, Work, 'None')
FROM Employee;
```

CORR()

Aggregate. Returns the correlation coefficient of a set of number pairs. The CORR function is an alias of the CORRCOEF() function.

Syntax

```
CORR(dependent-expression, independent-expression)
```

Table 100. Parameters

dependent-expression	The variable that is affected by the independent variable.
independent-expression	The variable that influences the outcome.

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL.

Both dependent-expression and independent-expression are numeric. The function is applied to the set of (dependent-expression, independent-expression) after eliminating the pairs for which either dependent-expression or independent-expression is NULL. The following computation is made:

```
COVAR_POP ( x, y ) / STDDEV_POP ( x ) * STDDEV_POP ( y )
```

where x represents the dependent-expression and y represents the independent-expression.

SQL/2003 SQL foundation feature outside of core SQL.

The following example performs a correlation to discover whether age is associated with income level.

With a GROUP BY clause:

```
SELECT CORR( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
group by depart_ID
```

With a window:

```
SELECT CORR( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
KEEP 10 ROWS
```

COS()

Scalar. Returns the cosine of a value.

Syntax
COS(*value* **)**

Table 101. Data Types

Return	***value***
Float	Float

Example

```
INSERT INTO OutStream
SELECT COS(InStream.FloatColumn)
FROM InStream;
```

COSD()

Scalar. Returns the cosine of a value, expressed in degrees.

Syntax
COSD(*value* **)**

Table 102. Data Types

Return	*value*
Float	Float

Example

```
INSERT INTO OutStream
SELECT COSD(InStream.FloatColumn)
FROM InStream;
```

COSH()

Scalar. Returns the hyperbolic cosine of a value.

Syntax
COSH(*value* **)**

Table 103. Data Types

Return	*value*
Float	Float

Example

```
INSERT INTO OutStream
SELECT COSH(InStream.FloatColumn)
FROM InStream;
```

COUNT()

Aggregate. Returns the number of rows containing a non-NULL value in a particular column.

Syntax

```
COUNT( { [DISTINCT] expression } | * )
```

Table 104. Parameter

expression	Note that you cannot use DISTINCT with an XML column.

Table 105. Data Types

Return	*expression*
Integer	Integer
	Long
	Float
	Interval
	Timestamp
	String
	Boolean
	Blob
	XML

Usage

COUNT counts the number of rows returned by a query. COUNT(*) returns the number of rows, regardless of the value of those rows and including duplicate values. COUNT(expression) returns the number of non-NULL values for that expression returned by the query. For example, COUNT(column_A) returns the number of non-NULL values in column_A. If all input rows submitted to COUNT have a value of NULL, COUNT returns a value of 0. COUNT never returns NULL.

Examples

```
INSERT INTO OutStream
SELECT COUNT(*)
FROM DeviceExceptions;
```

```
INSERT INTO OutStream
```

```
SELECT COUNT(commission_pct)
FROM SalesOrders;
```

```
INSERT INTO OutStream
SELECT COUNT(DISTINCT province_id)
FROM Contestants;
```

COVAR_POP()

Aggregate. Returns the population covariance of a set of number pairs.

Syntax

```
COVAR_POP( dependent-expression, independent-expression )
```

Table 106. Parameters

dependent-expression	The variable that is affected by the independent variable.
independent-expression	The variable that influences the outcome.

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL. Both dependent-expression and independent-expression are numeric. The function is applied to the set of (dependent-expression, independent-expression) pairs after eliminating all pairs for which either dependent-expression or independent-expression is NULL. The following computation is then made:

```
(SUM( x * y ) - SUM( y ) * SUM( y ) / n ) / n
```

where x represents the dependent-expression and y represents the independent-expression.

SQL/2003 SQL foundation feature (T621) outside of core SQL.

The following example measures the strength of association between employees' age and salary.

With a GROUP BY clause:

```
SELECT COVAR_POP( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
group by depart_ID
```

With a window:

```
SELECT COVAR_POP( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
KEEP 10 ROWS
```

COVAR_SAMP()

Aggregate. Returns the sample covariance of a set of number pairs.

Syntax

```
COVAR_SAMP( dependent-expression, independent-expression )
```

Table 107. Parameters

dependent-expression	The variable that is affected by the independent variable.	
independent-expression	The variable that influences the outcome.	

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL. Both dependent-expression and independent-expression are numeric. The function is applied to the set of (dependent-expression, independent-expression) pairs after eliminating all pairs for which either dependent-expression or independent-expression is NULL.

SQL/2003 SQL foundation feature (T621) outside of core SQL.

The following example measures the strength of association between employees' age and salary.

With a GROUP BY clause:

```
SELECT COVAR_SAMP( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
group by depart_ID
```

With a window:

```
SELECT COVAR_SAMP( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
KEEP 10 ROWS
```

DATECEILING()

Scalar. Computes a new timestamp based on the provided timesamp, multiple and date_part arguments, with subordinate parts set to zero. The result is then rounded up to the minimum date_part multiple that is greater than or equal to the input timestamp.

Syntax

```
DATECEILING( date_part, expression [, multiple] )
```

Table 108. Parameters

date_part	Keyword that identifies the granularity desired. The valid keywords are identical to the date parts supported for the existing function, datepart().
expression	Date-time expression that contains the value to be evaluated.
multiple	Contains a multiple of date_parts to be used in the operation. If supplied this should be a non-zero positive integer value. If none is provided or it is NULL, this is assumed to be 1.

Usage

This function determines the next largest date_part value expressed in the timestamp, and zeroes out all date_parts of finer granularity than date_part.

Date_part is a keyword, expression is any expression that evaluates or can be implicitly converted to a datetime (or timestamp) datatype, and multiple is an integer containing the multiples of date_parts to be used in performing the ceiling operation. For example, to establish a date ceiling based on 10 minute intervals, use MINUTE or MI for the date_part, and 10 as the multiple.

Known errors:

- The server generates an "invalid argument" error if the value of the required arguments evaluate to NULL.
- The server generates an "invalid argument" error if the value of the multiple argument is not within a range valid for the specified datepart argument. As an example, have the value of multiple be less than 60 if date_part mi is specified.

Standards and compatibility

Sybase extension.

Example

```
DATECEILING( MINUTE, "August 13, 2008 10:35.123AM")
returns "August 13, 2008 10:36.000AM"
```

DATEFLOOR()

Scalar. Computes a new timestamp based on the provided timestamp, multiple and date_part arguments, with subordinate parts set to zero. The result is then rounded down to the maximum date_part multiple that is less than or equal to the input timestamp.

Syntax

```
DATEFLOOR( date_part, expression [, multiple] )
```

Table 109. Parameters

date_part	Keyword that identifies the granularity desired. The valid keywords are identical to the date parts supported for the existing function, datepart().
expression	A date-time expression that contains the value to be evaluated.
multiple	Contains a multiple of date_parts to be used in the operation, which if supplied must be a non-zero positive integer value. If none is provided or it is NULL, this is assumed to be 1.

Valid date part keywords and their valid abbreviation keywords are:

```
Year             yy
Quarter         qq
Month             mm
Week             wk
day               dd
hour              hh
minute          mi
second          ss
millisecond         ms
```

Usage

This function zeroes out all datetime values with a granularity finer than that specified by date_part. Date_part is a keyword, and expression is any expression that evaluates or can be implicitly converted to a datetime (or timestamp) datatype. Multiple is an integer that contains the multiples of date_parts to be used in performing the floor operation. For example, to establish a date floor based on 10 minute intervals, use MINUTE or MI for date_part, and 10 as the multiple.

Known errors:

- The server generates an "invalid argument" error if the value of the required arguments evaluate to NULL.
- The server generates an "invalid argument" error if the value of the multiple argument is not within a range valid for the specified datepart argument. As an example, have the value of multiple be less than 60 if date_part mi is specified.

Standards and compatibility

Sybase extension.

Example

```
DATEFLOOR( MINUTE, "August 13, 2008 10:35.123AM")
returns "August 13, 2008 10:35.000AM"
```

DATEROUND()

Scalar. Computes a new timestamp based on the provided timestamp, multiple and date_part arguments, with subordinate date_parts set to zero. The result is then rounded to the value of a date_part multiple that is nearest to the input timestamp.

Syntax

```
DATEROUND( date_part, expression [, multiple] )
```

Table 110. Parameters

date_part	Keyword identifying the granularity desired, where the valid keywords are identical to the date parts supported for the existing function, datepart().
expression	A date-time expression that contains the value to be evaluated.
multiple	Contains a multiple of date_parts to be used in the operation. If supplied, this needs to be a non-zero positive integer value. If none is provided or is NULL, this is assumed to be 1.

Usage

This function rounds the datetime value to the nearest date_part or multiple of date_part, and zeroes out all date_parts of finer granularity than date_part or its multiple. For example, when rounding to the nearest hour, the minutes portion is determined, and if >= 30, then the hour portion is incremented by 1, and the minutes and other subordinate date parts are zeroes.

Date_part is a keyword, expression is any expression that evaluates or can be implicitly converted to a datetime (or timestamp) datatype, and multiple is an integer containing the multiples of date_parts to be used in performing the rounding operation. For example, to round to the nearest 10-minute increment, use MINUTE or MI for date_part, and 10 as the multiple.

Known errors:

- The server generates an "invalid argument" error if the value of the required arguments evaluate to NULL.
- The server generates an "invalid argument" error if the value of the multiple argument is not within a range valid for the specified datepart argument. As an example, the value of multiple must be less than 60 if date_part mi is specified.

Standards and compatibility

Sybase extension.

Example

```
DATEROUND( MINUTE,  "August 13, 2008 10:35.500AM")
returns "August 13, 2008 10:36.000AM"
```

DAYOFMONTH()

Scalar. Returns an integer representing the day of the month as extracted from a timestamp value.

Syntax

DAYOFMONTH(*timestamp* [, *timezone*])

Table 111. Parameters

timestamp	An expression that evaluates to a timestamp value.
timezone	A string representing the time zone. If omitted, Sybase CEP Engine assumes the local time zone. See "Sybase CEP Time Zone Database" in the Sybase CEP Integration Guide for more information about valid time zone strings.

Table 112. Data Types

Return	*timestamp*	*timezone*
Integer	Timestamp	String

Usage

If either parameter is NULL, the function returns NULL.

Example

```
INSERT INTO OutStream
SELECT DAYOFMONTH(InStream.OrderTime)
FROM InStream;
```

DAYOFWEEK()

Scalar. Returns an integer representing the day of the week (1 is Sunday) as extracted from a timestamp value.

Syntax

DAYOFWEEK(*timestamp* [, *timezone*])

Table 113. Parameters

timestamp	An expression that evaluates to a timestamp value.
timezone	A string representing the time zone. If omitted, Sybase CEP Engine assumes the local time zone. See "Sybase CEP Time Zone Database" in the Sybase CEP Integration Guide for more information about valid time zone strings.

Table 114. Data Types

Return	*timestamp*	*timezone*
Integer	Timestamp	String

Usage

If either parameter is NULL, the function returns NULL.

Example

```
INSERT INTO OutStream
SELECT DAYOFWEEK(InStream.OrderTime)
FROM InStream;
```

DAYOFYEAR()

Scalar. Returns an integer representing the day of the year (one-based) as extracted from a timestamp value.

Syntax

DAYOFYEAR(*timestamp* [, *timezone*])

Table 115. Parameters

timestamp	An expression that evaluates to a timestamp value.
timezone	A string representing the time zone. If omitted, Sybase CEP Engine assumes the local time zone. See "Sybase CEP Time Zone Database" in the Sybase CEP Integration Guide for more information about valid time zone strings.

Table 116. Data Types

Return	*timestamp*	*timezone*
Integer	Timestamp	String

Usage

If either parameter is NULL, the function returns NULL.

Example

```
INSERT INTO OutStream
SELECT DAYOFYEAR(InStream.OderTime)
FROM InStream;
```

DISTANCE()

Scalar. Returns a number representing the distance between two points in either two or three dimensions.

Syntax

DISTANCE(*point1x, point1y [, point1z], point2x, point2y [, point2z]*)

Table 117. Parameters

point1x	An expression that evaluates to a value representing the position of the first point on the x axis.
point1y	An expression that evaluates to a value representing the position of the first point on the y axis.
point1z	An expression that evaluates to a value representing the position of the first point on the z axis.
point2x	An expression that evaluates to a value representing the position of the second point on the x axis.
point2y	An expression that evaluates to a value representing the position of the second point on the y axis.
point2z	An expression that evaluates to a value representing the position of the second point on the z axis.

Table 118. Data Types

Return	*point1x*	*point1y*	*point1z*	*point2x*	*point2y*	*point2z*
Integer	Integer	Integer	Integer	Integer	Integer	Integer
Long	Long	Long	Long	Long	Long	Long
Float	Float	Float	Float	Float	Float	Float

Example

```
INSERT INTO OutStream
```

```
SELECT DISTANCE(4.7, 5.75, 1.2, 2.1, 6.33, 9.6)
FROM InStream;
```

DISTANCESQUARED()

Scalar. Returns a number representing the square of the distance between two points in either two or three dimensions.

Syntax

DISTANCESQUARED(*point1x, point1y* [*, point1z*]*, point2x, point2y* [*, point2z*])

Table 119. Parameters

point1x	An expression that evaluates to a value representing the position of the first point on the x axis.
point1y	An expression that evaluates to a value representing the position of the first point on the y axis.
point1z	An expression that evaluates to a value representing the position of the first point on the z axis.
point2x	An expression that evaluates to a value representing the position of the second point on the x axis.
point2y	An expression that evaluates to a value representing the position of the second point on the y axis.
point2z	An expression that evaluates to a value representing the position of the second point on the z axis.

Table 120. Data Types

Return	*point1x*	*point1y*	*point1z*	*point2x*	*point2y*	*point2z*
Integer	Integer	Integer	Integer	Integer	Integer	Integer
Long	Long	Long	Long	Long	Long	Long
Float	Float	Float	Float	Float	Float	Float

Example

```
INSERT INTO OutStream
SELECT DISTANCESQUARED(4.7, 5.75, 2.1, 6.33)
FROM InStream;
```

EXP()

Scalar. Returns a number representing the value of e raised to a specific exponent.

Syntax
EXP(*exponent* **)**

Table 121. Data Types

Return	*exponent*
Float	Float

Example

```
INSERT INTO OutStream
SELECT EXP(InStream.FloatColumn)
FROM InStream;
```

EXP_WEIGHTED_AVG()

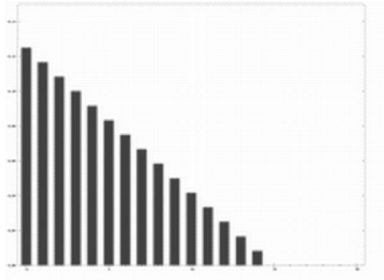
Aggregate. The EXP_WEIGHTED_AVG function calculates an exponential weighted average.

Syntax
`EXP_WEIGHTED_AVG( expression, period-expression )`

Table 122. Parameters

expression	a numeric expression for which a weighted value is to be computed
period-expression	a numeric expression specifying the period for which the average is to be computed

An exponential moving average (EMA), sometimes also called an exponentially weighted moving average (EWMA), applies weighting factors which decrease exponentially. The weighting for each older data point decreases exponentially, giving much more importance to recent observations while still not discarding older observations entirely. The graph at right shows an example of the weight decrease.



The degree of weighing decrease is expressed as a constant smoothing factor α, a number between 0 and 1. α may be expressed as a percentage, so a smoothing factor of 10% is equivalent to α=0.1. Alternatively, α may be expressed in terms of N time periods, where. For example,

$$\alpha = \frac{2}{N+1}$$

N=19 is equivalent to α=0.1.

The observation at a time period t is designated Yt, and the value of the EMA at any time period t is designated St. S1 is undefined. S2 may be initialized in a number of different ways, most commonly by setting S2 to Y1, though other techniques exist, such as setting S2 to an average of the first 4 or 5 observations. The prominence of the S2 initialization's effect on the resultant moving average depends on α; smaller α values make the choice of S2 relatively more important than larger α values, since a higher α discounts older observations faster.

This type of moving average reacts faster to recent price changes than a simple moving average. The 12- and 26-day EMAs are the most popular short-term averages, and they are used to create indicators like the moving average convergence divergence (MACD) and the percentage price oscillator (PPO). In general, the 50- and 200-day EMAs are used as signals of long-term trends.

Sybase extension.

The following example demonstrates how to use the EXP_WEIGHTED_AVG function to calculate exponential weighted averages using data from a CSV file:

```
-- create input stream schema
CREATE SCHEMA InSchema (
    id STRING,
    x FLOAT,
    y FLOAT
);

-- create output stream schema
CREATE SCHEMA OutSchema (
    id STRING,
    x FLOAT,
    y FLOAT,
```

```
    exp_weighted_avg_result FLOAT,
);

-- create input stream
CREATE INPUT STREAM StreamIn
SCHEMA InSchema;

-- create master window
CREATE MASTER WINDOW ResultWindow
SCHEMA OutSchema
KEEP 5 ROWS
;

-- create output stream
CREATE OUTPUT STREAM StreamOut
SCHEMA OutSchema;

-- input stream read data from csv file by ReadFromCsvFileAdapterType
adapter
ATTACH INPUT ADAPTER ReadFromCSVFile TYPE ReadFromCsvFileAdapterType
TO STREAM StreamIn
PROPERTIES
    FILENAME            = "$ProjectFolder\..\data\data.csv",
    TITLEROW            = "false",
    TIMESTAMPCOLUMN     = "false",
    RATE                = "1",
    USECURRENTTIMESTAMP = "true"
;

-- output stream write data to csv file by WriteToCsvFileAdapterType
adapter
ATTACH OUTPUT ADAPTER WriteToCSVFile TYPE WriteToCsvFileAdapterType
TO STREAM StreamOut
PROPERTIES
    FILENAME = "$ProjectFolder\..\data\result.csv"
;

-- insert the calculated result to window
INSERT INTO ResultWindow
SELECT id,x,y,
exp_weighted_avg(x, y)
FROM StreamIn
KEEP 5 ROWS;

INSERT INTO StreamOut
SELECT *
FROM ResultWindow;
```

EXTRACT()

Scalar. Returns a subset of the bytes from a BLOB value.

Syntax

EXTRACT(*blob, start, count* **)**

Table 123. Parameters

blob	The BLOB from which to extract bytes.
start	The starting byte (one-based) to extract from blob.
count	The number of bytes to extract.

Table 124. Data Types

Return	*blob*	*start*	*count*
BLOB	BLOB	Integer	Integer

Example

The following example returns the BLOB without its first character:

```
INSERT INTO OutStream
SELECT Extract(A.BlobObject, 2, Length(A.BlobObject)-1)
FROM A;
```

FIRST()

Other. Returns the value of a column from a specific row in a window.

Syntax

FIRST(*column* [*, offset*]) **GETTIMESTAMP(FIRST(** *name*))

Table 125. Parameters

column	The name of a column in a window.
offset	Which row to use, as offset from the first row in the window based on the window's sort order. If omitted or 0, uses the first row.
name	The name of a stream or window.

Table 126. Data Types

Return	*column*	*offset*
Integer	Integer	Integer
Long	Long	Integer
Float	Float	Integer
Interval	Interval	Integer
Timestamp	Timestamp	Integer
String	String	Integer
Boolean	Boolean	Integer
BLOB	BLOB	Integer
XML	XML	Integer

Return	*name*
Timestamp	String

Usage

An offset of 0 for a window that is not sorted by largest or smallest value indicates the oldest row in the window (the first to arrive), while an offset of 1 indicates the next oldest row, and so on.

The offset for a window defined with a LARGEST or SMALLEST clause is relative to the sorting order as specified by the LARGEST or SMALLEST clause. For example, if you define a window to keep the largest values as sorted by its Price column, then FIRST(Volume) returns the value of the Volume column from the row in the window with the lowest value in its Price column. FIRST(Volume, 1) returns the value of the Volume column from the row containing the second-lowest value in the Price column, and so on.

The offset for an unnamed window partitioned by a GROUP BY clause, or a window partitioned by one or more PER clauses, is based on the partition to which the incoming row belongs, not on the entire window.

For example, this following scenario assumes you have a window grouped by its Symbol column and your query includes FIRST(Volume). When a row arrives in the window, FIRST(Volume) returns the value of the Volume column from the oldest row with a value in its Symbol column that matches the value in the new row.

When used with a join, Sybase CEP Engine always evaluates FIRST() in the context of the row or rows being processed. In other words, the FIRST function applies to the partition identified by the row or rows being processed. If the row is Null-extended as the result of an outer join, the partition is the Null partition.

A special case of the FIRST function is embedded in the GETTIMESTAMP function and returns the timestamp of the first row in the specified window.

FIRST() can be used in a WHERE selection condition and in a SELECT clause select list.

Example
The following example returns the price of the third-oldest trade record in the window:

```
INSERT INTO OutStream
SELECT FIRST(Trades.Price, 2)
FROM Trades;
```

FIRST_VALUE()

Aggregate. Returns first value from an ordered set of values. FIRST_VALUE() is an alias for the FIRST() function in the Sybase CEP compiler.

Syntax
```
FIRST_VALUE( expression [IGNORE NULLS] )
```

Table 127. Parameters

expression	The expression on which to determine the first value in an ordered set.

FIRST_VALUE returns the first value in an ordered set of values. If the first value in the set is null, then the function returns NULL unless you specify IGNORE NULLS. If you specify IGNORE NULLS, then FIRST_VALUE returns the first non-null value in the set, or NULL if all values are null. You cannot use FIRST_VALUE or any other analytic function for expression. That is, you cannot nest analytic functions, but you can use other built-in function expressions for expression.

Sybase extension.

The following example computes the tick-by-tick open, close, minimum, and maximum values for trades for a one-minute interval:

```
INSERT INTO OutOpenCloseMinMax
SELECT
      FIRST(Price) as OpenPrice
FROM
    MyWindow
;

INSERT INTO OutOpenCloseMinMax2
SELECT

      LAST(Price) as OpenPrice,
      FIRST_VALUE(Price IGNORE NULLS) as OpenPrice2,
```

```
      GETTIMESTAMP(LAST(MyWindow)) as t1

FROM
    MyWindow
```

FLOOR()

Scalar. Returns the largest integer value less than or equal to a specific value.

Syntax
FLOOR(*expression* **)**

Table 128. Data Types

Return	*expression*
Float	Integer
	Long
	Float

Example
The following example returns the largest integer less than or equal to 3.45:

```
INSERT INTO OutStream
SELECT FLOOR(3.45)
FROM Devices;
```

GET___COLUMNBYNAME()

Other. Returns the value of a column identified by an expression evaluated at runtime.

Syntax
GETBLOBCOLUMNBYNAME(*name,column* **)**
GETBOOLEANCOLUMNBYNAME(*name,column* **)**
GETFLOATCOLUMNBYNAME(*name,column* **)**
GETINTEGERCOLUMNBYNAME(*name,column* **)**
GETINTERVALCOLUMNBYNAME(*name,column* **)**
GETLONGCOLUMNBYNAME(*name,column* **)**
GETSTRINGCOLUMNBYNAME(*name,column* **)**
GETTIMESTAMPCOLUMNBYNAME(*name column* **)**
GETXMLCOLUMNBYNAME(*name,column* **)**

Table 129. Parameters

name	The name of a stream or window included as a data source in the query's FROM clause.
column	An expression that evaluates at runtime to the name of a column of the type specified in the name of the function, in the stream or window specified with name. If NULL or the specified column doesn't exist, the function returns NULL and generates a warning message.

Table 130. Data Types

Return	*name*	*column*
As specified in the function name	String	String

Examples

In the following example, the WinningPhotos stream includes several BLOB columns, each of which contains photos from one of three artists in a given category. A Winner STRING column specifies the column that contains the image that won the category prize in each row. The OutStream stream captures only the winning photos by capturing only the contents of whichever column contains the winning image.

```
INSERT INTO OutStream
SELECT GETBLOBCOLUMNBYNAME(WinningPhotos, Winner)
FROM WinningPhotos;
```

In the following example, each row of the TestAnswers stream includes several BOOLEAN columns, which contain true or false answers. All answers except one in each row are false. The Correct column specifies which column contains the true answer for the row. The correct answers are published to the OutStream column based on the values contained in the Correct column.

```
INSERT INTO OutStream
SELECT GETBOOLEANCOLUMNBYNAME(TestAnswers, Correct)
FROM TestAnswers;
```

In the following example, each row of the Prices stream includes an item listing along with a range of prices. The Preferred column specifies which column contains the price preferred by the buyer for the item in question. The preferred prices are published to OutStream:

```
INSERT INTO OutStream
SELECT GETFLOATCOLUMNBYNAME(Prices, Preferred)
FROM Prices;
```

In the following example, each row of the ContactInformation stream includes a listing for each contact's home, business, and mobile phone numbers. The PrimaryPhone column specifies which of these three numbers is considered the main number for the contact, by listing the column name in which the number appears:

```
INSERT INTO OutStream
SELECT GETINTEGERCOLUMNBYNAME(ContactInformation, PrimaryPhone)
FROM ContactInformation;
```

In the following example, each row of the stream UtilitySettings includes several INTERVAL columns listing possible wait times associated with a utility. The preferred wait time is listed as a reference to the appropriate column name in the PreferredWaitTime column. The value associated with the specified column is published to OutStream:

```
INSERT INTO OutStream
SELECT GETINTERVALCOLUMNBYNAME(UtilitySettings, PreferredWaitTime)
FROM UtilitySettings;
```

In the following example, each row of InStream includes several LONG columns. Each row also contains a Choice column listing the name of one of the LONG columns. The value associated with the specified column is published to OutStream:

```
INSERT INTO OutStream
SELECT GETLONGCOLUMNBYNAME(InStream, Choice)
FROM InStream;
```

GETPREFERENCE___()

Other. Returns the value of a preference as set in the configuration file c8-server.conf.

Syntax

GETPREFERENCEBOOLEAN(*section, preference* [*, default*] **)**
GETPREFERENCEINTEGER(*section, preference* [*, default*] **)**
GETPREFERENCELONG(*section, preference* [*, default*] **)**
GETPREFERENCESTRING(*section, preference* [*, default*] **)**

Table 131. Parameters

section	The full path to the section containing the preference. See "Configuring Sybase CEP Engine" in the *Sybase CEP Installation Guide* for more information about the configuration file and the preference settings in it.
name	The name of the preference.

default	Return this value if the specified preference is not found. If omitted and the preference is not found, the function returns FALSE, 0, or the empty string, depending on the return type.

Table 132. Data Types

Return	*section*	*preference*	*default*
As specified in the function name	String	String	As specified in the function name.

Examples

In the following example, the GETPREFERENCEBOOLEAN function retrieves the value for the "EnableSSL" preference from the "SSL" subsection of the "Sybase/C8/Security" section of c8-server.conf. If no "EnableSSL" preference is specified, the function returns FALSE:

```
INSERT INTO OutStream
SELECT Hostname, GETPREFERENCEBOOLEAN("C8/Security/SSL",
"EnableSSL")
FROM InStream;
```

In the following example, the GETPREFERENCEINTEGER function retrieves the value for the "LoadLimit" preference from the "Container" subsection of the "Sybase/C8/Server" section of the file c8-server.conf. If no "LoadLimit" preference is specified, the function returns 0. Since 0 is also the default setting for this preference, no additional default value is specified:

```
INSERT INTO OutStream
SELECT Hostname, GETPREFERENCEINTEGER("C8/Server/Container",
"LoadLimit")
FROM InStream;
```

In the following example, the GETPREFERENCELONG function retrieves the value for the "SmallBlocksThreshold" preference of the "Sybase/C8/Memory" section of c8-server.conf. If no "SmallBlocksThreshold" preference is specified, the default value of 256 is returned:

```
INSERT INTO OutStream
SELECT Hostname, GETPREFERENCELONG("C8/Memory",
"SmallBlocksThreshold", 256)
FROM InStream;
```

In the following example, the GETPREFERENCESTRING function retrieves the value for the "NodeURI" preference, which resides in the "ManagerCluster" subsection of the "HighAvailability" subsection of the "Manager" subsection "Sybase/C8/Server" section of c8-server.conf. If no "NodeURI" preference is specified, the function returns the message "NOT SPECIFIED":

```
INSERT INTO OutStream
SELECT Hostname, GETPREFERENCEINTEGER("C8/Server/Manager/
HighAvailability/ManagerCluster",
     "NodeURI", "NOT SPECIFIED")
FROM InStream;
```

GETTIMESTAMP()

Scalar. Returns the timestamp of a row.

Syntax

GETTIMESTAMP(*name* **)** **GETTIMESTAMP(FIRST(** *name* **))**
GETTIMESTAMP(LAST(*name* **))** **GETTIMESTAMP(PREV(** *name* **))**
GETTIMESTAMP()

Table 133. Parameters

name	The name of a stream or window.

Table 134. Data Types

Return	*name*
Timestamp	String

Usage

Using this function with an embedded FIRST or LAST function returns the timestamp of the first or last row of a window, respectively, based on the window's sort order. Using it with an embedded PREV function returns the timestamp of the immediately previous row of a stream or window.

You can only use this function without an argument as part of a filter expression when subscribing to a stream or window. See "Out-of-process Adapter" in the Sybase CEP *Integration Guide* for more information.

Example

```
INSERT INTO OutStream
SELECT COUNT(*)
FROM InputWindow
WHERE GETTIMESTAMP(InputWindow) >= (NOW() - 5 SECONDS);
```

HOUR()

Scalar. Returns an integer representing the hour of the day (0 to 23) as extracted from a timestamp value

Syntax

HOUR(*timestamp* [*, timezone*] **)**

Table 135. Parameters

timestamp	An expression that evaluates to a timestamp value.
timezone	A string representing the time zone. If omitted, Sybase CEP Engine assumes the local time zone. See "Sybase CEP Time Zone Database" in the Sybase CEP Integration Guide for more information about valid time zone strings.

Table 136. Data Types

Return	*timestamp*	*timezone*
Integer	Timestamp	String

Usage

If either parameter is NULL, the function returns NULL.

Example

```
INSERT INTO OutStream
SELECT HOUR(InStream.OrderTime)
FROM InStream;
```

INSTR()

Scalar. Returns the starting position (one-based) of a string within another string.

Syntax

INSTR(*string, substring* [*, start* [*, occurrence*]]**)**

Table 137. Parameters

string	A string literal or the name of a String column.
substring	The string to locate, either a literal or the name of a String column. If the string is not found, the function returns 0.

start	The character position within string to start searching. If negative, the search starts at the end of string and proceeds backwards. If greater than the size of string, the function returns 0. If omitted, the search starts with the first character.
occurrence	Which occurrence of substring to locate. If omitted, the search looks for the first occurrence.

Table 138. Data Types

Return	*string*	*substring*	*start*	*occurrence*
Integer	String	String	Integer	Integer

Example
The following example uses the INSTR function to get the position of the value of the Prefix column in the value of the Title column:

```
INSERT INTO OutStream
SELECT INSTR(Devices.Title, Devices.Prefix)
FROM Devices;
```

LAST()

Other. Returns the value of a column from a specific row in a window.

Syntax
LAST(*column* [*, offset*]) **GETTIMESTAMP(LAST(** *name* **))**

Table 139. Parameters

column	The name of a column in a window.
offset	Which row to use, as offset from the last row in the window based on the window's sort order. If omitted or 0, uses the last row.
name	The name of a stream or window.

Table 140. Data Types

Return	*column*	*offset*
Integer	Integer	Integer
Long	Long	Integer
Float	Float	Integer

Return	*column*	*offset*
Interval	Interval	Integer
Timestamp	Timestamp	Integer
String	String	Integer
Boolean	Boolean	Integer
BLOB	BLOB	Integer
XML	XML	Integer

Return	*name*
Timestamp	String

Usage

An offset of 0 for a window that is not sorted by largest or smallest value indicates the most recent row in the window (the last to arrive), an offset of 1 indicates the next newest row, and so on.

The offset for a window defined with a LARGEST or SMALLEST clause is relative to the sorting order as specified by the LARGEST or SMALLEST clause. For example, if you define a window to keep the largest values as sorted by its Price column, then LAST(Volume) returns the value of the Volume column from the row in the window with the highest value in its Price column. LAST(Volume, 1) returns the value of the Volume column from the row containing the second-highest value in the Price column, and so on.

The offset for an unnamed window partitioned by a GROUP BY clause, or a window partitioned by one or more PER clauses, is based on the partition to which the incoming row belongs, not on the entire window. For example, say you have a window grouped by its Symbol column and your query includes LAST(Volume). When a row arrives in the window, LAST(Volume) returns the value of the Volume column from the newest row with a value in its Symbol column that matches the value in the new row.

When used with a join, Sybase CEP Engine always evaluates LAST in the context of the row or rows being processed. The LAST function applies to the partition identified by the row or rows being processed. If the row is Null-extended as the result of an outer join, the partition is the Null partition.

A special case of the LAST function is embedded in the GETTIMESTAMP function and returns the timestamp of the last row in the specified window.

You can use LAST() in a WHERE selection condition and a SELECT clause select list.

The LAST function is semantically identical to the [ ] row operator, when [ ] is used with windows, and is similar, but not identical, to the PREV function. For more information about

the [ ] row operator, see Row CCL Operators. For more information about the PREV function, see PREV().

Example
The following example returns value of the Price column from the last row in the window:

```
INSERT INTO OutStream
SELECT LAST(Trades.Price, 0)
FROM Trades;
```

LAST_VALUE()

Aggregate. Returns the last value from an ordered set of values. LAST_VALUE() is an alias for the LAST() function in the Sybase CEP compiler.

Syntax
```
LAST_VALUE( expression )
```

Table 141. Parameters

expression	The expression on which to determine the last value in an ordered set.

LAST_VALUE returns the last value in an ordered set of values. If the last value in the set is null, then the function returns NULL unless you specify IGNORE NULLS. If you specify IGNORE NULLS, then LAST_VALUE returns the last non-null value in the set, or NULL if all values are null.

You cannot use LAST_VALUE or any other analytic function as an expression. That is, you cannot nest analytic functions, but you can use other built-in function expressions for expression.

Sybase extension.

The following example computes the tick-by-tick open, close, minimum, and maximum values for trades for a one-minute interval:

```
INSERT INTO OutOpenCloseMinMax2
SELECT

      LAST_VALUE(Price) as OpenPrice,
      FIRST_VALUE(Price IGNORE NULLS) as OpenPrice2,
      GETTIMESTAMP(LAST_VALUE(MyWindow)) as t1

FROM
    MyWindow
```

LEFT()

Scalar. Returns a specified number of characters from the beginning of a given string.

Syntax
LEFT(*string, count* **)**

Table 142. Parameters

string	The string.
count	The number of characters to return.

Table 143. Data Types

Return	*string*	*count*
String	String	Integer

Example
The following example uses the LEFT function to select the first ten characters of the value in the column StockName:

```
INSERT INTO OutStream
SELECT LEFT(Trades.StockName,10)
FROM Trades;
```

LENGTH()

Scalar. Returns the length of a given string or BLOB, in characters or bytes, respectively.

Syntax
LENGTH(value)

Table 144. Data Types

Return	*value*
Integer	String BLOB

Example
The following example returns the length of the string found in Trades.StockName:

```
INSERT INTO OutStream
```

```
SELECT LENGTH(Trades.StockName)
FROM Trades;
```

LN()

Scalar. Returns the natural logarithm of a given value.

Syntax

LN(*expression* **)**

Table 145. Parameter

expression	An expression that evaluates to a value greater than or equal to 0.

Table 146. Data Types

Return	*expression*
Float	Float

Example

```
INSERT INTO OutStream
SELECT LN(InStream.FloatColumn)
FROM InStream;
```

LOG()

Scalar. Returns the logarithm of a given value to a specified base.

Syntax

LOG(*base, value* **)**

Table 147. Parameters

base	An expression that evaluates to a value greater than 1.
value	An expression that evaluates to a value greater than or equal to 0.

Table 148. Data Types

Return	*base*	*value*
Float	Float	Float

Example

```
INSERT INTO OutStream
SELECT LOG(InStream.FBase, InStream.FValue)
FROM InStream;
```

LOG10()

Scalar. Returns the logarithm of a given value to the base 10.

Syntax
LOG10(*value* **)**

Table 149. Parameter

value	An expression that evaluates to a value greater than or equal to 0.

Table 150. Data Types

Return	*value*
Float	Float

Example

```
INSERT INTO OutStream
SELECT LOG10(InStream.FloatColumn)
FROM InStream;
```

LOG2()

Scalar. Returns the logarithm of a given value to the base 2.

Syntax
LOG2(*value* **)**

Table 151. Parameter

value	An expression that evaluates to a value greater than or equal to 0.

Table 152. Data Types

Return	*value*
Float	Float

Example

```
INSERT INTO OutStream
SELECT LOG2(InStream.FloatColumn)
FROM InStream;
```

LOWER()

Scalar. Returns a given string with all characters converted to lower case.

Syntax

LOWER(*string* **)**

Table 153. Data Types

Return	*string*
String	String

Example

The following example uses the LOWER function to convert a stock name to all lowercase characters.

```
INSERT INTO OutStream
SELECT LOWER(Trades.StockName)
FROM Trades;
```

LTRIM()

Scalar. Returns a given string with all leading spaces removed.

Syntax

LTRIM(*string* **)**

Table 154. Data Types

Return	*string*
String	String

Example

The following example uses the LTRIM function to remove leading spaces:

```
INSERT INTO OutStream
SELECT LTRIM(Trades.Symbol)
FROM Trades;
```

MAKETIMESTAMP()

Scalar. Constructs a timestamp.

Syntax

MAKETIMESTAMP(*year, month, day, hour, minute, second, microsecond* [*, timezone*] **)**

Table 155. Parameters

year	An expression that evaluates to a value from 0001 to 9999. Values outside of the range 1970 to 2099 may result in inaccuracies due to leap years and daylight savings time.
month	An expression that evaluates to a value specifying the month. 0-12 indicate January to December, with both 0 and 1 representing January. Values larger than 12 roll over into subsequent years, while negative values subtract months from January of the specified year.
day	An expression that evaluates to a value specifying the day of the month. 0 and 1 both represent the first day of the year. Values larger than the valid number of days for the specified month roll over into subsequent months, while negative values subtract days from the first day of the specified month.

hour	An expression that evaluates to a value specifying the hour of the day. Values larger than 23 roll over into subsequent days, while negative values subtract hours from midnight of the specified day.
minute	An expression that evaluates to a value specifying the minute. Values larger than 59 roll over into subsequent hours, while negative values subtract minutes from the specified hour.
second	An expression that evaluates to a value specifying the second. Values larger than 59 roll over into subsequent minutes, while negative values subtract seconds from the specified minute.
microsecond	An expression that evaluates to a value specifying the microsecond. Values larger than 999999 roll over into subsequent seconds, while negative values subtract microseconds from the specified second.
timezone	A string representing the time zone. If omitted, assumes the local time zone. See "Sybase CEP Time Zone Database" in the Sybase CEP *Integration Guide* for more information about valid time zone strings.

Table 156. Data Types

Return	*year*	*month*	*day*	*hour*	*minute*	*second*	*microsecond*	*timezone*
Timestamp	Integer	Integer	Integer	Integer	Integer	Integer	Integer	String

Usage

If any argument is Null, the function returns Null.

Example

```
INSERT INTO OutStream
SELECT MAKETIMESTAMP(2007,5,15,14,30,0,0)
FROM InStream;
```

MAX()

Aggregate. Returns the maximum value of a given expression over multiple rows.

Syntax
MAX(*expression* **)**

Table 157. Data Types

Return	*expression*
Integer	Integer
Long	Long
Float	Float
Interval	Interval
Timestamp	Timestamp
String	String
BLOB	BLOB

Usage
DISTINCT is permitted, but has no effect because the maximum of the distinct values is the same as the maximum of all values.

Example
The following example calculates the maximum value of the Price column over the last five minutes:

```
INSERT INTO OutStream
SELECT MAX(Trades.Price)
FROM Trades KEEP 5 MINUTES;
```

MAX()

Scalar. Returns the maximum value from a list of expressions.

Syntax
MAX(*expression, expression* [, ...] **)**

Table 158. Data Types

Return	*expression*
Integer	Integer
Long	Long
Float	Float
Interval	Interval
Timestamp	Timestamp
String	String
BLOB	BLOB

Usage

If any parameter is NULL, the function returns NULL.

Example

The following example calculates the maximum value of three columns in the stream Orders:

```
INSERT INTO OutStream
SELECT MAX(Orders.line1price, Orders.line2price, Orders.line3Price)
FROM Orders;
```

MEANDEVIATION()

Aggregate. Returns the mean absolute deviation (the mean of the absolute value of the deviations from the mean of all values) of a given expression over multiple rows.

Syntax

MEANDEVIATION([DISTINCT] *expression* **)**

Table 159. Data Types

Return	*expression*
Float	Integer
	Long
	Float
Interval	Interval
	Timestamp

Example
The following example returns the mean absolute deviation of values in the temperature column:

```
INSERT INTO OutStream
SELECT MEANDEVIATION(Devices.temperature)
FROM Devices KEEP 24 HOURS;
```

MEDIAN()

Aggregate. Returns the median value of a given expression over multiple rows.

Syntax
MEDIAN([DISTINCT] *expression* **)**

Table 160. Data Types

Return	*expression*
Float	Integer
	Long
	Float
Interval	Interval
	Timestamp

Example
The following example calculates the median value of the Price column over the last five minutes:

```
INSERT INTO OutStream
SELECT MEDIAN(Trades.Price)
FROM Trades KEEP 5 MINUTES;
```

MICROSECOND()

Scalar. Returns an integer representing the microsecond as extracted from a timestamp value.

Syntax
MICROSECOND(*timestamp* [, *timezone*] **)**

Table 161. Parameters

timestamp	An expression that evaluates to a timestamp value.
timezone	A string representing the time zone. If omitted, Sybase CEP Engine assumes the local time zone. See "Sybase CEP Time Zone Database" in the Sybase CEP Integration Guide for more information about valid time zone strings.

Table 162. Data Types

Return	*timestamp*	*timezone*
Integer	Timestamp	String

Usage

If either parameter is NULL, the function returns NULL.

Example

```
INSERT INTO OutStream
SELECT MICROSECOND(InStream.OrderTime)
FROM InStream;
```

MID()

Scalar. Returns a range of characters from a given string.

Syntax

MID(*string, start, end* **)**

Table 163. Parameters

string	The string.
start	The position of the first character to return (one-based).
end	The position of the last character to return.

Table 164. Data Types

Return	*string*	*start*	*end*
String	String	Integer	Integer

Example
The following example uses the MID function to select the fifth through seventh characters of the column named LastName:

```
INSERT INTO OutStream
SELECT MID(Employees.LastName, 5, 7)
FROM Employees;
```

If the employee's last name were "Johnson", this would return "son".

MIN()

Aggregate. Returns the minimum value of a given expression over multiple rows.

Syntax
MIN(*expression* **)**

Table 165. Data Types

Return	*expression*
Integer	Integer
Long	Long
Float	Float
Interval	Interval
Timestamp	Timestamp
String	String
BLOB	BLOB

Usage
DISTINCT is valid, but has no effect, because the minimum of the distinct values is the same as the minimum of all values.

Example
The following example calculates the minimum value of the Price column over the last five minutes:

```
INSERT INTO OutStream
SELECT MIN(Trades.Price)
FROM Trades KEEP 5 MINUTES;
```

MIN()

Scalar. Returns the minimum value from a list of expressions.

Syntax

```
MIN( expression, expression [, ...] )
```

Table 166. Data Types

Return	*expression*
Integer	Integer
Long	Long
Float	Float
Interval	Interval
Timestamp	Timestamp
String	String
BLOB	BLOB

Usage

If any parameter is NULL, the function returns NULL.

Example

The following example calculates the minimum value of three columns in the Orders stream:

```
INSERT INTO OutStream
SELECT MIN(Orders.line1price, Orders.line2price, Orders.line3Price)
FROM Orders;
```

MINUTE()

Scalar. Returns an integer representing the minute as extracted from a timestamp value.

Syntax

```
MINUTE( timestamp [, timezone ] )
```

Table 167. Parameters

timestamp	An expression that evaluates to a timestamp value.

timezone — A string representing the time zone. If omitted, Sybase CEP Engine assumes the local time zone. See "Sybase CEP Time Zone Database" in the Sybase CEP Integration Guide for more information about valid time zone strings.

Table 168. Data Types

Return	*timestamp*	*timezone*
Integer	Timestamp	String

Usage

If either parameter is NULL, the function returns NULL.

Example

```
INSERT INTO OutStream
SELECT MINUTE(InStream.OrderTime)
FROM InStream;
```

MOD()

Scalar. Returns the remainder of dividing one expression by another.

Syntax

MOD(*dividend, divisor* **)**

Table 169. Data Types

Return	*dividend*	*divisor*
Integer	Integer	Integer
Long	Long	Long
Float	Float	Float
Interval	Interval	Interval

Example

The following example calculates the modulo of two columns:

```
INSERT INTO OutStream
SELECT MOD(Orders.value, Orders.price)
FROM Orders;
```

MONTH()

Scalar. Returns an integer representing the month as extracted from a timestamp value.

Syntax

MONTH(*timestamp* [, *timezone*] **)**

Table 170. Parameters

timestamp	An expression that evaluates to a timestamp value.
timezone	A string representing the time zone. If omitted, Sybase CEP Engine assumes the local time zone. See "Sybase CEP Time Zone Database" in the Sybase CEP Integration Guide for more information about valid time zone strings.

Table 171. Data Types

Return	*timestamp*	*timezone*
Integer	Timestamp	String

Usage

If either parameter is NULL, the function returns NULL.

Example

```
INSERT INTO OutStream
SELECT MONTH(InStream.OrderTime)
FROM InStream;
```

NEXTVAL()

Scalar. The first call to this function returns 1, and then each subsequent call returns a value larger than that returned by the previous call.

Syntax

NEXTVAL()

Table 172. Data Types

Return
Long

Usage
The return value from subsequent calls to NEXTVAL() always increases, but not necessarily by one; the increase may be larger. Sybase CEP Engine maintains state for NEXTVAL() by project instance, using a separate sequence for each instance. If persistence is enabled for your project, calling NEXTVAL() after a restart returns a value larger than the last call to NEXTVAL() before the restart. Each call to NEXTVAL() returns a new value, even if it is called more than once in a single statement.

Example

```
INSERT INTO OutStream
SELECT Price, Nextval()
FROM InStream;
```

NOW()

Scalar. Returns a timestamp corresponding to the current time. Note that this function is lower resolution than Highresolutionnow(), but users considerable fewer resources. Use this version when performance is a higher priority than resolution.

Syntax
NOW()

Table 173. Data Types

Return
Timestamp

Example
The following example returns the current time minus the row timestamp:

```
INSERT INTO OutStream
SELECT NOW() - GETTIMESTAMP(Devices)
FROM Devices;
```

PI()

Scalar. Returns an approximation of the constant pi.

Syntax
PI()

Table 174. Data Types

Return
Float

Example

```
INSERT INTO OutStream (Area)
SELECT PI() * Radius ^ 2
FROM InStream;
```

POWER()

Scalar. Returns the value of a given base raised to a specified exponent.

Syntax
POWER(*base, exponent* **)**

Table 175. Data Types

Return	*base*	*exponent*
Float	Integer	Float
	Long	
	Float	

Example
The following example returns the value of the velocity column to the fourth power:

```
INSERT INTO OutStream
SELECT POWER(Devices.velocity,4.0)
FROM Devices;
```

PREV()

Other. Returns the value of a given column from a prior row.

Syntax
PREV(*column* [*, offset*]**)** **GETTIMESTAMP(PREV(** *name* **))**

Table 176. Parameters

column	The name of a column of any type.

offset	Which row to use, with 0 as the current row, 1 as the immediately previous row, and so on. If omitted, defaults to1.
name	The name of a stream or window.

Table 177. Data Types

Return	*column*	*offset*
The same type as the specified column.	String	Integer

Return	*name*
Timestamp	String

Usage

When you use PREV() on a data source grouped with GROUP BY or PER, offset refers to a previous row within the same group as the newly arrived row.

PREV embedded in GETTIMESTAMP returns the timestamp of the previous row in the specified stream or window.

You can use PREV() in a WHERE selection condition or a SELECT clause select list.

When you use PREV() on data streams, it is semantically identical to the [ ] row operator.

Example

The following example returns the value of the Price column from the row two previous to the current row:

```
INSERT INTO OutStream
SELECT PREV(Trades.Price, 2)
FROM Trades;
```

RANDOM()

Scalar. Returns a random value greater than or equal to 0 and less than 1.

Syntax

RANDOM()

Table 178. Data Types

Return
Float

Example

The following example uses the RANDOM function to generate a new stock price up to 10% higher than its original value:

```
INSERT INTO OutStream
SELECT (1 + RANDOM() /10.0) * Trades.StockPrice
FROM Trades;
```

REGEXP_FIRSTSEARCH()

Scalar. Returns the first occurrence of a POSIX regular expression pattern found in a given string.

Syntax

REGEXP_FIRSTSEARCH(*string, regex* **)**

Table 179. Parameters

string	A string.
regex	A POSIX regular expression pattern. This pattern is limited to the Perl syntax.

Table 180. Data Types

Return	*string*	*regex*
String	String	String

Usage

If string does not contain a match for the pattern, or if the specified pattern is not a valid regular expression, the function returns NULL.

One or more subexpressions can be included in the pattern, each enclosed in parentheses. If string contains a match for the pattern, the function only returns the parts of the pattern specified by the first subexpression.

Examples

The following example uses the REGEXP_FIRSTSEARCH function to return the occurrence of the letters "Corp" at the end of the string in the column StockName, if present, or Null:

```
INSERT INTO OutStream
```

```
SELECT REGEXP_FIRSTSEARCH(Trades.StockName, "Corp$")
FROM Trades;
```

The following example extracts the components of a phone number. Only the components matching the subexpressions are returned. In the case of the third REGEXP_FIRSTSEARCH, only components matching the first subexpression are returned; the second subexpression is ignored.

```
INSERT INTO OutStream
SELECT
    REGEXP_FIRSTSEARCH("(650) 210-3821", "\D*(\d{3})"),
    REGEXP_FIRSTSEARCH("(650) 210-3821", "\D*\d{3}\D*(\d{3})"),
    REGEXP_FIRSTSEARCH("(650) 210-3821","\D*\d{3}\D*\d{3}\D*(\d\d\d
\d)")
FROM Entries;
```

REGEXP_REPLACE()

Scalar. Returns a given string with the first occurrence of a match for a POSIX regular expression pattern replaced with a second, specified string.

Syntax

REGEXP_REPLACE(*string, regex, replacement* **)**

Table 181. Parameters

string	A string.
regex	A POSIX regular expression pattern. This pattern is limited to the Perl syntax.
replacement	A string to replace the part of string that matches regex.

Table 182. Data Types

Return	*string*	*regex*	*replacement*
String	String	String	String

Usage

If string does not contain a match for regex, the function returns string with no replacements. If regex is not a valid regular expression, the function returns NULL.

Example

```
INSERT INTO OutStream
SELECT
   REGEXP_REPLACE(InStream.src, InStream.regexp,
InStream.replace_str),
   InStream.src,
```

```
   InStream.regexp,
   InStream.replace_str
FROM InStream;
```

REGEXP_SEARCH()

Scalar. Determines whether or not a string contains a match for a POSIX regular expression pattern.

Syntax

REGEXP_SEARCH(*string, regex* **)**

Table 183. Parameters

string	A string.
regex	A POSIX regular expression pattern. This pattern is limited to the Perl syntax.

Table 184. Data Types

Return	*string*	*regex*
Boolean	String	String

Example

The following example uses the REGEXP_SEARCH function to determine if the value in the StockName column has the string "Corp" at the end:

```
INSERT INTO OutStream
SELECT REGEXP_SEARCH(Trades.StockName, "Corp$")
FROM Trades;
```

REGR_AVGX()

Aggregate. Computes the average of the independent variable of the regression line.

Syntax

```
REGR_AVGX( dependent-expression , independent-expression )
```

Table 185. Parameters

dependent-expression	The variable that is affected by the independent variable.
independent-expression	The variable that influences the outcome.

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an

empty set, then it returns NULL. The function is applied to the set of (dependent-expression and independent-expression) pairs after eliminating all pairs for which either dependent-expression or independent-expression is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where y represents the independent-expression:

```
AVG( y )
```

SQL/2003 SQL foundation feature (T621) outside of core SQL.

The following example calculates the average of the dependent variable, employee age.

With a GROUP BY clause:

```
SELECT REGR_AVGX()( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
group by depart_ID
```

With a window:

```
SELECT REGR_AVGX()( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
KEEP 10 ROWS
```

REGR_AVGY()

Aggregate. Computes the average of the dependent variable of the regression line.

Syntax

```
REGR_AVGY( dependent-expression , independent-expression )
```

Table 186. Parameters

dependent-expression	The variable that is affected by the independent variable.
independent-expression	The variable that influences the outcome.

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL. The function is applied to the set of (dependent-expression and independent-expression) pairs after eliminating all pairs for which either dependent-expression or independent-expression is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where x represents the dependent-expression:

```
AVG( x )
```

SQL/2003 SQL foundation feature (T621) outside of core SQL.

The following example calculates the average of the independent variable, employee salary.

With a GROUP BY clause:

```
SELECT REGR_AVGY()( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
group by depart_ID
```

With a window:

```
SELECT REGR_AVGY()( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
KEEP 10 ROWS
```

REGR_COUNT()

Aggregate. Returns an integer that represents the number of non-NULL number pairs used to fit the regression line.

Syntax

```
REGR_COUNT( dependent-expression , independent-expression )
```

Table 187. Parameters

dependent-expression	The variable that is affected by the independent variable.
independent-expression	The variable that influences the outcome.

This function returns a LONG as the result.

SQL/2003 SQL foundation feature (T621) outside of core SQL.

The following example returns the number of non-NULL pairs that were used to fit the regression line.

With a GROUP BY clause:

```
SELECT REGR_COUNT()( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
group by depart_ID
```

With a window:

```
SELECT REGR_COUNT()( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
KEEP 10 ROWS
```

REGR_INTERCEPT()

Aggregate. Computes the y-intercept of the linear regression line that best fits the dependent and independent variables.

Syntax

```
REGR_INTERCEPT( dependent-expression , independent-
expression )
```

Table 188. Parameters

dependent-expression	The variable that is affected by the independent variable.
independent-expression	The variable that influences the outcome.

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL. The function is applied to the set of (dependent-expression and independent-expression) pairs after eliminating all pairs for which either dependent-expression or independent-expression is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where x represents the dependent-expression and y represents the independent-expression:

```
    AVG( x ) - REGR_SLOPE( x, y ) * AVG( y )
```

SQL/2003 SQL foundation feature (T621) outside of core SQL.

The following example returns the y-intercept of the linear regression line.

With a GROUP BY clause:

```
SELECT REGR_INTERCEPT()( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
group by depart_ID
```

With a window:

```
SELECT REGR_INTERCEPT()( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
KEEP 10 ROWS
```

REGR_R2()

Aggregate. Computes the coefficient of determination (also referred to as R-squared or the goodness of fit statistic) for the regression line.

Syntax

```
REGR_R2( dependent-expression , independent-expression )
```

Table 189. Parameters

dependent-expression	The variable that is affected by the independent variable.
independent-expression	The variable that influences the outcome.

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL. The function is applied to the set of (dependent-expression and independent-expression) pairs after eliminating all pairs for which either dependent-expression or independent-expression is NULL.

SQL/2003 SQL foundation feature (T621) outside of core SQL.

The following example returns the coefficient of determination for the regression line.

With a GROUP BY clause:

```
SELECT REGR_R2()( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
group by depart_ID
```

With a window:

```
SELECT REGR_R2()( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
KEEP 10 ROWS
```

REGR_SLOPE()

Aggregate. Computes the slope of the linear regression line fitted to non-NULL pairs.

Syntax

```
REGR_SLOPE( dependent-expression , independent-expression )
```

Table 190. Parameters

dependent-expression	The variable that is affected by the independent variable.
independent-expression	The variable that influences the outcome.

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL. The function is applied to the set of (dependent-expression and independent-expression) pairs after eliminating all pairs for which either dependent-expression or independent-expression is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where x represents the dependent-expression and y represents the independent-expression:

```
COVAR_POP( x, y ) / VAR_POP( y )
```

SQL/2003 SQL foundation feature (T621) outside of core SQL.

With a GROUP BY clause:

```
SELECT REGR_SLOPE()( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
group by depart_ID
```

With a window:

```
SELECT REGR_SLOPE()( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
KEEP 10 ROWS
```

REGR_SXX()

Aggregate. Returns the sum of squares of the independent expressions used in a linear regression model. Use the REGR_SXX function to evaluate the statistical validity of a regression model.

Syntax

```
REGR_SXX( dependent-expression , independent-expression )
```

Table 191. Parameters

dependent-expression	The variable that is affected by the independent variable.
independent-expression	The variable that influences the outcome.

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL. The function is applied to the set of (dependent-expression

and independent-expression) pairs after eliminating all pairs for which either dependent-expression or independent-expression is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where x represents the dependent-expression and y represents the independent-expression:

```
REGR_COUNT( x, y ) * VAR_POP( x )
```

SQL/2003 SQL foundation feature (T621) outside of core SQL.

With a GROUP BY clause:

```
SELECT REGR_SXX()( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
group by depart_ID
```

With a window:

```
SELECT REGR_SXX()( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
KEEP 10 ROWS
```

REGR_SXY()

Aggregate. Returns the sum of products of the dependent and independent variables. The REGR_SXY function can be used to evaluate the statistical validity of a regression model.

Syntax

```
REGR_SXY( dependent-expression , independent-expression )
```

Table 192. Parameters

dependent-expression	The variable that is affected by the independent variable.
independent-expression	The variable that influences the outcome.

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL. The function is applied to the set of (dependent-expression and independent-expression) pairs after eliminating all pairs for which either dependent-expression or independent-expression is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where x represents the dependent-expression and y represents the independent-expression:

```
REGR_COUNT( x, y ) * COVAR_POP( x, y )
```

SQL/2003 SQL foundation feature (T621) outside of core SQL.

The following example returns the sum of products of the dependent and independent variables.

With a GROUP BY clause:

```
SELECT REGR_SXY()( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
group by depart_ID
```

With a window:

```
SELECT REGR_SXY()( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
KEEP 10 ROWS
```

REGR_SYY()

Aggregate. Returns values that can evaluate the statistical validity of a regression model.

Syntax

```
REGR_SYY( dependent-expression , independent-expression )
```

Table 193. Parameters

dependent-expression	The variable that is affected by the independent variable.
independent-expression	The variable that influences the outcome.

This function converts its arguments to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL. The function is applied to the set of (dependent-expression and independent-expression) pairs after eliminating all pairs for which either dependent-expression or independent-expression is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where x represents the dependent-expression and y represents the independent-expression:

```
REGR_COUNT( x, y ) * VAR_POP( y )
```

SQL/2003 SQL foundation feature (T621) outside of core SQL.

With a GROUP BY clause:

```
SELECT REGR_SYY()( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
group by depart_ID
```

With a window:

```
SELECT REGR_SYY()( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees
KEEP 10 ROWS
```

REPLACE()

Scalar. Returns a given string with every instance of a specified substring replaced with another specified string.

Syntax

REPLACE(*string, find, replacement* **)**

Table 194. Parameters

string	A string.
find	The string to locate.
replacement	The string to substitute for every occurrence of find.

Table 195. Data Types

Return	*string*	*find*	*replacement*
String	String	String	String

Example

The following example uses the REPLACE function to substitute the result of multiplying the values of two columns, converted to a string, for a square bracketed x in the filledOrder column:

```
INSERT INTO OutStream
SELECT REPLACE(Transaction.filledOrder, '[x]',
       TO_STRING(InStock.price*InStock.quantity) )
FROM Orders, InStock;
```

RIGHT()

Scalar. Returns a specified number of characters from the end of a given string.

Syntax

RIGHT(*string, count* **)**

Table 196. Data Types

Return	*string*	*count*
String	String	Integer

Example
The following example uses the RIGHT function to extract the last 25 characters of the value in the Comments column:

```
INSERT INTO OutStream
SELECT RIGHT(Transactions.Comments,25)
FROM Transactions;
```

ROUND()

Scalar. Returns a given value rounded to a specified number of decimal places.

Syntax
ROUND(*value, precision* **)**

Table 197. Parameters

value	The value to round.
precision	The number of places to the right or, if negative, the left of the decimal.

Table 198. Data Types

Return	*value*	*precision*
Float	Float	Integer

Examples
The following example returns the value of the velocity column rounded to the nearest thousandth:

```
INSERT INTO OutStream
SELECT ROUND(Devices.velocity, 3)
FROM Devices;
```

The following example returns the value rounded to the nearest hundred, which in this case would be 800.0 (note the negative value for the precision):

```
INSERT INTO OutStream
SELECT ROUND(799.9, -2)
FROM Devices;
```

RTRIM()

Scalar. Returns a specified string after stripping trailing spaces.

Syntax
RTRIM(*string* **)**

Table 199. Data Types

Return	*string*
String	String

Example
The following example uses the RTRIM function to remove trailing spaces from the value in the column: Symbol:

```
INSERT INTO OutStream
SELECT RTRIM(Trades.Symbol)
FROM Trades;
```

SECOND()

Scalar. Returns an integer representing the second as extracted from a timestamp value.

Syntax
SECOND(*timestamp* [*, timezone*] **)**

Table 200. Parameters

timestamp	An expression that evaluates to a timestamp value.
timezone	A string representing the time zone. If omitted, Sybase CEP Engine assumes the local time zone. See "Sybase CEP Time Zone Database" in the Sybase CEP Integration Guide for more information about valid time zone strings.

Table 201. Data Types

Return	*timestamp*	*timezone*
Integer	Timestamp	String

Usage
If either parameter is NULL, the function returns NULL.

Example

```
INSERT INTO OutStream
SELECT SECOND(InStream.OrderTime)
FROM InStream;
```

SIGN()

Scalar. Determines whether a given value is positive or negative.

Syntax
SIGN(*value* **)**

Table 202. Data Types

Return	*value*
Integer	Integer
	Long
	Float
	Interval

Usage
SIGN() returns 1 if value is positive, -1 if it is negative, and 0 otherwise.

Example
The following example uses the SIGN function to compute an additional commission discount of $50 if there are more than 10,000 shares of a stock traded, $0 discount if there are exactly 10,000 shares traded, and adds $50 to commission if less than 10,000 shares are traded:

```
INSERT INTO OutStream
SELECT SIGN(10000-Trades.Quantity)*50 + Trades.Commission,
           Trades.StockName
FROM Trades;
```

SIN()

Scalar. Returns the sine, in radians, of a given value.

Syntax
SIN(*value* **)**

Table 203. Data Types

Return	*value*
Float	Float

Example

```
INSERT INTO OutStream
SELECT SIN(InStream.FloatColumn)
FROM InStream;
```

SIND()

Scalar. Returns the sine, in degrees, of a given value.

Syntax
SIND(*value* **)**

Table 204. Data Types

Return	*value*
Float	Float

Example

```
INSERT INTO OutStream
SELECT SIND(InStream.FloatColumn)
FROM InStream;
```

SINH()

Scalar. Returns the hyperbolic sine of a given value.

Syntax
SINH(*value* **)**

Table 205. Data Types

Return	*value*
Float	Float

Example

```
INSERT INTO OutStream
```

```
SELECT SINH(InStream.FloatColumn)
FROM InStream;
```

SQRT()

Scalar. Returns the square root of a given value.

Syntax
SQRT(*value* **)**

Table 206. Data Types

Return	*value*
Float	Float

Example
The following example returns the square root of the value in the velocity column:

```
INSERT INTO OutStream
SELECT SQRT(Devices.velocity)
FROM Devices;
```

STDDEV()

Aggregate. Computes the standard deviation of a sample consisting of a numeric-expression, as a DOUBLE. Alias for STDDEV_SAMP().

STDDEVIATION()

Aggregate. Returns the standard deviation of a given expression over multiple rows.

Syntax
STDDEVIATION([DISTINCT] *expression* **)**

Table 207. Data Types

Return	*expression*
Float	Integer
	Long
	Float

Return	*expression*
Interval	Interval
	Timestamp

Example

The following example returns the standard deviation of the values of the temperature column for the last 24 hours:

```
INSERT INTO OutStream
SELECT STDDEVIATION(Devices.temperature)
FROM Devices KEEP 24 HOURS;
```

STDDEV_POP()

Aggregate. Computes the standard deviation of a population consisting of a numeric-expression, as a DOUBLE.

Syntax

```
STDDEV_POP( numeric-expression )
```

Table 208. Parameters

numeric-expression	The expression whose population-based standard deviation is calculated over a set of rows. The expression is commonly a column name.

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. The population-based standard deviation (s) is computed according to the following formula:

```
s = [ (1/N) * SUM( xi - MEAN( x ) )2 ]1/2
```

This standard deviation does not include rows where numeric-expression is NULL. It returns NULL for a group containing no rows.

SQL/2003 SQL foundation feature (T621) outside of core SQL.

The following example demonstrates how to use the STDDEV_POP function to calculate the standard deviation of a population using data from a CSV file:

```
-- create input stream schema
CREATE SCHEMA InSchema (
    id STRING,
    x FLOAT,
    y FLOAT
```

```
);

-- create output stream schema
CREATE SCHEMA OutSchema (
    id STRING,
    x FLOAT,
    y FLOAT,
    stddev_pop_result FLOAT,
);

-- create input stream
CREATE INPUT STREAM StreamIn
SCHEMA InSchema;

-- create master window
CREATE MASTER WINDOW ResultWindow
SCHEMA OutSchema
KEEP 5 ROWS
;

-- create output stream
CREATE OUTPUT STREAM StreamOut
SCHEMA OutSchema;

-- input stream read data from csv file by ReadFromCsvFileAdapterType
adapter
ATTACH INPUT ADAPTER ReadFromCSVFile TYPE ReadFromCsvFileAdapterType
TO STREAM StreamIn
PROPERTIES
    FILENAME            = "$ProjectFolder\..\data\data.csv",
    TITLEROW            = "false",
    TIMESTAMPCOLUMN     = "false",
    RATE                = "1",
    USECURRENTTIMESTAMP = "true"
;

-- output stream write data to csv file by WriteToCsvFileAdapterType
adapter
ATTACH OUTPUT ADAPTER WriteToCSVFile TYPE WriteToCsvFileAdapterType
TO STREAM StreamOut
PROPERTIES
    FILENAME = "$ProjectFolder\..\data\result.csv"
;

-- insert the calculated result to window
INSERT INTO ResultWindow
SELECT id,x,y,
stddev_pop(x)
FROM StreamIn
KEEP 5 ROWS;

INSERT INTO StreamOut
SELECT *
FROM ResultWindow;
```

STDDEV_SAMP()

Aggregate. Computes the standard deviation of a sample consisting of a numeric-expression, as a DOUBLE.

Syntax

```
STDDEV_SAMP( numeric-expression )
```

Table 209. Parameters

numeric-expression	The expression whose sample-based standard deviation is calculated over a set of rows. The expression is commonly a column name.

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. The standard deviation (s) is computed according to the following formula, which assumes a normal distribution:

```
s = [ (1/( N - 1 )) * SUM( xi - mean( x ) )2 ]1/2
```

This standard deviation does not include rows where numeric-expression is NULL. It returns NULL for a group containing either 0 or 1 rows.

SQL/2003 SQL foundation feature (T621) outside of core SQL.

The following example demonstrates how to use the STDDEV_SAMP function to calculate the standard deviation of a sample using data from a CSV file:

```
-- create input stream schema
CREATE SCHEMA InSchema (
    id STRING,
    x FLOAT,
    y FLOAT
);

-- create output stream schema
CREATE SCHEMA OutSchema (
    id STRING,
    x FLOAT,
    y FLOAT,
    stddev_samp_result FLOAT,
);

-- create input stream
CREATE INPUT STREAM StreamIn
SCHEMA InSchema;

-- create master window
CREATE MASTER WINDOW ResultWindow
SCHEMA OutSchema
```

```
KEEP 5 ROWS
;

-- create output stream
CREATE OUTPUT STREAM StreamOut
SCHEMA OutSchema;

-- input stream read data from csv file by ReadFromCsvFileAdapterType
adapter
ATTACH INPUT ADAPTER ReadFromCSVFile TYPE ReadFromCsvFileAdapterType
TO STREAM StreamIn
PROPERTIES
    FILENAME            = "$ProjectFolder\..\data\data.csv",
    TITLEROW            = "false",
    TIMESTAMPCOLUMN     = "false",
    RATE                = "1",
    USECURRENTTIMESTAMP = "true"
;

-- output stream write data to csv file by WriteToCsvFileAdapterType
adapter
ATTACH OUTPUT ADAPTER WriteToCSVFile TYPE WriteToCsvFileAdapterType
TO STREAM StreamOut
PROPERTIES
    FILENAME = "$ProjectFolder\..\data\result.csv"
;

-- insert the calculated result to window
INSERT INTO ResultWindow
SELECT id,x,y,
stddev_samp(x)
FROM StreamIn
KEEP 5 ROWS;

INSERT INTO StreamOut
SELECT *
FROM ResultWindow;
```

SUBSTR()

Scalar. Returns a specified number of characters from a specific position within a given string.

Syntax

SUBSTR(*string, start* [*, count*])

Table 210. Parameters

string	The string.
start	The first character to return (one-based).

count The number of characters to return. If omitted, returns all characters from start to the end of the string.

Table 211. Data Types

Return	*string*	*start*	*count*
String	String	Integer	Integer

Example

The following example uses the SUBSTR function to select the fifth and sixth characters of the value in the LastName column:

```
INSERT INTO OutStream
SELECT SUBSTR(Employees.LastName, 5, 2)
FROM Employees;
```

If the value were "Johnson", the returned value would be "so".

SUM()

Aggregate. Returns the sum of the values of a given expression over multiple rows.

Syntax

SUM([DISTINCT] *expression* **)**

Table 212. Data Types

Return	*expression*
Integer	Integer
Long	Long
Float	Float
Interval	Interval

Example

The following example calculates the sum of the values in the Volume column over the last 5 minutes:

```
INSERT INTO OutStream
SELECT SUM(Volume) AS VolumeSum
FROM Trades KEEP 5 MINUTES;
```

TAN()

Scalar. Returns the tangent, in radians, of a given value.

Syntax
TAN(*value* **)**

Table 213. Data Types

Return	*value*
Float	Float

Example

```
INSERT INTO OutStream
SELECT TAN(InStream.FloatColumn)
FROM InStream;
```

TAND()

Scalar. Returns the tangent, in degrees, of a given value.

Syntax
TAND(*value* **)**

Table 214. Data Types

Return	*value*
Float	Float

Example

```
INSERT INTO OutStream
SELECT TAND(InStream.FloatColumn)
FROM InStream;
```

TANH()

Scalar. Returns the hyperbolic tangent of a given value.

Syntax
TANH(*value* **)**

Table 215. Data Types

Return	*value*
Float	Float

Example

```
INSERT INTO OutStream
SELECT TANH(InStream.FloatColumn)
FROM InStream;
```

THRESHOLD()

Scalar. Determines whether or not the value in a given column crosses a specified threshold.

Syntax

THRESHOLD(UP | DOWN | ANY, *column*, *value*)

Table 216. Parameters

column	The name of a stream or window column containing a numeric or time value.
value	The threshold value. Must be the same data type as the column specified with column.

Table 217. Data Types

Return	*column*	*value*
Boolean	String	Integer
		Long
		Float
		Interval
		Timestamp

Usage

This function returns True if the value in the specified column of the current row is greater than value after having been less than or equal to value in the previous row (for UP), or less than value after previously being greater than or equal to value (for DOWN), or either (for ANY). Note that this function returns True for the first row in a stream or window if the value of the specified column is greater than value (for UP), less than value (for DOWN), or not equal to value (for ANY).

If using this function for a window partitioned with GROUP BY or PER, the threshold comparison applies to each partition individually, not the window as a whole.

Example
The following example query publishes a row to its destination stream when the value of the Trades.Price column exceeds 100:

```
INSERT INTO OutStream
SELECT Symbol, Price, Volume
FROM Trades
WHERE THRESHOLD(UP, Trades.Price, 100);
```

TO_BLOB()

Scalar. Converts a given string to a BLOB value.

Syntax
TO_BLOB(*value* **)**

Table 218. Data Types

Return	*value*
BLOB	String
	Blob

Example
The following example returns a BLOB based on the input string:

```
INSERT INTO OutStream
SELECT TO_BLOB('0123456789abcdef') -- BLOB of 8 bytes
FROM ImageData;
```

TO_BOOLEAN()

Scalar. Converts a given string to a Boolean value

Syntax
TO_BOOLEAN(*value* **)**

Table 219. Parameter

value	The value to convert. The strings "True" and "Yes", regardless of case, or the numeral "1" returns True. NULL returns NULL. Any other string returns False.

Table 220. Data Types

Return	*value*
Boolean	String
	Boolean

Examples

The following examples return a Boolean based on the input string:

```
INSERT INTO OutStream
SELECT TO_BOOLEAN('FALSE') --returns FALSE
FROM InStream;
```

```
INSERT INTO OutStream
SELECT TO_BOOLEAN('TRUE') --returns TRUE
FROM InStream;
```

```
INSERT INTO OutStream
SELECT TO_BOOLEAN( LEFT( LTRIM(' false status'), 5 ) );
   --returns FALSE
FROM InStream;
```

TO_FLOAT()

Scalar. Converts a given value to a floating point value.

Syntax

TO_FLOAT(*value* **)**

Table 221. Parameter

value	The value to convert. A String converts based on the format for a Float literal. An Interval returns a value representing a number of microseconds. A Timestamp returns a value representing the number of microseconds from the epoch (midnight, January 1, 1970 UTC). Timestamps prior to the epoch convert to a negative value.

Table 222. Data Types

Return	*value*
Float	Integer
	Long
	Float

Return	*value*
	Interval
	Timestamp
	String

Example
The following example takes the integer value of NumContracts, converts it to a Float and then multiplies it by the floating point number 100:

```
INSERT INTO OutStream
SELECT 100.0 * TO_FLOAT(Trades.NumContracts)
FROM Trades;
```

TO_INTEGER()

Scalar. Converts a given value to an Integer value.

Syntax
TO_INTEGER(*value* **)**

Table 223. Parameter

value	The value to convert. Numeric values return the integer portion of the value. Values outside the valid range for an integer, or non-numeric characters in a string value, return NULL.

Table 224. Data Types

Return	*value*
Integer	Integer
	Long
	Float
	String

Example
The following example uses the integer portion of the result of dividing the value in the Price column by 9:

```
INSERT INTO OutStream
SELECT TO_INTEGER(Trades.Price/9.0)>=5
FROM Trades;
```

TO_INTERVAL()

Scalar. Converts a given value to an Interval.

Syntax
TO_INTERVAL(*value* **)**

Table 225. Parameter

value	A value representing a number of microseconds. Strings must follow the format for an Interval literal. See Time Literals for more information.

Table 226. Data Types

Return	*value*
Interval	Long
	Float
	Interval
	String

Examples
The following example computes an Interval value from the value in a column representing microseconds:

```
INSERT INTO OutStream
SELECT TO_INTERVAL(Devices.MicrosecsSinceReset)
FROM Devices;
```

The following example computes an Interval value from a String literal:

```
INSERT INTO OutStream
SELECT TO_INTERVAL('1 02:03:04.567890')
FROM InStream;
```

TO_LONG()

Scalar. Converts a given value to a Long.

Syntax
TO_LONG(value)

Table 227. Parameter

value	The value to convert. Numeric values return the integer portion of the value. Values outside the valid range for a Long, or non-numeric characters in a string value, return Null. An Interval returns a number of microseconds. A Timestamp returns a value representing the number of microseconds from the epoch (midnight, January 1, 1970 UTC). Timestamps prior to the epoch convert to a negative value.

Table 228. Data Types

Return	*value*
Long	Integer
	Long
	Float
	Interval
	Timestamp
	String

Examples

The following example employs the Long integer portion of a Float value in a calculation:

```
INSERT INTO OutStream
SELECT TO_LONG(Trades.Price/9.)>=5
FROM Trades;
```

The following example returns a number of microseconds:

```
INSERT INTO OutStream
SELECT TO_LONG(Trades.TradeTime)-TO_LONG(Trades.ExchangeTime)
FROM Trades;
```

TO_STRING()

Scalar. Converts a given value to a String.

Syntax

TO_STRING(*value* [*, format*] [*, timezone*])

Table 229. Parameters

value	The value to convert.

format	A format string. Only valid if value is an Integer, Long, Float, or Timestamp.
timezone	A time zone. Only valid if value is a Timestamp. If omitted, assumes the local time zone.

Table 230. Data Types

Return	*value*	*format*	*timezone*
String	Integer	String	N/A
	Long		
	Float		
	Timestamp		String
	Interval	N/A	N/A
	String		
	BLOB		
	Boolean		
	XML		

Usage

This function coverts values as follows:

- For an Integer or Long value, you can include an optional format string that specifies the format for the string output. The format string follows the ISO standard for fprintf. The default for Integer expressions is "%ld", while the default for Long expressions is "%lld". Note: Sybase CEP Engine follows whatever variations or restrictions are placed on printf for the operating system on which your Sybase CEP Server is being run, so those variations apply to this format string as well. Also note that an incorrect format string may cause Sybase CEP Server to exit.
- For a Timestamp value, you can include a format string to control the output format of the string. The default format is YYYY-MM-DD HH24:MI:SS.FF. See Timestamp Format Codes for more information. There is also an optional time zone string option. See "Sybase CEP Time Zone Database" in the Sybase CEP *Integration Guide* for more information about valid time zone strings.
- This function coverts a BLOB value to a String containing hexadecimal characters.
- See XMLSERIALIZE for information about converting an XML value to a String.
- For a Float value, you can include an optional format string that specifies the format for the output of the floating point number. The format string can include the following characters:

Character	Description
. or D	Returns a decimal point in the specified position. Only one decimal point can be specified, or the output will contain number signs instead of the value.
9	Replaced in the output by a single digit of the value. The value is returned with as many characters as there are 9s in the format string. If the value is positive, a leading space is included to the left of the value. If the value is negative, a leading minus sign is included to the left of the value. Excess 9s to the left of the decimal point are replaced with spaces, while excess 9s to the right of the decimal point are replaced with zeroes. Insufficient 9s to the left of the decimal point returns number signs instead of the value, while insufficient 9s to the right of the decimal point result in rounding. Use FM to strip leading spaces.
0	To the left of the decimal point, replaced in the output by a single digit of the value or a zero, if the value does not have a digit in the position of the zero. To the right of the decimal point, treated as a 9. If the value is positive, a leading space is included to the left of the value. If the value is negative, a leading minus sign is included to the left of the value. Use FM to strip leading spaces.
EEEE	Returns the value in scientific notation. The output for this format always includes a single digit before the decimal. Combine with a decimal point and 9s to specify precision. 9s to the left of the decimal point are ignored. Must be placed at the end of the format string.
S	Returns a leading or trailing minus sign (-) or plus sign (+), depending on whether the value is positive or negative. Can only be placed at the beginning or end of the format string. Eliminates the usual single leading space, but not leading spaces as the result of excess 9s, zeroes, or commas.
$	Returns a leading dollar sign in front of the value. Can be placed anywhere in the format string.

Character	Description
,	Returns a comma in the specified position. If there are no digits to the left of the comma, the comma is replaced with a space. You can specify multiple commas, but cannot specify a comma as the first character in the format, or to the right of the decimal point.
FM	Strips leading spaces from the output.

The following table shows several examples of format strings and the resulting output. Note that a space in the output is represented by a small circle (·):

Example	Output
TO_STRING(1234,'9999')	"·1234"
TO_STRING(1234.567,'9999')	"·1235"
TO_STRING(1234.567,'999')	"####"
TO_STRING(1234.567,'9999D999')	"·1234.567"
TO_STRING(1234.567,'9999D9')	"·1234.6"
TO_STRING(1234.567,'9999.9999')	"·1234.5670"
TO_STRING(1234.567,'999999.9999')	"···1234.5670"
TO_STRING(1234.567,'009999.999')	"·001234.567"
TO_STRING(1234.567,'00000.999')	"·01234.567"
TO_STRING(1234.567,'9,999.999')	"·1,234.567"
TO_STRING(1234.567,'9,9,999.999')	"···1,234.567"
TO_STRING(1234.567,'FM9,9,999.999')	"1,234.567"
TO_STRING(1234.567,'$9,999.999')	"·$1,234.567"
TO_STRING(1234.567,'9,999.$999')	"·$1,234.567"
TO_STRING(1234.567,'EEEE')	"1E+03"
TO_STRING(1234.567,'.999EEEE')	"1.235E+03"
TO_STRING(1234.567,'999.999EEEE')	"1.235E+03"
TO_STRING(1234.567,'.99999999EEEE')	"1.23456700E+03"
TO_STRING(1234.567,'S9,999.99')	"+1,234.57"
TO_STRING(1234.567,'9,999.99S')	"1,234.57+"

Example	Output
TO_STRING(1234.567,'9,9,999.99S')	"··1,234.57+"
TO_STRING(1234.567,'FM9,9,999.99S')	"1,234.57+"

Examples

The following example converts a numeric value to a string:

```
INSERT INTO OutStream
SELECT TO_STRING(Trades.Price)
FROM Trades;
```

The following example converts a TIMESTAMP to a STRING with a format specifying precision:

```
INSERT INTO OutStream
SELECT TO_STRING(Trades.TradeTime, 'YYYY-MM-DD HH:MI')
FROM Trades;
```

TO_TIMESTAMP()

Scalar. Converts a given value to a Timestamp.

Syntax

TO_TIMESTAMP(*value* **)** **TO_TIMESTAMP(** *value, format* **)**

Table 231. Parameters

value	The value to convert. Numeric values represent the number of microseconds from the epoch (midnight, January 1, 1970 UTC).
format	A format string. Only valid if value is a String. See Timestamp Format Codes for more information.

Table 232. Data Types

Return	*value*	*format*
Timestamp	Long	N/A
	Float	
	Timestamp	
	String	String

Example
The following example reads a string in the specified format and converts it to a Timestamp:

```
INSERT INTO OutStream
SELECT TO_TIMESTAMP(Log.Time, 'DY MON DD HH24:MI:SS YYYY')
   AS Time
FROM Log;
```

XMLPARSE() and TO_XML() functions

Scalar. Converts (casts) a string expression to XML.

Syntax
XMLPARSE(*value* **) TO_XML(** *value* **)**

Table 233. Parameters

value	The value to convert.

Table 234. Data Types

Return	*value*
XML	String
	XML

See Also

- XMLSERIALIZE

Example
The following example shows an extraction, serialization (to string), and reparsing:

```
INSERT INTO OutStream
SELECT XMLPARSE(XMLSERIALIZE(XMLEXTRACT(OrderVal2,'//item')))
FROM Orders;
```

TRIM()

Scalar. Returns a given string after removing leading and trailing spaces.

Syntax
TRIM(*string* **)**

Table 235. Data Types

Return	*string*
String	String

Example
The following example uses the TRIM function to remove leading and trailing spaces from the value in the Symbol column:

```
INSERT INTO OutStream
SELECT TRIM(Trades.Symbol)
FROM Trades;
```

UPPER()

Scalar. Returns a given string with all characters converted to upper case.

Syntax
UPPER(*string* **)**

Table 236. Data Types

Return	*string*
String	String

Example
The following example uses the UPPER function to convert the value of StockName to all upper-case characters:

```
INSERT INTO OutStream
SELECT UPPER(Trades.StockName)
FROM Trades;
```

USERNAME()

Scalar. Returns the user name of the owner of the current query module file. Only valid if the Sybase CEP Server is configured for user authentication.

Syntax
USERNAME()

Table 237. Data Types

Return
String

Example

```
INSERT INTO OutStream
SELECT USERNAME(), InStream.Password
FROM InStream;
```

VARIANCE()

Aggregate. Computes the statistical variance of a sample consisting of a numerical-expression, as a DOUBLE. The VARIANCE() function is an alias for VAR_SAMP(); the alias is a vendor extension and not specified by ANSI SQL.

VAR_POP()

Aggregate. Computes the statistical variance of a population consisting of a numeric-expression, as a DOUBLE.

Syntax

```
VAR_POP( numeric-expression )
```

Table 238. Parameters

numeric-expression	The expression whose population-based variance is calculated over a set of rows. The expression is commonly a column name.

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. The population-based variance (s2) of numeric-expression (x) is computed according to the following formula:

```
s2 = (1/N) * SUM( xi - mean( x ) )2
```

This variance does not include rows where numeric-expression is NULL. It returns NULL for a group containing no rows.

SQL/2003 SQL foundation feature (T611) outside of core SQL.

The following example demonstrates how to use the VAR_POP function to calculate the statistical variance of a populationusing data from a CSV file:

```
-- create input stream schema
CREATE SCHEMA InSchema (
```

```
    id STRING,
    x FLOAT,
    y FLOAT
);

-- create output stream schema
CREATE SCHEMA OutSchema (
    id STRING,
    x FLOAT,
    y FLOAT,
    var_pop_result FLOAT,
);

-- create input stream
CREATE INPUT STREAM StreamIn
SCHEMA InSchema;

-- create master window
CREATE MASTER WINDOW ResultWindow
SCHEMA OutSchema
KEEP 5 ROWS
;

-- create output stream
CREATE OUTPUT STREAM StreamOut
SCHEMA OutSchema;

-- input stream read data from csv file by ReadFromCsvFileAdapterType
adapter
ATTACH INPUT ADAPTER ReadFromCSVFile TYPE ReadFromCsvFileAdapterType
TO STREAM StreamIn
PROPERTIES
    FILENAME            = "$ProjectFolder\..\data\data.csv",
    TITLEROW            = "false",
    TIMESTAMPCOLUMN     = "false",
    RATE                = "1",
    USECURRENTTIMESTAMP = "true"
;

-- output stream write data to csv file by WriteToCsvFileAdapterType
adapter
ATTACH OUTPUT ADAPTER WriteToCSVFile TYPE WriteToCsvFileAdapterType
TO STREAM StreamOut
PROPERTIES
    FILENAME = "$ProjectFolder\..\data\result.csv"
;

-- insert the calculated result to window
INSERT INTO ResultWindow
SELECT id,x,y,
var_pop(x)
FROM StreamIn
KEEP 5 ROWS;

INSERT INTO StreamOut
```

```
SELECT *
FROM ResultWindow;
```

VAR_SAMP()

Aggregate. Computes the statistical variance of a sample consisting of a numeric-expression, as a DOUBLE.

Syntax

```
VAR_SAMP( numeric-expression )
```

Table 239. Parameters

numeric-expression	The expression whose sample-based variance is calculated over a set of rows. The expression is commonly a column name.

Usage

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. The variance (s2) of numeric-expression (x) is computed according to the following formula, which assumes a normal distribution:

```
s2 = (1/( N - 1 )) * SUM( xi - mean( x ) )2
```

This variance does not include rows where numeric-expression is NULL. It returns NULL for a group containing either 0 or 1 rows.

SQL/2003 SQL foundation feature outside of core SQL. The VARIANCE syntax is a vendor extension.

The following example demonstrates how to use the VAR_SAMP functionto calculate the statistical variance of a sample of data from a CSV file:

```
-- create input stream schema
CREATE SCHEMA InSchema (
    id STRING,
    x FLOAT,
    y FLOAT
);

-- create output stream schema
CREATE SCHEMA OutSchema (
    id STRING,
    x FLOAT,
    y FLOAT,
    var_samp_result FLOAT,
);

-- create input stream
CREATE INPUT STREAM StreamIn
SCHEMA InSchema;
```

```
-- create master window
CREATE MASTER WINDOW ResultWindow
SCHEMA OutSchema
KEEP 5 ROWS
;

-- create output stream
CREATE OUTPUT STREAM StreamOut
SCHEMA OutSchema;

-- input stream read data from csv file by ReadFromCsvFileAdapterType
adapter
ATTACH INPUT ADAPTER ReadFromCSVFile TYPE ReadFromCsvFileAdapterType
TO STREAM StreamIn
PROPERTIES
    FILENAME            = "$ProjectFolder\..\data\data.csv",
    TITLEROW            = "false",
    TIMESTAMPCOLUMN     = "false",
    RATE                = "1",
    USECURRENTTIMESTAMP = "true"
;

-- output stream write data to csv file by WriteToCsvFileAdapterType
adapter
ATTACH OUTPUT ADAPTER WriteToCSVFile TYPE WriteToCsvFileAdapterType
TO STREAM StreamOut
PROPERTIES
    FILENAME = "$ProjectFolder\..\data\result.csv"
;

-- insert the calculated result to window
INSERT INTO ResultWindow
SELECT id,x,y,
var_samp(x)
FROM StreamIn
KEEP 5 ROWS;

INSERT INTO StreamOut
SELECT *
FROM ResultWindow;
```

VWAP()

Aggregate. The VWAP function computes a volume-weighted average price for an instrument.

Syntax

```
VWAP ( price_expression, volume_expression )
```

Table 240. Parameters

price_expression	a numeric expression containing a price to be incorporated into a volume-weighted average.
volume_expression	a numeric expression containing a volume to be used in calculating a volume-weighted average.

VWAP is a trading acronym for Volume-Weighted Average Price, the ratio of the value traded to total volume traded over a particular time horizon (usually one day). It is a measure of the average price a stock traded at over the trading horizon.

VWAP can be measured between any two points in time but is displayed as the one corresponding to elapsed time during the trading day by the information provider.

VWAP is often used in algorithmic trading. The VWAP is calculated using the following formula:

$$P_{\mathrm{VWAP}} = \frac{\sum_j P_j \cdot Q_j}{\sum_j Q_j}$$

where:

PVWAP = Volume Weighted Average Price

Pj = price of trade j

Qj = quantity of trade

j = each individual trade that takes place over the defined period of time, excluding cross trades and basket cross trades.

Sybase extension.

The following example demonstrates how the VWAP function is used to calculate volume weighted averages for data in a CSV file,

where:

id = company trade symbol

x = stock price

y = volume traded

```
-- create input stream schema
CREATE SCHEMA InSchema (
    id STRING,
    x FLOAT,
    y FLOAT
);

-- create output stream schema
CREATE SCHEMA OutSchema (
```

```
    id STRING,
    x FLOAT,
    y FLOAT,
    vwap_result FLOAT
);

-- create input stream
CREATE INPUT STREAM StreamIn
SCHEMA InSchema;

-- create master window
CREATE MASTER WINDOW ResultWindow
SCHEMA OutSchema
KEEP 5 ROWS
;

-- create output stream
CREATE OUTPUT STREAM StreamOut
SCHEMA OutSchema;

-- input stream read data from csv file by ReadFromCsvFileAdapterType
adapter
ATTACH INPUT ADAPTER ReadFromCSVFile TYPE ReadFromCsvFileAdapterType
TO STREAM StreamIn
PROPERTIES
    FILENAME            = "$ProjectFolder\..\data\data.csv",
    TITLEROW            = "false",
    TIMESTAMPCOLUMN     = "false",
    RATE                = "1",
    USECURRENTTIMESTAMP = "true"
;

-- output stream write data to csv file by WriteToCsvFileAdapterType
adapter
ATTACH OUTPUT ADAPTER WriteToCSVFile TYPE WriteToCsvFileAdapterType
TO STREAM StreamOut
PROPERTIES
    FILENAME = "$ProjectFolder\..\data\result.csv"
;

-- insert the calculated result to window
INSERT INTO ResultWindow
SELECT id,x,y,
vwap(y,1)
FROM StreamIn
KEEP 5 ROWS;

INSERT INTO StreamOut
SELECT *
FROM ResultWindow;
```

Sample result:

Timestamp	id	x	y	vwap_result

...	...	...	...	...
1.25922E+15	IBM	75.15	9800	9800
1.25922E+15	SWY	21.7	400	5100
1.25922E+15	MW	37.08	200	3466.666667
...	...	...	...	...

WEIGHTED_AVG()

Aggregate. Calculates an arithmetically (or linearly) weighted average.

Syntax

```
WEIGHTED_AVG( expression )
```

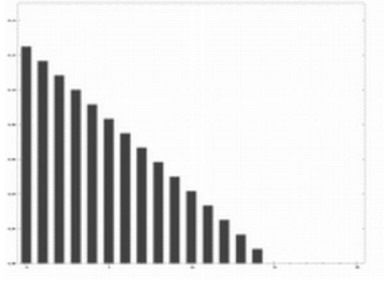
Table 241. Parameters

expression	A numeric expression for which a weighted value is to be computed.

An arithmetic weighted average is any average that has multiplying factors to give different weights to different data points. But in technical analysis a weighted moving average (WMA) has the specific meaning of weights which decrease arithmetically. In an n-day WMA the latest day has weight n, the second latest n − 1, etc, down to zero:

$$WMA_M = \frac{np_M + (n-1)p_{M-1} + \cdots + 2p_{M-n+2} + p_{M-n+1}}{n + (n-1) + \cdots + 2 + 1}$$

The graph at the right shows how the weights decrease, from highest weight for the most recent data points, down to zero. It can be compared to the weights in the exponential moving average which follows. It's important to remember that for more exaggerated weighting on the ongoing values, you may use an EMA. You could also average two or more WMA together.



Sybase extension.

The following example demonstrates how to use the WEIGHTED_AVG functionto calculate the weighted average of data in a CSV file:

```
-- create input stream schema
CREATE SCHEMA InSchema (
    id STRING,
    x FLOAT,
    y FLOAT
);

-- create output stream schema
CREATE SCHEMA OutSchema (
    id STRING,
    x FLOAT,
    y FLOAT,
    weighted_avg_result FLOAT,
);

-- create input stream
CREATE INPUT STREAM StreamIn
SCHEMA InSchema;

-- create master window
CREATE MASTER WINDOW ResultWindow
SCHEMA OutSchema
KEEP 5 ROWS
;

-- create output stream
CREATE OUTPUT STREAM StreamOut
SCHEMA OutSchema;

-- input stream read data from csv file by ReadFromCsvFileAdapterType
adapter
ATTACH INPUT ADAPTER ReadFromCSVFile TYPE ReadFromCsvFileAdapterType
TO STREAM StreamIn
PROPERTIES
    FILENAME            = "$ProjectFolder\..\data\data.csv",
    TITLEROW            = "false",
    TIMESTAMPCOLUMN     = "false",
    RATE                = "1",
    USECURRENTTIMESTAMP = "true"
;

-- output stream write data to csv file by WriteToCsvFileAdapterType
adapter
ATTACH OUTPUT ADAPTER WriteToCSVFile TYPE WriteToCsvFileAdapterType
TO STREAM StreamOut
PROPERTIES
    FILENAME = "$ProjectFolder\..\data\result.csv"
;

-- insert the calculated result to window
INSERT INTO ResultWindow
SELECT id,x,y,
weighted_avg(x)
```

```
FROM StreamIn
KEEP 5 ROWS;

INSERT INTO StreamOut
SELECT *
FROM ResultWindow;
```

XMLAGG()

Aggregate. Returns a single XML result aggregated over a group of rows.

Syntax

XMLAGG(*expression* **)**

Table 242. Parameter

expression	An expression that evaluates to an XML value.

Table 243. Data Types

Return	*expression*
XML	XML

Usage

All results are concatenated (ignoring Nulls). Note that windows are not required with XMLAGG.

See Also

- XMLELEMENT

Example

The following example shows the construction of an element called "orders", where the children of orders are "orderid" elements, one of each from each resulting row of the join:

```
INSERT INTO OutStream
SELECT XMLELEMENT("orders",
    XMLAGG(XMLELEMENT("orderid", orders.orderid)))
FROM Orders KEEP 1 HOUR, Shipments;
```

XMLATTRIBUTES()

Scalar. Appends one or more attributes to an XML element being constructed with XMLELEMENT().

Syntax

XMLATTRIBUTES(*value* **AS** *name* [, ...]**)**

Table 244. Parameters

value	The value of the attribute. If Null, the function generates no attribute.
name	The unique name of the attribute. The name must obey XML naming conventions and be in-scope for XML name spaces.

Table 245. Data Types

Return	*value*	*name*
XML	String	String

See Also

- XMLELEMENT

Example

The following example shows an attribute being inserted during the construction of an XML element:

```
INSERT INTO OutStream
SELECT XMLELEMENT("order",
   XMLATTRIBUTES(orders.customerName AS "custname"),
   XMLELEMENT("orderid", orders.orderid))
FROM Orders;
```

XMLCOMMENT()

Scalar. Appends a comment to an xml element being constructed with XMLELEMENT().

Description

Scalar. Appends a comment to an XML element being constructed with XMLELEMENT().

Syntax
XMLCOMMENT(*comment* **)**

Table 246. Parameter

comment	The comment.

Table 247. Data Types

Return	*comment*
XML	String

See Also

- XMLELEMENT

Example
The following example shows a comment node being inserted during the construction of an XML element:

```
INSERT INTO OutStream
SELECT XMLELEMENT("billing",
   XMLCOMMENT('put your comment here'),
   'some value',
   XMLELEMENT("done"))
FROM Billing;
...
```

XMLCONCAT()

Scalar. Concatenates XML expressions.

Syntax
XMLCONCAT(*expression, expression* [, ...]**)**

Table 248. Parameter

expression	An expression that evaluates to an XML value. Ignored if Null. If a sequence of XML trees, Sybase CEP Engine concatenates each element tree in order.

Table 249. Data Types

Return	*expression*
XML	XML

Example

The following example shows concatenation of two XML elements:

```
INSERT INTO OutStream
SELECT XMLCONCAT(XMLELEMENT("name", orders.customerName),
    XMLELEMENT("city", orders.customerCity))
FROM Orders;
```

XMLDELETE()

Scalar. Deletes element trees from an XML value.

Syntax

XMLDELETE(*value, xpath* **)**

Table 250. Parameters

value	The XML to modify.
xpath	An XPATH string specifying what to delete from value.

Table 251. Data Types

Return	*value*	*xpath*
XML	XML	String

Usage

Sybase CEP Engine uses XPATH to locate the element trees specified by xpath in value. Sybase CEP Engine logs a warning for any non-element nodes it finds in value, but otherwise ignores them. It deletes each element tree it finds matching xpath, in document order. If both a tree and an ancestor of the tree match, Sybase CEP Engine only deletes the ancestor.

See Also

- XMLUPDATE
- XMLINSERT

Example
The following example shows removing the shipping METHOD nodes from the value in the orderVal column:

```
INSERT INTO OutStream
SELECT XMLDELETE (O.orderVal, '//METHOD')
FROM O;
```

XMLELEMENT()

Scalar. Constructs an XML element tree.

Syntax
XMLELEMENT([**NAME**] *"name"* [, namespace] [, xmlattribute] { [, { xmlpi | xmlcomment | content }] [, ...] } **)**

Table 252. Parameters

name	The name of the element. Must conform to XML name conventions including XML name spaces.
namespace	Defines a name space for this element. See namespace for more information.
xmlattribute	Defines an XML attribute. See XMLATTRIBUTES() for more information.
xmlpi	An XML processing instruction. See XMLPI() for more information.
xmlcomment	A comment. See XMLCOMMENT() for more information.
content	Content for the element.

Table 253. Data Types

Return	*name*	*content*
XML	String	String
		XML

namespace
XMLNAMESPACES({ *uri* **AS** *prefix* | **DEFAULT** *uri* | **NO DEFAULT** } [, ...] **)**

Table 254. Parameters

uri	The URI of an XML name space.
prefix	A literal specifying the prefix for the name space. Each prefix must be unique for each call to this function.

Table 255. Data Types

Return	*uri*	*prefix*
String	String	String

See Also

- XMLAGG
- XMLPI
- XMLATTRIBUTES
- XMLCOMMENT

Example

The following example shows the construction of an XML element tree:

```
SELECT
   XMLELEMENT("order",
    XMLELEMENT("orderid", orders.orderid),
    XMLELEMENT("billing",
      XMLELEMENT("name", orders.customername),
      XMLELEMENT("city", orders.customerCity),
      XMLELEMENT("state", orders.customerState)
    )
   )
FROM Orders;
```

XMLEXISTS()

Scalar. Determines whether or not the results of an XPATH expression exists in an XML element.

Syntax

XMLEXISTS(*xpath* [, **PASSING** *value*]**)**

Table 256. Parameters

xpath	The XPATH expression.

value	The XML value. Can be omitted when used with XMLTABLE.

Table 257. Data Types

Return	*xpath*	*value*
Boolean	String	XML

Usage

Sybase CEP evaluates xpath and returns True if it locates the results in value, False if it does not, and Null if evaluating xpath results in Null or an error.

See Also

- XMLEXTRACT

Example

The following example shows testing for PA as the state:

```
INSERT INTO OutStream
SELECT orderid, customername
FROM Order2
WHERE XMLEXISTS ('//state[.=''PA'']', PASSING orderval2);
```

XMLEXTRACT()

Scalar. Extracts the portion of an XML value that matches a given XPATH expression.

Syntax

XMLEXTRACT([*value* ,] *xpath*]**)**

Table 258. Parameters

value	The XML value. Can be omitted when used with XMLTABLE.
xpath	The XPATH expression.

Table 259. Data Types

Return	*value*	*xpath*
XML	XML	String

Usage

Sybase CEP Engine evaluates the XPATH expression xpath against the specified value (or the implicit value in XMLTABLE) and returns the matching portion, if found. If xpath is a

sequence, Sybase CEP Engine evaluates the sequence in order and concatenates the results. If value is Null, XMLEXTRACT returns Null. If a partial XPATH result is not an element tree or a sequence of element trees, XPATH generates a warning and ignores the partial result.

See Also

- XMLEXTRACTVALUE
- XMLTABLE Expressions in the FROM Clause

Example
The following example shows an extraction of all "orderid" element nodes:

```
INSERT INTO OutStream
SELECT XMLEXTRACT(OrderVal2,'//orderid')
FROM Orders;
```

XMLEXTRACTVALUE()

Scalar. Extracts the value of an XML element tree that matches a given XPATH expression.

Syntax
XMLEXTRACTVALUE([*value ,*] *xpath*]**)**

Table 260. Parameters

value	The XML value. Can be omitted when used with XMLTABLE.
xpath	The XPATH expression.

Table 261. Data Types

Return	*value*	*xpath*
String	XML	String

Usage
If value is Null, XMLEXTRACTVALUE returns Null. If the XPATH comparison returns more than one possible value, Sybase CEP Engine returns the first value and issues a warning that other values were found.

See Also

- XMLEXTRACT
- XMLTABLE Expressions in the FROM Clause

Example
The following example shows an extraction of all "items" element nodes:

```
INSERT INTO OutStream
SELECT XMLEXTRACTVALUE(OrderVal1,'//items')
FROM Orders;
```

XMLINSERT()

Scalar. Inserts one XML value into another XML value.

Syntax
XMLINSERT(*value , insert,* { **FIRST|LAST|AFTER|BEFORE** } *xpath*])

Table 262. Parameters

value	The original XML value.
insert	The XML value to insert.
xpath	An XPATH expression. insert is inserted into value based on the location of a match for this expression.

Table 263. Data Types

Return	*value*	*insert*	*xpath*
XML	XML	XML	String

Usage
Sybase CEP Engine locates the element nodes in value that match xpath. If Sybase CEP Engine locates any non-element nodes, it logs a warning and ignores the nodes. For each matched element tree, Sybase CEP Engine inserts the specified XML value as follows:

- FIRST: Adds insert as the first child of the matched tree.
- LAST: Adds insert as the last child of the matched tree.
- AFTER: Adds insert after the matched tree (at the same level as the selected tree).
- BEFORE: Adds insert before the matched tree (at the same level as the selected tree).

See Also

- XMLDELETE
- XMLUPDATE

Example
The following example shows adding a tracking number to an order after the METHOD nodes:

```
INSERT INTO OutStream
SELECT XMLINSERT (O.orderVal,
   XMLPARSE ('<TRACKINGNO>T060023456</TRACKINGNO>'),
   AFTER '//METHOD')
FROM O;
```

XMLPATTERNMATCH()

Aggregate. Generates an XML tree contianing all matches detected by a pattern-matching query.

Syntax
XMLPATTERNMATCH([*root*] **)**

Table 264. Parameters

root	The root element name for the generated tree. If omitted, the root is PatternMatch.

Table 265. Data Types

Return	*root*
XML	String

Usage
The XMLPATTERNMATCH function performs causality tracking by generating an XML tree that contains all matches (except non-events) detected by a pattern-matching query. This function is allowed only in the SELECT clause of a query that contains a MATCHING clause. XMLPATTERNMATCH() generates an XML tree for every group of events that produces the match specified in the MATCHING clause, except events specified by the !stream-name syntax.

The root element contains an element for each of the data streams involved in the pattern match. Where the streams in question are aliased, the corresponding element has the name of the stream's alias. Each element has the same name as the corresponding stream (or its alias, if one exists).

Every element associated with a data stream contains a sub-element for each column defined in the stream schema and having the same name. Each of the column sub-elements contains its respective row value. Regardless of whether or not a timestamp column is defined in the

schema, a sub-element called "C8_Timestamp" containing the row timestamp value is automatically created in the element associated with the stream.

Important: While stream and column names are case-insensitive in CCL, XML element names are case-sensitive. In creating the element tree, XMLPATTERNMATCH() uses the same capitalization as is specified in the root-element-name argument, stream aliases or (in the absence of an alias) stream definitions, and schema column definitions.

Example

The following example shows two stream schema definitions, followed by a query that uses the XMLPATTERNMATCH function:

```
CREATE STREAM Trades SCHEMA (Symbol STRING,
  Quantity FLOAT, Price FLOAT);
CREATE STREAM Quotes SCHEMA (Symbol STRING,
  Bid FLOAT, Ask FLOAT);
INSERT INTO TradeMismatch
SELECT XMLPATTERNMATCH()
FROM Quotes as Q, Trades as T
MATCHING [10 seconds: Q, T]
ON Q.Symbol = T.Symbol
WHERE T.Price < Q.Bid OR T.Price > Q.Ask
```

The following sample XML tree was generated by XMLPATTERNMATCH when a pattern match is detected by the query:

```
<PatternMatch>
  <Q>
    <C8_Timestamp>
      2207-11-26 16:23:27.203157
    </C8_Timestamp>
    <Symbol>
      IBM
    </Symbol>
    <Bid>
      102
    </Bid>
    <Ask>
      104
    </Ask>
  </Q>
  <T>
    <C8_Timestamp>
      2207-11-26 16:25:53.307201
    </C8_Timestamp>
    <Symbol>
      IBM
    </Symbol>
    <Quantity>
      100
    </Quantity>
    <Price>
```

```
      98
    </Price>
  </T>
<PatternMatch>
```

XMLPI()

Scalar. Adds a processing instruction node to an XML element being constructed with XMLELEMENT().

Syntax

XMLPI(*target* [, *value*])

Table 266. Parameters

target	The processing instruction.
value	The value.

Table 267. Data Types

Return	*target*	*value*
none	String	String

Usage

Adds a processing instruction node to an XML element. The target must follow the rules of the XMLELEMENT name, and therefore must be quoted. This function can only be used as an argument to XMLELEMENT.

See Also

- XMLELEMENT

Example

The following example shows a simple XML processing instruction:

```
INSERT INTO OutStream
SELECT XMLELEMENT("billing",
   XMLPI("sort", 'ascending'),
   'some value',
   XMLELEMENT("done"))
FROM Orders;
```

XMLSERIALIZE()

Scalar. Converts an XML value to a String. Note that this function is the same as TO_STRING() with an XML parameter.

Syntax

XMLSERIALIZE(*value* **)**

Table 268. Parameters

value	The value to convert.

Table 269. Data Types

Return	*value*
String	XML
	String

See Also

- XMLPARSE

Example

The following example shows an extraction and serialization (to string):

```
INSERT INTO OutStream
SELECT XMLSERIALIZE(XMLEXTRACT(OrderVal2,'//item'))
FROM Orders;
```

XMLTRANSFORM()

Scalar. Applies and XSL transformation (XSLT) to an XML value.

Syntax

XMLTRANSFORM(*value, transform* **)**

Table 270. Parameters

value	The value to modify.
transform	The XSLT string to apply.

Table 271. Data Types

Return	*value*	*transform*
XML	XML	XML

Usage

Any transform returning a value of Null is eliminated from the result. If value is Null, or if the sequence produced is of zero length, then the function returns Null.

See Also

- XMLPARSE
- XMLSERIALIZE

Example

The following example shows a transform. Note: The string inside of XMLPARSE is intended to be one long string:

```
INSERT INTO OutStream
SELECT XMLTRANSFORM(OrderVal1
   XMLPARSE('<xsl:stylesheet version = ''1.0''
 xmlns:xsl=''http://www.w3.org/1999/XSL/Transform''>
      <xsl:template match="/">
      <custname><xsl:value-of select="//name"/></custname>
      <providerID><xsl:value-of select=
      "//orderid"/></providerID>
      </xsl:template>
      </xsl:stylesheet>'))
FROM Trades;
```

XMLUPDATE()

Scalar. Replaces sections of an XML value matching a given XPATH string with another XML value.

Syntax

XMLUPDATE(*value, update, xpath* **)**

Table 272. Parameters

value	The value to modify.
update	The replacement.
xpath	The XPATH string. Element trees in value matching this string are replaced with update.

Table 273. Data Types

Return	*value*	*update*	*xpath*
XML	XML	XML	String

Usage

Sybase CEP Engine locates the element nodes in value that match xpath. If Sybase CEP Engine locates any non-element nodes, it logs a warning and ignores the nodes. Sybase CEP Engine replaces each matched element tree with update. Sybase CEP Engine performs the replacement in document order, so if both a tree and an ancestor of the tree match, Sybase CEP Engine only replaces the ancestor.

See Also

- XMLDELETE
- XMLINSERT

Example

The following example shows changing the shipping METHOD nodes to "air":

```
INSERT INTO OutStream
SELECT XMLUPDATE (O.orderVal,
   XMLPARSE ('<METHOD>air</METHOD>'),
   '//METHOD')
FROM O;
```

XPATH()

Scalar. Returns the value in the string representation of an XML element matching an XPATH expression.

Syntax

XPATH(*value, xpath* **)**

Table 274. Parameters

value	The value to search.
xpath	The XPATH string. The value of the element in value matching this string is extracted.

Table 275. Data Types

Return	*value*	*xpath*
String	String	String

Example

The following example uses the XPATH function to return the first item element in the transaction element contained in the STRING column identified by Orders.lineitems:

```
INSERT INTO OutStream
SELECT XPATH(Orders.lineitems, "//transaction/item[1]")
FROM Orders;
```

YEAR()

Scalar. Returns an integer representing the year as extracted from a timestamp value.

Syntax

YEAR(*timestamp* [*, timezone*] **)**

Table 276. Parameters

timestamp	An expression that evaluates to a timestamp value.
timezone	A string representing the time zone. If omitted, Sybase CEP Engine assumes the local time zone. See "Sybase CEP Time Zone Database" in the Sybase CEP Integration Guide for more information about valid time zone strings.

Table 277. Data Types

Return	*timestamp*	*timezone*
Integer	Timestamp	String

Usage

If either parameter is NULL, the function returns NULL.

Example

```
INSERT INTO OutStream
SELECT YEAR(InStream.OrderTime)
FROM InStream;
```

Sybase CEP Function Language Function Language

Sybase CEP Function Language creates CCL user-defined functions which are used interchangeable with predefined functions.

Use the Sybase CEP Function Language (CFL) to create CCL user-defined functions (UDFs), which can be used interchangeably with predefined functions. Use the Create Function statement to create a CCL UDF. The CCL Function Language consists of simple statements and block statements.

Simple Statements

Simple statements include:

- Create Variable statement: Declares a variable name and data type and optionally assigns a value to the variable.
- Assignment statement: Assigns a value to a variable created with the Create Variable statement.
- Return statement: Indicates the end of function execution.
- Break statement: Exits a WHILE loop (used only inside a While statement).
- Continue statement: Interrupts the execution of a WHILE loop and forces a re-evaluation of the WHILE condition (used only inside a While statement).

The following rules apply to all simple CFL statements:

- Simple statements always appear inside the Create Function statement, either directly or as part of an enclosing block statement.
- Except where otherwise specified, expressions used in the simple statements can include any of the following:
  - CCL operators.
  - Function parameters declared in the Create Function statement.
  - Variables defined with a CFL Create Variable statement, whose scope makes them visible to the referring statement.
  - Predefined Sybase CEP scalar functions (aggregate and miscellaneous predefined functions cannot be used in CCL UDF definitions).
  - Other CCL UDFs.

Block Statements

Block statements include:

- If statement: Provides alternative execution based on one or more conditional expressions.

- Case statement: Provides alternative execution based on one or more conditional expressions.
- While statement: Provides repetitive execution (a loop) based on a simple conditional expression.
- The following rules apply to all block CFL statements:
- Block statements always appear inside the Create Function statement, either directly or nested in other block statements.
- The conditions specified within block statements are Boolean expressions. These Boolean expressions are subject to the same restrictions as expressions within simple CFL statements.

ASSIGNMENT statement

Changes the value of a variable.

Syntax

```
variable = value;
```

Table 278. Components

variable	The name of a variable defined with a Create Variable statement. The variable must be within the scope of the current code block.
value	An expression that evaluates to a value of the type specified for the variable when it was created, or a type that is implicitly converted to the correct data type (see Implicit Data Type Conversions for more information).

Restrictions

- If the Assignment statement attempts to change the value of a variable that is not within its scope, a compilation error occurs.
- If expression evaluates to a data type that is different from the variable, a compilation error occurs.

See Also

- Create Variable Statement

Examples

The following example creates a variable called ii, which is visible to the entire function. Assignment statements then assign values to the variable throughout the function.

```
CREATE FUNCTION MyFunction(InputPar FLOAT) RETURNS FLOAT
```

```
   CREATE VARIABLE FLOAT res = SIN(InputPar) + COS(InputPar);
   CREATE VARIABLE INTEGER ii;
   IF InputPar > 0 THEN
      ii = ROUND(InputPar, 0);
      WHILE(ii > 0) DO
         res = res + ii * sin(ii / 2);
         ii = ii - 1;
      END;
   ELSE
      ii = ROUND(-InputPar, 0);
      WHILE(ii > 0) DO
         res = res + ii * cos(ii / 2);
         ii = ii - 1 ;
      END;
   END;
   RETURN res;
END FUNCTION;
```

BREAK statement

Forces an immediate exit from a WHILE loop. Only valid inside the DO clause of a WHILE statement.

Syntax
BREAK;

Example

```
CREATE FUNCTION BreakContinueDemo(
   Value Integer, SkipMultiples Integer, StopValue Integer )
RETURNS INTEGER
BEGIN
   CREATE VARIABLE INTEGER Result = 0;
   WHILE 0 < Value DO
      IF VALUE = StopValue THEN
          BREAK;
      END IF;
      IF SkipMultiples != 0
        and VALUE mod SkipMultiples = 0 THEN
         Value  = Value - 1;
         CONTINUE;
      END IF;
      Result = Result + Value;
      Value  = Value - 1;
   END WHILE;
   RETURN Result;
END FUNCTION;
```

CASE statement

Executes the frist statement block whose condition is true.

Syntax

```
CASE { WHEN condition THEN [cfl_statement] [...] } [...]
[ ELSE [cfl_statement] [...] ] END [CASE] ;
```

Table 279. Components

condition	A Boolean expression, consisting of CCL operators, variables defined within the UDF definition, and/or parameters defined in the Create Function Statement.
cfl_statement	A CFL statement. See Create Function Statement for more information.

Usage

The CFL Case statement is similar to the CCL CASE expression, but uses Sybase CEP Function Language components in its conditions and THEN and ELSE clauses, and is limited in scope according to the same rules as other block CFL statements.

The Case statement specifies one or more WHEN clauses, each of which contains a Boolean condition and a THEN subclause with a block of CFL statements. The WHEN clauses are evaluated in sequence.

The first clause that evaluates to true executes the block of statements associated with its THEN subclause. All subsequent WHEN clauses are then ignored.

If none of the WHEN clause conditions are true, the statements associated with the ELSE clause (if one is specified) are executed.

If no ELSE clause exists and none of the WHEN clause conditions are true, no statements are executed.

See Also

- If Statement

Examples

The following example defines three cases, which result in a return of -1, 0, or 1, depending on the value of the passed-in parameter.

```
CREATE FUNCTION Compare(Value INTEGER)
RETURNS INTEGER
   CASE
```

```
   WHEN Value < 0 THEN RETURN -1
   WHEN Value =0 THEN RETUNRN 0;
   ELSE RETURN 1;
   END CASE;
END FUNCTION;
```

CONTINUE statement

Interrupts the execution of a WHILE loop and forces a reevaluation of the WHILE condition. Only valid inside the DO clause of a While Statement.

Syntax

CONTINUE;

Usage

The Continue statement interrupts the execution of a WHILE loop and re-evaluates the WHILE condition. If the condition is still true, the next execution of the loop begins. If the condition is false, then the While statement is exited and the next statement in the function outside of the loop is executed.

Example

```
CREATE FUNCTION BreakContinueDemo(
   Value Integer, SkipMultiples Integer, StopValue Integer )
RETURNS INTEGER
BEGIN
   CREATE VARIABLE INTEGER Result = 0;
   WHILE 0 < Value DO
      IF VALUE = StopValue THEN
          BREAK;
      END IF;
      IF SkipMultiples != 0
        and VALUE mod SkipMultiples = 0 THEN
         Value  = Value - 1;
         CONTINUE;
      END IF;
      Result = Result + Value;
      Value  = Value - 1;
   END WHILE;
   RETURN Result;
END FUNCTION;
```

Create Variables statement (within function)

Defines a variable within the function and initializes its variable.

Syntax

```
CREATE VARIABLE  data_type name [= value] ;
```

Table 280. Components

data_type	The CCL data type of the variable.
name	The name of the variable.
value	A literal of the specified type.

Usage

This statement creates a variable which can be used by subsequent CFL statements inside the block. As with other CFL statements, the scope of this form of the Create Variable statement is the block. A different, though syntactically identical, usage of the Create Variable statement is within a query module. For more information, see Create Variable Statement. The variable is created by defining its data type and name. Optionally, the variable can also be initialized with a constant or expression of the same type as variable. (Literals of type STRING must be enclosed in quotation marks.) If a variable is not initialized in the Create Variable statement, the variable's value is initialized to NULL. The variable value can be set later using the Assignment Statement.

Restrictions

- The variable must be created before being used in other CFL statements in the block.
- A variable name must be unique within the Create Function statement.

See Also

- Assignment Statement

Examples

The following example creates several variables. The variable called res is visible to the whole function, but the two occurrences of variable ii are visible only in the respective blocks of the If statement.

```
CREATE FUNCTION MyFunction(InputPar FLOAT) RETURNS FLOAT;
   CREATE VARIABLE FLOAT res = SIN(InputPar) + COS(InputPar);
   IF InputPar > 0 THEN
      CREATE VARIABLE INTEGER ii = TO_INTEGER(ROUND(InputPar, 0));
      WHILE(ii > 0) DO
         res = res + ii * sin(ii / 2);
```

```
         ii = ii - 1;
      END;
   ELSE
      CREATE VARIABLE INTEGER ii = TO_INTEGER(ROUND(-InputPar, 0));
      WHILE(ii > 0) DO
         res = res + ii * cos(ii / 2);
         ii = ii - 1 ;
      END;
   END;
   RETURN res;
END FUNCTION;
```

CFL Variable Scope

CFL variable Scope describes the part of the UDF where the variable can be used.

The scope of a CFL variable describes the part of the UDF where the variable can be used. All variables have scope only within the block in which they are defined, and only after they are defined in the block (forward references are not supported). A block is a set of statements contained within:

- A CCL UDF defined in a Create Function statement.
- Either the THEN or ELSE clauses of an If statement.
- Any one of the WHEN clauses or the ELSE clause of a Case statement.
- A While statement.

The conditions that appear in the If statement, Case statement, and While statement are in the scope of the surrounding block or function.

The scope of all parameters to any function and all variables that are defined outside of any nested block in the Create Function statement itself is the entire UDF. The scope of all variables that are defined inside any block except the Create Function statement itself is that block and any blocks nested inside that block.

Any variable that is created inside the Create Function statement or any nested block can be used only after it has been created (forward references are not permitted).

IF statement

Executes the first statement block whose condition is true.

Syntax

```
IF ( condition ) THEN { cfl_statement } [, ...] [ ELSEIF
(condition) THEN { cfl_statement } [, ...] ] [ ELSE
{ cfl_statement } [, ...] ] END [IF] ;
```

Table 281. Components

condition	A Boolean expression, consisting of CCL operators, variables defined within the UDF, and/or parameters defined in the Create Function statement.
cfl_statement	Any valid CFL statement. See CCL Function Language Overview and Create Function Statement for more information.

Usage

The CFL If statement is similar to the CCL IF expression, but uses CFL components in its conditions and THEN, ELSEIF, and ELSE clauses, and is limited in scope according to the same rules as other CFL block statements.

The IF clause of the statement specifies a Boolean condition. If this condition is true, the block of statements after the first THEN clause are executed.

If the condition is false, the conditions of the ELSEIF clauses (if any are specified) are evaluated in sequence. The first of these conditions that evaluates to true causes the block of statements following the associated THEN clause to be executed.

If no ELSEIF clauses are used, or none of the Boolean conditions are true, the block of statements associated with the ELSE clause are executed, if one is specified.

If no ELSE clause exists and none of the Boolean conditions are true, no statements are executed.

See Also

- Case Statement

Examples

The following example uses Return statements inside an If statement, resulting in a function that returns a value of -1, 0, or 1.

```
CREATE FUNCTION Compare(VALUE integer)
RETURNS INTEGER
   IF ( value < 0 ) THEN
      RETURN -1;
   ELSEIF ( value = 0 ) THEN
      RETURN 0;
   ELSE
      RETURN 1;
   END IF;
END FUNCTION;
```

RETURN statement

Returns control from a function with the specified return value.

Syntax

```
RETURN value ;
```

Table 282. Components

value	An expression that evaluates to a value of the same data type as specified in the RETURNS clause of the Create Function statement.

Usage

The Return statement causes the current invocation of the function to finish and returns the value of the expression to the caller. The expression must be of the same data type as specified in the RETURNS clause for the function.

WHILE statement

Executes a block of statements while a specified condition remains true.

Syntax

```
WHILE condition DO [cfl_statement] [...] END [WHILE] ;
```

Table 283. Components

condition	A Boolean expression containing CCL operators, variables defined within the CCL function definition, and/or parameters defined in the Create Function Statement.
cfl_statement	A CFL statement. See CCL Function Language Overview and Create Function Statement for more information.

Usage

The CFL While statement creates a loop which executes the CFL statements contained in the DO clause, as long as the specified condition remains true.

The condition is evaluated before the loop is executed for the first time, and then every time the loop is about to be repeated. Since the condition is evaluated before the first execution, it is possible that the statement block will not be executed at all (if the condition is false on the initial evaluation).

Danger! You can create a While statement that never exits (an"endless loop). Always avoid this situation by making sure that the loop contains sufficient logic so that the condition will ultimately evaluate as false.

CFL includes two simple statements that can only be used inside the DO clause of a While statement: the Break statement, which forces an immediate exit from the loop, and the Continue statement, which interrupts the execution of the loop and forces an immediate re-evaluation of the WHILE condition.

See Also

- Break Statement
- Continue Statement

Examples

Here is an illustration of a While statement:

```
CREATE FUNCTION Pow(Value FLOAT, Power INTEGER)
RETURNS FLOAT
   CREATE VARIABLE FLOAT Result =1;
   IF (Power < 0) THEN
      Power = -Power;
      Value = 1/Value
   END IF;
   WHILE 0 < Power DO
      Result = Result * Value
      Power = Power -1;
   END WHILE;
   RETURN Result;
END FUNCTION;
```

Documentation Tags

Sybase CEP uses a variety of documentation tags to label CCL components.

CCL Documentation Tags

Labels, identifies, and provides comments regarding various CCL components.

CCL modules can include documentation tags that label, identify, and provide comments regarding various CCL components. A few of these tags are interpreted by Sybase CEP Studio, which displays the text provided by the tags in combination with the associated components. You can use CCL documentation to help organize and structure your comments.

The following characteristics apply to all documentation tags, except where noted otherwise.

- Documentation tags are entered within query module .ccl files, and are set off by special characters similar to comments.
- Documentation tags appear in *tag blocks*. A tag block is a continuous group of lines marked with documentation tag characters.
- Any of the following methods can be used to designate tag blocks:

	Beginning of Tag Block	End of Tag Block
Method 1:	Three hyphens (---) on a new line.	New line, which is *not* immediately followed by another documentation tag line.
Method 2:	Three slashes (///) on a new line.	New line, which is *not* immediately followed by another documentation tag line.
Method 3:	Two hyphens and an exclamation point (--!) on a new line.	New line, which is *not* immediately followed by another documentation tag line.
Method 4:	Two slashes and an exclamation point (//!) on a new line.	New line, which is *not* immediately followed by another documentation tag line.
Method 5:	A slash and two asterisks (/**).	An asterisk and slash (*/).

Note that leading spaces are ignored for methods 1 through 4. When the tag lines are processed:

- Their markings are stripped, along with any white spaces surrounding them.

    - Any multiple blank lines within the tag block are stripped to one blank line and multiple blank spaces are stripped to one space. If the resulting tag block contains one or more blank lines, the fragments of text separated by the blank lines are interpreted as separate paragraphs. It is possible to use this feature to create multi-paragraph documentation tags. For example, the first paragraph could be used as a summary and a second paragraph to provide a more detailed description. However, note that second and subsequent paragraphs are not displayed in the Sybase CEP Studio Explorer View.
    - If Method 5 is used to designate documentation tag lines, any additional asterisks, appearing in a sequence with the asterisks that mark the beginning or end of the tag block are stripped.
    - If Method 5 is used to designate documentation tag lines, any leading asterisks at the beginning of the second or subsequent lines of the tag block are also stripped.
- The tag marker in methods 1-4, or the opening tag marker in method 5, is followed by one or more valid documentation tags. Tags are made up of the following components:
    - All tags begin with an initial "at" sign (@) (the "at" sign must appear as the first character after the tag marker, not counting white space, to be interpreted as the beginning of a tag; all other "at" signs within the tag markers are treated as literal text), followed immediately by a valid *tag name* with no intervening space.
    - Some tags require that the tag name be followed by an *identifier*, consisting of one or more non-space characters, which identify the module or CCL statement component being tagged.
    - Most (but not all) tag types accept one or more lines of *tag text*.
- Documentation tags can be applied to modules or to some, or all types of CCL statements, depending on the tag type.
    - Tags that apply to statements must appear before the statements to which they apply with no other intervening statements.
    - No tags can appear after the last statement in the module.
    - Tags that apply to the module must appear inside the same tag block.
- No statement can have more than one associated tag with the same tag type and identifier.

@author

Lists the authors associated with a module. Any @author tags must be in the same tag block as the @module tag.

Description

Lists the authors associated with a module. Any @author tags must be in the same tag block as the @module tag.

Syntax

`@author` *`string`*

Example
The following example lists the authors of the MyModule module:

```
---@module J. Doe's Module
---@author J. Doe and Development Team
```

@category

The @category documentation tag classifies the current module as belonging to a named category. Categories can be used to group related modules together. A module can have no more than one @category tag, which must appear in the same tag block as the @module tag.

SYNTAX

```
@category category-name
```

category-name: The name of a category.

Example
The following example assigns a category to the MyModule module:

```
///@module My RFID Module
///@category RFID Tracking
```

@column

Associated with Create schema, stream, and window statements to label the columns defined by the statement.

SYNTAX

```
@column column-name text
```

column-name: The name of a column in the associated schema, stream, or window.

text: Text to display in association with the specified column.

DESCRIPTION
One or more @column documentation tags can be associated with any of the following statements to label the columns defined by the statement:

- Create Schema Statement.
- Create Stream Statement.
- Create Window Statement.

In all cases the documentation tags must precede the statement whose columns are being labeled and should only contain references to columns defined within the statements. If a Create Stream Statement or Create Window Statement does not have any associated @column tags, but the schema used by the stream or window is tagged, the @column documentation tags associated with the schema are inherited by the window or stream.

Example

The following example labels the Symbol and Price columns of the TradeSchema schema:

```
--!@column Symbol Stock Symbol
--!@column Price Current Price
CREATE SCHEMA TradeSchema (Symbol STRING, Price FLOAT);
```

@description

Associates a description with a module, or with any statement within the module.

SYNTAX

```
@description text
```

text: Text that will be displayed in association with the specified module or statement.

DESCRIPTION

The @description documentation tag associates a description with a module, or with any statement within the module. Only one @description tag can be associated with a module or a statement. If the description is associated with a module, the @description tag must appear in the same tag block as the @module tag. Avoid long description text strings as they may be truncated.

Examples

The following example provides a description of the MyModule module:

```
//!@module MyModule
//!@description This is a further module description
```

This example provides a description of the query that follows it:

```
//!@description pre-connect message confirmation
INSERT INTO MsgOut
SELECT
    "OK",
    "Pre-Connect"
FROM
    MsgIn
WHERE
    MessageType = "OK?" AND Data="Pre-Connect";
```

@module

Specifies the name that should be displayed in association with the module.

SYNTAX

```
@module module-name
```

module-name: A name for the current module.

DESCRIPTION

The @module documentation tag specifies the name that should be displayed in association with the module. A module can have no more than one @module tag. Any other tags associated with the module (but not the with CCL statements in the module) must appear in the same tag block as the @module tag.

Example

The following example associates the name Module1 with the current module:

```
/**@module Another Module*/
```

@name

Specifies the name that should be displayed in association with the CCL statement that follows.

SYNTAX

```
@name statement-name
```

statement-name: A name for the following statement.

DESCRIPTION

The @name documentation tag specifies the name that should be displayed in association with the CCL statement that follows. A CCL statement can have no more than one associated @name tag.

Example

The following example associates the name FirstQuery with the query that follows:

```
---@name FirstQuery
INSERT INTO TempOut
```

```
SELECT *
FROM TempIn;
```

Quick References

A collection of reference guides for proper CCL usage.

CCL Statements Syntax Summary

Summerizes CCL statements syntax.

Attach Adapter

```
ATTACH { INPUT | OUTPUT } ADAPTER name TYPE type TO STREAM
stream [PROPERTIES { prop = value } [, ...] ] ;
```

Create Function

```
CREATE FUNCTION fname ( [ pname type [, ...] ] ) RETURNS type {
statement [, ...] } END FUNCTION ;
```

statement

```
{ create_variable_statement | assignment_statement |
return_statement | break_statement | continue_statement } | {
if_statement | case_statement | while_statement }
```

Create Parameter

```
CREATE PARAMETER data_type name [= value] ;
```

Create Schema

```
CREATE SCHEMA  name { (col_name type [, ...]) | INHERITS
[FROM] schema_name [, ...] [ (col_name type [, ...]) ] } ;
```

Create Stream

```
CREATE [ INPUT | OUTPUT | LOCAL ] STREAM name [schema_clause]
[PROPERTIES prop_def [, ...] ] ;
```

prop_def

```
{ GUARANTEED DELIVERY = {INHERIT | ENABLE | DISABLE} } |
{ GUARANTEED DELIVERY MAXIMUM QUEUE SIZE = size } |
{ GUARANTEED DELIVERY MAXIMUM AGE = age } | { MAXIMUM DELAY =
max_delay } | { OUT OF ORDER DELAY = delay } |
{ SYNCHRONIZATION = { INHERIT | IN ORDER | OUT OF ORDER | USE
SERVER TIMESTAMP } } | { FILTERCOLUMNS = " col_name [, ...]
" }
```

Create Variable

```
CREATE VARIABLE  data_type name [= value] ;
```

Create Window

```
CREATE [ PUBLIC | MASTER ] WINDOW win_name schema_clause
{ keep_clause [, keep_clause] } | { MIRROR master_window }
[INSERT REMOVED [ROWS] INTO name ] [ properties_clause ]
```

properties_clause

```
PROPERTIES [ INDEXCOLUMNS=" col_name [, ...] " ]
[ FILTERCOLUMNS=" col_name [, ...] " | [ FILTER="  value
[, ...] " ] [ FILTEREXPR=" expression " ] ]
```

Database

```
exec_clause select_clause from_clause [matching_clause]
[on_clause] [where_clause] [group_by_clause] [having_clause]
[order_by_clause] [limit_clause] [output_clause] ;
```

Delete

```
on_clause [ when_clause ] DELETE FROM window_name
[ where_clause ] ;
```

Import

```
IMPORT file ;
```

Insert Values

```
insert_clause values_clause output_clause ;
```

Query

```
insert_clause select_clause from_clause [matching_clause]
[on_clause] [where_clause] [group_by_clause] [having_clause]
[order_by_clause] [limit_clause] [output_clause] ;
```

Remote Procedure

```
exec_rem_proc_clause select_clause from_clause
[matching_clause] [on_clause] [where_clause]
[group_by_clause] [having_clause] [order_by_clause]
[limit_clause] [output_clause] ;
```

Set Variable

```
on_clause [when_clause] set_clause ;
```

Update Window
```
on_clause [when_clause] update_clause set_clause
[where_clause] [otherwise_insert_clause] ;
```

CCL Clauses Syntax Summary

Summerizes the CCL Clauses Syntax.

Cache
```
CACHE { { CLEAR [ALL] ON source } | { MAXIMUM AGE age} |
{ MAXIMUM { MEMORY USAGE | SIZE } limit } } [, ...]
```

Execute Remote Procedure
```
EXECUTE REMOTE PROCEDURE "service"
```

Execute Statement Database
```
EXECUTE STATEMENT DATABASE "service" [[ statements ]]
```

From: Comma-Separated Syntax
```
FROM { stream [ [AS] alias] | stream [ [AS] alias] keep_clause
[keep_clause] | window_name [ [AS] alias] | (  nested_join
[on_clause] ) | xmltable_exp | subquery } [, ...]
```

xmltable_exp
```
{ name [AS] alias } | { stream [AS] alias keep_clause
[keep_clause] } xmltable_func
```

xmltable_func
```
XMLTABLE ( column ROWS xpath COLUMNS { expression AS
out_column } [, ...] )
```

From: Database and Remote Subquery Syntax
```
FROM { { db_sub | rem_sub} { , | [LEFT OUTER] JOIN } stream
[[AS] alias ] } | { stream [[AS] alias ] { , | [RIGHT OUTER]
JOIN} { db_sub | rem_sub} }
```

db_sub
```
( DATABASE "service" schema_clause [[ statements ]]
[cache_clause] ) [AS] alias
```

rem_sub
(REMOTE QUERY "*service*" *schema_clause* ([{*value* [**AS**] *param*} [, ...]]) [*cache_clause*]) [**AS**] *alias*

From: Join Syntax
FROM { *stream* [[**AS**] *alias*] | *stream* [[**AS**] *alias*] *keep_clause* [*keep_clause*] | *window_name* [[**AS**] *alias*] |nested_join | xmltable_exp | *subquery* } [**RIGHT | LEFT | FULL**] [**OUTER**] **JOIN** { *stream* [[**AS**] *alias*] | *stream* [[**AS**] *alias*] *keep_clause* [*keep_clause*] | *window_name* [[**AS**] *alias*] |nested_join | xmltable_exp |subquery }

nested_join
FROM { *stream* [[**AS**] *alias*] | *stream* [[**AS**] *alias*] *keep_clause* [*keep_clause*] | *window_name* [[**AS**] *alias*] |nested_join | xmltable_exp | *subquery* } [**RIGHT | LEFT | FULL**] [**OUTER**] **JOIN** { *stream* [[**AS**] *alias*] | *stream* [[**AS**] *alias*] *keep_clause* [*keep_clause*] | *window_name* [[**AS**] *alias*] |nested_join | xmltable_exp |subquery } [*on_clause*]

subquery
(*select_clause from_clause* [*matching_clause*] [*on_clause*] [*where_clause*] [*group_by_clause*] [*having_clause*] [*order_by_clause*] [*output_clause*]) [**AS**] *alias* [*keep_clause*] [*keep_clause*]

Group By
GROUP BY { *column* | *gettimestamp* } [, ...]

Having
HAVING *boolean*

Insert Into
INSERT INTO *name* [(*column* [, ...])]

Insert When
INSERT { **WHEN** *condition* **THEN** *name* [(*column* [, ...])] } [, ...] [**ELSE** *name* [(*column* [, ...])]]

Keep
time_policy | count_policy

time_policy
```
KEEP { { [EVERY] interval [OFFSET times_list  BY interval] } |
FOR interval_col | UNTIL times_list } [ PER column [, ...] |
UNGROUPED ]
```

times_list
```
time_spec | (time_spec [, ...])
```

time_spec
```
' [ SUN | MON | TUE | WED | THU | FRI | SAT ] hour : minute [ :
second [. fraction ]] [timezone]'
```

count_policy
```
KEEP { [EVERY] count  BUCKETS { BY column } [...] } |
{ { [EVERY] count  ROW[S] } | { LAST [ROW] } | { count
{ LARGEST | SMALLEST } [DISTINCT] ROW[S] { BY column } [...] }
| { ALL [ROW[S]] } [ { PER column } [...] | UNGROUPED ] }
```

Limit
```
LIMIT count [ROW[S]] [ OFFSET skip [ROW[S]] ] [ PER column
[ ... ] ]
```

Matching
```
MATCHING [ ONCE [ ( event [, ...] ) ] [ interval : pattern ] [
on_clause ]
```

pattern
```
[!] { event | ( pattern ) | [ interval : pattern ] } [ {&& | ||
| ,} [!] { event | ( pattern ) | [ interval : pattern ] } ]
[...]
```

On: Trigger Syntax
```
ON name [ [AS] alias ]
```

On: Join Syntax
```
ON inner_condition | source.column = source.column [AND ...]
```

On: Pattern Matching Syntax
```
ON {source.column = source.column [= ...] } [AND ...]
```

Order By
```
ORDER BY column [ ASC[ENDING] | DESC[ENDING] ] [, ...]
```

Otherwise Insert
```
OTHERWISE INSERT value [AS column] [, ...]
```

Output
```
OUTPUT { [ALL] EVERY { count ROW[S] | interval [OFFSET BY
interval] } } | { {AFTER | FIRST WITHIN} {count ROW[S] |
interval} } | { [ALL] AT times_list } [ UNGROUPED | PER column
[...] ]
```

Schema
```
SCHEMA name | 'file' | ( column type [, ...] )
```

Select
```
SELECT { expression [AS column | alias | parameter ] }
[, ...]
```

Set: Set Variable Statement Syntax
```
SET name = value [, ...]
```

Set: Window Syntax
```
SET { value  AS column [, ...] } | { column = value
[, ...] }
```

Update
```
UPDATE window[ [AS] alias ]
```

Values
```
VALUES ( column_expression [, ...] ) [, ...]
```

When
```
WHEN trigger
```

Where
```
WHERE condition
```

Operators and Operand Data Types

List of operators and operand data types.

Operator	Result	Operands
+, - (as unary operators)	Integer	Integer

Operator	Result	Operands
+, - (as unary operators)	Long	Long
+, - (as unary operators)	Float	Float
+, - (as unary operators)	Interval	Interval
+,-,*, /, mod	Integer	Integer, Integer
+,-,*, /, mod	Long	Long, Long
+,-,*, /, ^, mod	Float	Float, Float
+	BLOB	BLOB
^	Float	Integer, Float
^	Float	Long, Float
+,-,mod	Interval	Interval, Interval
/	Float	Interval, Interval
+	Timestamp	Interval, Timestamp
*,/	Interval	Interval, Integer
*,/	Interval	Interval, Long
*,/	Interval	Interval, Float
-	Interval	Timestamp, Timestamp
+,-	Timestamp	Timestamp, Interval
=, !=, <>, <, >, <=, >=	Boolean	Boolean, Boolean
=, !=, <>, <, >, <=, >=	Boolean	Integer, Integer
=, !=, <>, <, >, <=, >=	Boolean	Long, Long
=, !=, <>, <, >, <=, >=	Boolean	Float, Float
=, !=, <>, <, >, <=, >=	Boolean	Interval, Interval
=, !=, <>, <, >, <=, >=	Boolean	String, String
=, !=, <>, <, >, <=, >=	Boolean	Timestamp, Timestamp
=, !=, <>, <, >, <=, >=	Boolean	BLOB
NOT	Boolean	Boolean
AND, OR, XOR	Boolean	Boolean, Boolean
IN	Boolean	Boolean, Boolean

Operator	Result	Operands
IN	Boolean	Float, Float
IN	Boolean	Integer, Integer
IN	Boolean	Long, Long
IN	Boolean	Interval, Interval
IN	Boolean	String, String
IN	Boolean	Timestamp, Timestamp
||, +	String	String, String
[ ]. (previous operator)	Boolean	Integer, Boolean
[ ]. (previous operator)	Float	Integer, Float
[ ]. (previous operator)	Integer	Integer, Integer
[ ]. (previous operator)	Long	Integer, Long
[ ]. (previous operator)	String	Integer, String
[ ]. (previous operator)	Interval	Integer, Interval
[ ]. (previous operator)	Timestamp	Integer, Timestamp
[ ]. (previous operator)	BLOB	Integer, BLOB
[ ]. (previous operator)	XML	Integer, XML
LIKE	Boolean	String, String
REGEXP_LIKE	Boolean	String, String

Note that you cannot divide a TIMESTAMP by an INTERVAL, and you cannot use an expression of the form TIMESTAMP mod INTERVAL.

Timestamp Format Codes

A list of valid components that can be used to specify the format of Sybase CEP Timestamp values.

This section lists valid components that can be used to specify the format of Sybase CEP Timestamp values:

- Specify timestamp formats with either the Sybase CEP time formatting codes or a subset of timestamp conversion codes provided by the C++ strftime() function.

- A valid timestamp specification can contain no more than one occurrence of a code specifying a particular time unit (for example, a code specifying the year).
- All designations of year, month, day, hour, minute, or second can also read a fewer number of digits than is specified by the code. For example, DD reads both two-digit and one-digit day entries.

Sybase CEP Time Formatting Codes

Column Code	Description	Input	Output
MM	Month (01-12; JAN = 01).	Y	Y
YYYY	Four-digit year.	Y	Y
YYY	Last three digits of year.	Y	Y
YY	Last two digits of year.	Y	Y
Y	Last digit of year.	Y	Y
Q	Quarter of year (1, 2, 3, 4; JAN-MAR = 1).	N	Y
MON	Abbreviated name of month (JAN, FEB, and so on).	Y	Y
MONTH	Name of month, padded with blanks to nine characters (JANUARY, FEBRUARY, and so on).	Y	Y
RM	Roman numeral month (I-XII; JAN = I).	Y	Y
WW	Week of year (1-53) where week 1 starts on the first day of the year and continues to the seventh day of the year.	N	Y
W	Week of month (1-5) where week 1 starts on the first day of the month and continues to the seventh.	N	Y
D	Day of week (1-7; SUNDAY=1).	N	Y
DD	Day of month (1-31).	Y	Y
DDD	Day of year (1-366).	N	Y
DAY	Name of day (SUNDAY, MONDAY, and so on).	Y	Y
DY	Abbreviated name of day (SUN, MON, and so on).	Y	Y
HH	Hour of day (1-12).	Y	Y
HH12	Hour of day (1-12).	Y	Y
HH24	Hour of day (0-23).	Y	Y

Column Code	Description	Input	Output
AM	Meridian indicator (AM/PM).	Y	Y
PM	Meridian indicator (AM/PM).	Y	Y
MI	Minute (0-59).	Y	Y
SS	Second (0-59).	Y	Y
SSSSS	Seconds past midnight (0-86399).	Y	Y
SE	Seconds since epoch (January 1, 1970 UTC). This format can only be used by itself, with the FF format, and/or with the time zone codes TZD, TZR, TZH and TZM.	Y	Y
MIC	Microseconds since epoch (January 1, 1970 UTC).	Y	Y
FF	Fractions of seconds (0-999999). When used in output, FF produces six digits for microseconds. FFFF produces twelve digits, repeating the six digits for microseconds twice. (In most circumstances, this is not the desired effect.) When used in input, FF collects all digits until a non-digit is detected, and then uses only the first six, discarding the rest.	Y	Y
FF[1-9]	Fractions of seconds. For output only, produces the specified number of digits, rounding or padding with trailing zeroes as needed.	N	Y
MS	Milliseconds since epoch (January 1, 1970 UTC). When used for input, this format code can only be combined with FF (microseconds) and the timezone codes TZD, TZR, TZH, TZM. All other format code combinations generate errors. Furthermore, when MS is used with FF, the MS code must precede the FF code: for example, MS.FF.	Y	Y
FM	Fill mode toggle: suppress zeros and blanks or not (default: not).	Y	Y
FX	Exact mode toggle: match case and punctuations exactly (default: not).	Y	Y
RR	Lets you store 20th century dates in the 21st century using only two digits.	Y	N

Column Code	Description	Input	Output
RRRR	Round year. Accepts either four-digit or two-digit input. If two-digit, provides the same return as RR.	Y	N
TZD	Abbreviated time zone designator such as PST.	Y	Y
TZH	Time zone hour displacement. For example, -5 indicates a time zone five hours earlier than GMT.	N	Y
TZM	Time zone hour and minute displacement. For example, -5:30 indicates a time zone that is five hours and 30 minutes earlier than GMT.	N	Y
TZR	Time zone region name. For example, US/Pacific for PST.	N	Y

Strftime() Timestamp Conversion Codes

Instead of using Sybase CEP time formatting codes, output timestamp formats can be specified using a subset of the C++ strftime() function codes. The following rules apply:

- Sybase CEP Engine treats any timestamp format specification that includes a percent sign (%) as a strftime() code.
- Strings that include a strftime() code string cannot also include Sybase CEP time formatting codes.
- Some strftime() codes are valid only on Microsoft Windows or only on UNIX-like operating systems. Different implementations of strftime() also include minor differences in code interpretation. Sybase CEP Engine attempts to processes whatever formatting codes you supply without trying to determine if they are valid for the platform on which Sybase CEP Engine is running. To avoid errors, make sure that the codes you provide meet the requirements for your platform and that Sybase CEP Server and Sybase CEP Studio are running on the same platform and are using compatible strftime() implementations.
- Sybase CEP does not support strftime() locale handling features: all time zones for formats specified with strftime() are assumed to be the local time zone.
- strftime() codes can be used to specify timestamp output only; they cannot be used to specify timestamp input.

Sybase CEP supports the following strftime() codes:

Strftime() Code	Description
%a	Abbreviated weekday name: for example: "Mon".
%A	Full weekday name: for example "Monday".

Strftime() Code	Description
%b	Abbreviated month name: for example: "Feb".
%B	Full month name: for example "February".

%c	Full date and time string: the output format for this code differs, depending on whether Microsoft Windows or a UNIX-like operating system is being used.
	Microsoft Windows output example:
	`08/26/08 20:00:00`
	UNIX-like operating system output example:
	`Tue Aug 26 20:00:00 2008`

%d	Day of the month, represented as a two-digit decimal integer with a value between 01 and 31.
%H	Hour, represented as a two-digit decimal integer with a value between 00 and 23.
%I	Hour, represented as a two-digit decimal integer with a value between 01 and 12.
%j	Day of the year, represented as a three-digit decimal integer with a value between 001 and 366.
%m	Month, represented as a two-digit decimal integer with a value between 01 and 12.
%M	Minute, represented as a two-digit decimal integer with a value between 00 and 59.
%p	Locale's equivalent of AM or PM.
%S	Second, represented as a two-digit decimal integer with a value between 00 and 61.
%U	Number of the week in the year, represented as a two-digit decimal integer with a value between 00 and 53, with Sunday considered the first day of the week.
%w	Weekday number, represented as a one-digit decimal integer with a value between 0 and 6, with Sunday represented as 0.

Strftime() Code	Description
%W	Number of the week in the year, represented as a two-digit decimal integer with a value between 00 and 53, with Monday considered the first day of the week.

%x	Full date string (no time): The output format for this code differs, depending on whether you are using Microsoft Windows or a UNIX-like operating system.
	Microsoft Windows output example:
	`08/26/08`
	UNIX-like operating system output example:
	`Tue Aug 26 2008`

%X	Full time string (no date).
%y	Year, without the century, represented as a two-digit decimal number with a value between 00 and 99.
%Y	Year, with the century, represented as a four-digit decimal number.
%%	Replaced by %.

Reserved Words

Words which are used to mark the structure of query.

Reserved words (also known as keywords) are used to mark the structure of a query. As such, reserved words cannot be used as identifiers for any CCL objects. Identifiers that start with either XML or C8_ are also reserved.

ADAPTER
AFTER
ALL
AND
ANY
AS
ASC
ASCENDING
AT
ATTACH
BEFORE
BEGIN
BREAK
BY
CASE
COLUMNS
CONTINUE
CREATE
DATABASE
DAY
DAYS
DEFAULT
DELETE
DESC
DESCENDING
DISTINCT
DO
DOWN
ELSE
ELSEIF
MICROSECOND
MICROSECONDS
MILLISECOND
MILLISECONDS
MINUTE
MINUTES
MIRROR
MOD
MODULE
NO
NOT
NULL
OFFSET
ON
OR
ORDER
OTHERWISE
OUTER
OUTPUT
PARAMETER
PARAMETERS
PASSING
PER
PROCEDURE
PROPERTIES
QUERY
REGEXP_LIKE
REMOTE
RETURN
RETURNS

END
EVERY
EXECUTE
FALSE
FOR
FROM
FULL
FUNCTION
GETBLOBCOLUMNBYNAME
GETBOOLEANCOLUMNBYNAME
GETFLOATCOLUMNBYNAME
GETINTEGERCOLUMNBYNAME
GETINTERVALCOLUMNBYNAME
GETLONGCOLUMNBYNAME
GETSTRINGCOLUMNBYNAME
GETTIMESTAMP
GETTIMESTAMPCOLUMNBYNAME
GETXMLCOLUMNBYNAME
GROUP
HAVING
HOUR
HOURS
IF
IMPORT
IN
INHERITS
INPUT
INSERT
INTO
IS
RIGHT
ROW
ROWS
SCHEMA
SECOND
SECONDS
SELECT
SET
SMALLEST
STARTING
STARTUP
STATEMENT
STREAM
STREAMS
THEN
TO
TRUE
UNGROUPED
UNTIL
UP
UPDATE
VARIABLE
WEEK
WEEKS
WHEN
WHERE
WHILE
WINDOW
WITHIN
XMLATTRIBUTES

JOIN
KEEP
LARGEST
LAST
LEFT
LIKE
LIMIT
LOAD
LOCAL
MATCHING
XMLCOMMENT
XMLELEMENT
XMLEXISTS
XMLINSERT
XMLNAMESPACES
XMLPATTERNMATCH
XMLPI
XMLTABLE
XOR

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y