SYBASE®

Programmer's Guide

Sybase CEP Option R4

DOCUMENT ID: DC01029-01-0400-01
LAST REVISED: February 2010

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

Contents

Preface

This guide teaches you how to program for the Sybase® CEP Engine.

- To get started using Sybase CEP, please see the *Sybase CEP Getting Started* tutorial.
- For Sybase CEP Server and Studio installation and configuration instructions, please see the *Sybase CEP Installation Guide*.
- For Sybase CEP Studio usage information, please see the *Sybase CEP Studio Guide*.
- For more advanced Sybase CEP Engine and CCL programming concepts, please see:
  - *Sybase CEP CCL Reference*.
  - *Sybase CEP Integration Guide*.

An Introduction to Sybase CEP

Provides information on the features and functions of the Sybase CEP Engine.

What is Sybase CEP Engine?

Historically, most businesses were most concerned about the need for accurate storage, retrieval, and analysis of data. Today, businesses are operating in environments where the need to monitor events comes at an ever-accelerating pace. Sybase CEP Engine is a leader in a new generation of tools for processing, analyzing, and managing events in these highly dynamic environments.

Traditional Systems

Event-driven systems have typically been built around either relational databases or real-time messaging systems, or a combination of both.

While these technologies have their advantages, neither is particularly well suited for managing and analyzing events in rapidly changing environments:

- Relational database servers can process large amounts of stored data and can analyze the information with relative ease but are not designed to operate in real-time environments, and do not provide an effective way to monitor rapidly changing data.
- Messaging systems permit data to be monitored in real time but are not generally capable of complex computations, correlations, pattern matching or references to historical data.

For these reasons, custom applications must often be combined with these technologies to create a viable solution. The use of custom applications to compensate for the limitations of these technologies creates new complications:

- Custom applications become increasingly complex very quickly as an organization's need for progressively more sophisticated analysis grows.
- Custom applications are also costly to modify, and they do not scale well as organizational needs change.

Introducing Sybase CEP Engine

Sybase CEP Engine is a powerful new tool in the field of Complex Event Processing (CEP) and Event Stream Processing (ESP) applications. Sybase CEP Engine can be used to build applications for processing and analyzing large volumes of fast-moving data in highly dynamic environments.

Sybase CEP couples the best features of relational databases and messaging technologies, combined with many new features required for the rapidly changing world faced by today's businesses.

Like a real-time messaging system, Sybase CEP reacts to each new piece of data immediately. Like a relational database system, Sybase CEP supports the use of a query language similar to SQL to process and analyze incoming data, and to correlate it with historical data. Sybase CEP also provides sophisticated pattern-matching capabilities without requiring large amounts of custom application code.

Features of Sybase CEP Engine

A description of features of Sybase CEP Engine.

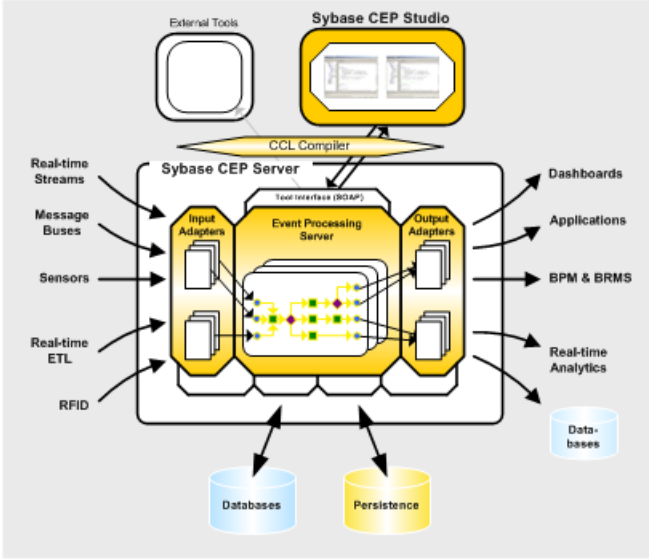
- A relational model for accessing real-time data streams.
- Queries that run continuously, not just when a query is submitted.
- A rich language, based on SQL, but adapted for processing real-time event data, and extended to deal with the temporal characteristics of messages.
- The ability to analyze data and event patterns across a wide variety of data sources to locate pertinent information.
- Extensive capabilities for integrating data from external relational databases.
- The ability to easily extend the language by building and registering custom scalar and aggregator functions.
- Adapters that can be used to integrate Sybase CEP Engine with a variety of external systems.
- SDKs for a variety of languages (Java 1.5, C, C++, Perl, Python, .NET3, and so on) that permit customized interfaces and adapters to be easily implemented.
- A modern development environment for developing, debugging, monitoring and managing queries.
- Optimized performance that can accommodate hundreds of thousands of messages per second per server, with a processing latency measured in milliseconds.
- Guaranteed delivery, high availability, security, and other important enterprise requirements.

Sybase CEP Components

A description of the major components of the Sybase CEP Engine: the Continuous Computational Langauge, Sybase CEP Server, Sybase CEP Data Model, Input and Output Adapters, and Sybase CEP Studio.

- **Continuous Computation Language (CCL):** CCL is an SQL-like language used for writing continuous queries, which manipulate Sybase CEP data. CCL is converted into an executable form by the CCL Compiler.
- **Sybase CEP Server:** Sybase CEP Server processes streams of incoming data against registered continuous queries written in CCL.
- **Sybase CEP Data Model:** The Sybase CEP data model consists of streams and windows providing a foundation on which CCL queries can process data.

- **Input and Output Adapters:** Adapters translate data from a wide variety of external sources into a format compatible with Sybase CEP's data model, and also translate Sybase CEP data back into formats that can be sent to external destinations.
- **Sybase CEP Studio:** Sybase CEP Studio is a graphical user interface, which lets users easily edit CCL queries, view streams and windows, control and monitor Sybase CEP servers, and debug Sybase CEP applications in an interactive development environment.



The Continuous Computation Language

Sybase CEP Continuous Computation Language (CCL) constructs the queries that process data received by the Sybase CEP Server.

Most CCL queries take data from one or more continuous data streams, analyze or manipulate the data, and then output the results to a destination, which is usually another continuous data stream.

CCL is based on SQL, which is the industry-standard query language for use with relational databases. SQL users will find many similarities between CCL and SQL. Like SQL, CCL supports sophisticated data selection and calculation capabilities, including such features as data grouping, aggregations and joins. However, CCL has been extended to include features that are required to support the manipulation of data during real-time continuous processing, such as windows on data streams, pattern and event matching, and output timing controls.

The key distinguishing feature of CCL is its ability to continuously process dynamic data. An SQL query typically executes only once each time it is submitted to a database server and must be resubmitted every time a user or an application needs to re-execute the query. By contrast, a CCL query is continuous, that is, once it is submitted to the Sybase CEP Server, it is registered for continuous execution and stays active indefinitely, executing each time data arrives from one of its data sources.

CCL queries are converted to an executable form by the CCL Compiler. Compilation is typically performed from the Sybase CEP Studio, but it can also be performed by directly invoking the CCL Compiler from the command line or through an API call using one of the Sybase CEP SDKs.

Sybase CEP Server

Sybase CEP Server executes continuous CCL queries to process incoming data streams.

Sybase CEP Server is designed to process hundreds of thousands of messages per second, with latency measured in milliseconds. Sybase CEP Server may run on a single processor, or may consist of multiple Servers (a Server Cluster) distributed across multiple machines, each containing one or more processors.

Sybase CEP Servers can be configured to perform distributed query processing on one or more processors, parallel query processing (clustering) and automatic failover (High Availability). These advanced concepts are discussed in later chapters in this manual and in the *Sybase CEP Installation Guide*.

Sybase CEP Data Model

Just as tables and views provide the foundation for database processing in SQL, the Sybase CEP data model consists of streams and windows which provide a foundation for building continuous queries in CCL.

Streams

Data streams are the basic components for transmitting data within Sybase CEP. CCL queries take data from streams arriving in the Sybase CEP Server, process the data according to the CCL queries, and insert the results into other streams.

SQL users may find it convenient to think of data streams as being similar to database tables. Like tables, data in streams are formatted in rows and columns. Each data stream has an associated schema, which defines the stream's columns and each column's datatype. Whenever a data stream receives data, it appears in the form of a row, with a value for each of the columns represented in the schema.

Just like SQL, the value in a column may be NULL, meaning that the value in that column is unknown or nonexistent.

Note: SQL table schemas usually must specify whether columns in a table may store NULL data. This is not the case in CCL. Every column in every Sybase CEP data stream may contain NULLs.

StockTrades

timestamp	07:15:13	07:15:09	07:15:06	07:15:05	07:15:03	07:15:02	07:15:01
tradeid	5006	5005	5004	5003	5002	5001	5000
symbol	ORCL	AAPL	AAPL	MSFT	MSFT	ORCL	MSFT
volume	500	800	500	1000	1000	1500	500
price	$20.00	$82.00	$81.00	$31.00	$28.00	$18.00	$29.00

Despite these similarities to SQL tables, there are also important differences. SQL tables are finite-sized sets of rows, while data streams are stateless streams of data. Typically, a row that passes through a data stream in a query is available to be processed only at the precise moment in time when it appears in the stream. Once an incoming row has been processed, it is normally no longer available to a query. However, CCL does have several ways of either temporarily or permanently maintaining the state of rows that have already flowed through a stream.

Illustrating Data Streams in the Programmer's Guide

A number of figures in this manual illustrate the way data flows through a CCL query. It is extremely difficult to graphically visualize this process (which, by nature, involves data changing as time passes) using a static picture.

Figures that depict stream data flowing through queries show the input streams above or to the left of the query and the output streams below or to the right, with the rows in the streams shown flowing from the left side of the page to the right. The data in each row is arranged vertically, with the timestamp at the top.

The earliest rows (those with the oldest timestamps) that passed through the query will be on the right of the streams and the most recent rows (those with the newest timestamps) will be on the left. The same data rows will be shown in both input and output streams and timestamps will be shown for all rows to illustrate when each row arrived in the stream.

Sybase CEP timestamps normally include both a date and time resolved to the microsecond ("2006/11/01 09:45:13.000099"). To save space in the illustrations, all timestamps will be shown without the date and with time resolution expressed only to the second ("09:45:13"). Rows that arrive second by second provide sufficient granularity for the examples in this manual. Please remember that Sybase CEP is capable of processing data at much higher rates (many thousands of rows per second). When rows are processed at that speed, timestamps need to be stored and viewed at the most precise resolution.

Windows

A window is a collection of rows, not unlike a database table.

While the "one-row-at-a-time" data stream model is very useful for picking out rows based on selection criteria and for performing basic intra-row calculations, it does not accommodate complex analysis of data involving multiple rows in a stream. In practice, it is often necessary for queries to deal with more than just the current row in the data stream.

For example, the average value of the stock trades that occurred in the last hour in a stock data trading data stream can only be calculated effectively if values from all of the rows for the last hour are available. This requires maintaining the state of more than one previously processed row. A principal way Sybase CEP maintains state is through the concept of windows.

A window can retain the state of many rows from a data stream (or from more than one data stream), enabling CCL to operate on all of these rows at the same time. Windows are filled with the incoming rows from a data stream, and rows are kept based on a variety of criteria:

- Time intervals ("keep one hour's worth of data").
- Row counts ("keep the last 1000 rows").
- Value groups ("keep the last row for each stock symbol").
- Rankings ("keep the 10 largest stock trades by volume").

Illustrating Windows in the Programmer's Guide

Windows are illustrated differently from streams. Where streams are depicted flowing from left to right on the page, with each row arranged vertically, windows will be shown much like a table of rows and columns. Data in the windows will be depicted as entering at the bottom of the illustration, so that the oldest rows in the table will be shown at the top. The columns of data in each row are presented left to right, with the timestamp in the first column on the left. Where rows have expired from a window, they will be shown grayed out and with a strike-through of all data values. This indicates that the row was once in the window but is no longer present because it has expired or has been deleted.

StockTradesMicrosoft Window		
timestamp	**shares**	**price**
~~07:15:01~~	~~500~~	~~$29.00~~
~~07:15:03~~	~~1000~~	~~$28.00~~
07:15:05	1000	$31.00
07:15:16	1500	$28.00
07:15:21	500	$29.00

A window can be used as either a data source or destination for CCL queries. The state of a window is constantly changing. New rows from data streams flow into their associated windows and expire from the windows regularly, based on the window policies. With each change to the window, all continuous queries that access that window are executed and the results are passed along to subsequent streams and queries. These changes can then cause other queries to execute, in an ongoing and continuous process.

Input and Output Adapters

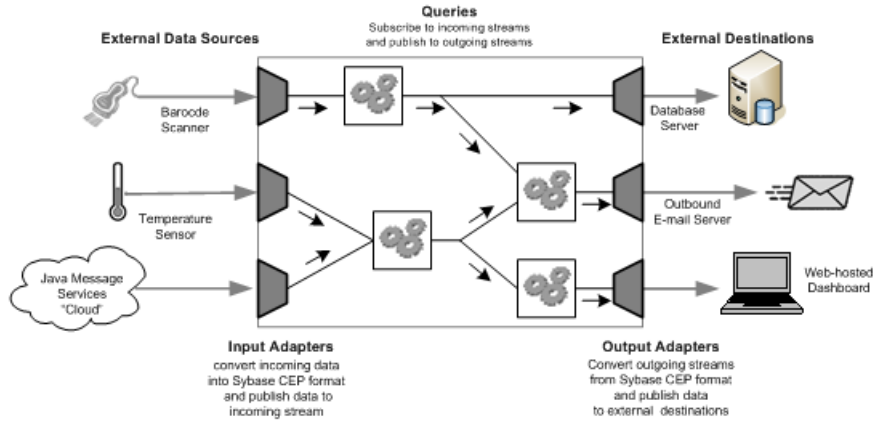
Input and output adapters enable Sybase CEP to send and receive messages from dynamic and static external sources and destinations.

External sources or destinations can include:

- Data feeds.
- Sensor devices.
- Messaging systems.
- Radio frequency identification (RFID) readers.
- E-mail servers.
- Relational databases.

Input adapters are used to translate incoming messages from external sources into a format that can be processed by the Sybase CEP Engine. Output adapters translate processed Sybase CEP rows into a message format compatible with external destinations.

The following illustration shows a series of input adapters which translates messages from a temperature sensor, bar code scanner and a Java Message Service (JMS) "cloud" into formats compatible with Sybase CEP. After the data is processed by several Sybase CEP queries, output adapters convert the Sybase CEP result rows into updates to a database server, e-mail messages and data for a web services dashboard.



Standard Adapters

Sybase CEP standard adapters include the Read From CSV/XML/Regex Socket or File, Write To CSV/XML Socket or File, and JMS adapters.

- Read From CSV/XML/Regex Socket or File: These adapters provide an input interface to continuous data sources (via a socket) in comma-separated value (CSV) format, XML format or through the use of a regular expression to filter the incoming data. For testing purposes, there are also versions of these adapters that read from files.
- Write To CSV/XML Socket or File: These adapters provide an output interface for data generated in the Sybase CEP Engine in CSV or XML format. Data can be written to a socket (for connection to other processes) or to files (for archival, testing, or reporting purposes).

- JMS Adapter: The JMS adapter provides an interface for data being read from or written to a Java Message Service.
- Other Adapters: There are a number of specialized adapters that can be used to read or write data in special formats or from special interfaces. These include:
  - An Email output adapter which can send email containing a text message and the formatted contents of the data from a stream row.
  - A Ganglia input adapter.
  - ATOM/RSS feed input adapters.
  - An IBM WebSphere MQ adapter.
  - Tibco Rendezvous input and output adapters.

Custom Adapters

In addition to the adapters provided by Sybase CEP, you can write your own adapters to integrate into the Sybase CEP Engine. You can design adapters to handle a wide variety of external requirements that cannot be handled by the standard adapters.

A variety of SDKs is provided so that adapters can be written in a number of programming languages, including:

- C.
- C++.
- Java 1.5.
- .NET3 (C#, Visual Basic, and so on).
- Python.
- Perl.

Adapters may be in-process or out-of-process. In-process adapters are executed within the Sybase CEP server process. The Engine starts and stops in-process adapters in conjunction with the execution of a query module. Out-of-process adapters are executed in a separate process (on the same or different machines). Out-of-process adapters are started and stopped independently of the Engine and associated query modules.

For more detailed information about how to create custom adapters, please see the *Sybase CEP Integration Guide*.

Sybase CEP Studio and Other Tools

Sybase CEP Studio is a visual development tool that enables users to interact with Sybase CEP Engine.

Sybase CEP Studio allows you to:

- Develop and store CCL queries:
  - Enter and edit CCL query text.
  - Define data streams.
  - Describe schemas that set the format for data streams and windows.

    - Connect adapters to streams.
    - Set a variety of adapter, stream and application parameters.
- Start, stop and manage queries.
- View stream and window contents or running queries.
- Monitor query execution and performance.
- Debug applications.

While Sybase CEP Studio is the most commonly used way of creating Sybase CEP applications and interfacing with Sybase CEP Engine, Sybase CEP also provides a number of command line, programmatic (SDK), and SOAP interfaces to the functions of Sybase CEP Engine. It is possible to use these other tools to compile, register, load, start, and stop the execution of CCL queries.

For more detailed information about how to use Sybase CEP Studio, see the Sybase CEP Studio Guide. For more detailed information about how to use the Sybase CEP command line tools and SDKs, see the Sybase CEP Integration Guide.

Project Portability

A description of best practices for deploying Sybase CEP projects from the development environment to the production environment. Includes instruction on directory layout, bindings, copying files, and recompiling.

You are probably going to develop your Sybase CEP project on a development system, communicating with a development Sybase CEP Server before deploying your project to production systems that communicate with production Sybase CEP Servers. To make the transition from the development environment to the production environment as easy as possible, follow these best practices:

Directory Layout

Use a single top-level directory for all files related to a single project (the project file, query modules, and schemas, as well as data input files). Put the project file (.ccp) at the top level of the project directory. If your project uses adapters that read from files (the CSV, Binary, or XML read-from-file adapters), specify the path to any input file relative to the $Project variable. For more information about specifying input file paths, see "Configuring Sybase CEP Adapters" in the Sybase CEP Integration Guide.

Bindings

When you bind data streams, specify relative URIs rather than absolute URIs. For example, instead of ccl://host:port:/Stream/Default/InstrumentIn/StocksIn, just use /Stream/Default/InstrumentIn/StocksIn. As long as both projects are running within the same Sybase CEP cluster, Sybase CEP Server will locate the correct project and data stream. For more

information about bindings, see "Creating Data Stream Bindings" in the Sybase CEP Studio Guide.

Copying Files

When you are ready to deploy your project, copy your project files to the new server, placing all your .ccl files together in same place (duplicating the directory structure from your development system is probably the easiest solution).

If the input files for your adapters are in a different location on the new server, be sure to change the associated path property to match.

Recompiling

You can compile your project locally and run it on a remote server using the command c8_client --cmd=compile-and-run. For more information about this command, see "Compiling and Running a Project" in the Sybase CEP Integration Guide.

The StockTrades Data Stream

The StockTrades data stream is used in many of the examples provided by Sybase CEP. The information here is provided for your reference in order to give you a more complete understanding of the examples.

- *The Problem Domain* describes the sector within which this data stream is designed to operate.
- *Data Stream and Schema* describes the principal external data stream used by the application and the stream's schema.

The Problem Domain

The *StockTrades* data stream provides only a small number of data columns. However, you can use it to provide answers for many queries with Sybase CEP, including to output the average trading price for each stock during a specific period for each new trade, to raise an alert whenever shares of a certain stock rise above a particular price, and to indicate whenever the number of shares traded for a stock exceeds the average volume of that stock over a specific period.

The *StockTrades* data stream used in this guide is a very simplified example that deals with data commonly found in financial trading. The financial trading industry is a domain with a significant amount of dynamic information. Stock trades flow continuously providing a large volume of detailed transactional data. Brokers and analysts monitor trade data on an ongoing basis, and are often interested in data that arrive within various intervals of time ranging from minutes to more than one day. Although the *StockTrades* data stream provides only a small number of data columns, it can be used to provide answers for many queries with Sybase CEP including:

- Outputting the average trading price for each stock during a specific period for each new trade.
- Raising an alert whenever shares of Microsoft trade above a particular price.
- Indicating whenever the number of shares traded for a stock exceeds the average volume of that stock over a specific period.

Note: This data stream is an extremely simplified example of stock trading data used to illustrate various concepts throughout this guide. However, the use of financial trading data in no way implies that Sybase CEP is only useful within the financial trading industry, or that the Sybase CEP functionality applies only to a specific domain. The Sybase CEP Engine is a flexible and versatile set of tools that are applicable to a wide variety of industries and problem domains. The purpose of this guide is to teach you how to use the tools, which you may then apply to your particular needs.

Data Stream and Schema

The *StockTrades* data stream is used to simulate how Sybase CEP Engine receives messages from an external data source, such as a stock exchange ticker.

Since this is a simulation, the *StockTrades* data stream does not actually receive "live" messages from a stock exchange. Instead, there is a data file containing a set of sample trade data that is used to simulate the actual use of a live data feed.

The stream includes columns representing a trade identifier, stock symbol, number of shares traded, and the price per share. The stream schema that defines these columns for the *StockTrades* stream is:

Column	Datatype	Description
tradeid	Integer	A unique sequence number that identifies each trade.
symbol	String	The stock symbol associated with this trade (such as MSFT, DELL, or ORCL).
volume	Integer	The number of shares exchanged for this trade.
price	Float	The price of each share exchanged for this trade.

Here is a small sample of input data from the *StockTrades* stream which is used in most of the examples in this Guide:

timestamp	tradeid	symbol	volume	price
07:15:01	5001	MSFT	500	29.00
07:15:02	5002	ORCL	1500	18.00

timestamp	tradeid	symbol	volume	price
07:15:03	5003	MSFT	1000	28.00
07:15:05	5004	MSFT	1000	31.00
07:15:06	5005	AAPL	500	81.00
07:15:09	5006	AAPL	800	82.00
07:15:13	5007	ORCL	500	20.00
07:15:15	5008	AAPL	2000	83.00
07:15:16	5009	MSFT	1500	28.00
07:15:18	5010	AAPL	1000	83.00
07:15:21	5011	MSFT	500	29.00

Other Data Streams

Data streams can provide additional input to the Sybase CEP application and output streams which connect to external adapters.

In addition to this basic data stream, there are a number of other data streams that will be used in conjunction with the examples throughout this guide. Like *StockTrades*, some of these streams are data sources which provide additional input to the Sybase CEP application. Others are output streams which are the ultimate destination of information from a query. Some of these output streams would normally be connected to external adapters (such as an e-mail adapter or a JMS message queue), while others would be used solely inside the Sybase CEP engine to send data along to other Sybase CEP queries.

When these additional streams are first used in the guide, the associated schemas may be presented either in tabular or textual form.

Using CCL

A guide to using Continuous Computation Language to manage and analyze data within the Sybase CEP Engine.

More About Sybase CEP Components

A closer look at Sybase CEP components including time, data streams, windows, schemas, datatypes, adapters and query modules. You will need a basic understanding of these components to begin working with CCL queries.

How Streams Work in Sybase CEP

Data streams are the fundamental way in which data is transmitted throughout Sybase CEP. CCL queries take data from data streams, process that data, and send it to other streams. While data streams are similar to database tables, database tables are comparatively static while the data flow in a stream is ongoing and continuous.

Data streams may be accessed by more than one query. A data stream may be used as a data source for multiple queries, and it may also be used simultaneously as a data source for some queries and a destination for other queries.

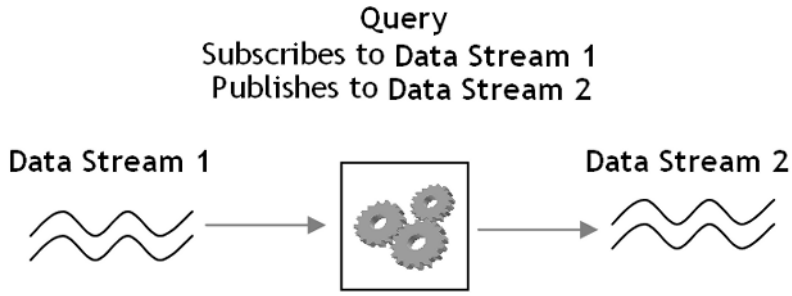
Data streams alone do not retain any state. Therefore, only the most recent row in a data stream is "visible" to the queries with which it is interacting, and that row is only visible at the moment when it appears in the stream. If an arriving row is not either consumed immediately by the query or retained in some way, it cannot be accessed later. Fortunately, Sybase CEP has several ways of retaining the state of rows that have entered via a stream, which are explained later in this chapter.

Publishing and Subscribing to Streams

Data streams and CCL queries work together to move and manipulate the data within Sybase CEP Engine.

When a query takes data from a data stream, the query is said to subscribe to that data stream. When a query places data into a stream, it is said to publish data to that stream.

The following figure shows the flow of data from one stream to another through a query:



Every Sybase CEP data stream must be defined before it can be used by a query. A query that refers to data streams that have not been defined will not execute. A stream definition typically consists of a stream name and a schema definition for the stream.

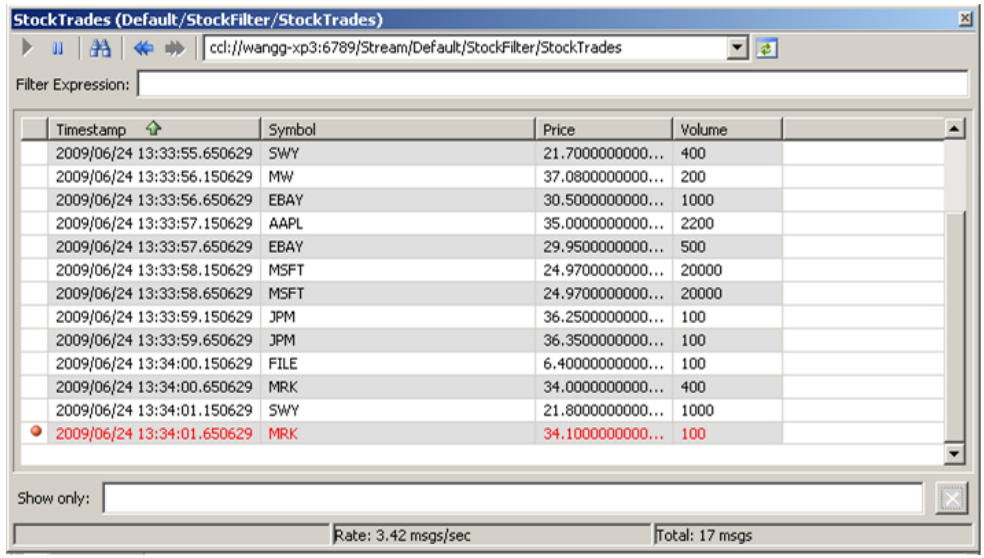
Note: If you are using Sybase CEP Studio to build applications, refer to the Sybase CEP Studio Guide for instructions on how to create data streams in Sybase CEP Studio.

Time in Data Streams

Time is an essential aspect of Sybase CEP. Data rows arrive continuously, and the CCL queries associated with a data stream execute every time a new row arrives. For queries to run smoothly, Sybase CEP must keep track of the sequence in which rows arrive. Sequencing is done by timestamps that are associated with each row.

Row Timestamps

Every row in Sybase CEP has an associated row timestamp. When a stream or window is viewed in a Sybase CEP Studio viewer, this timestamp value appears as the left-most column of each data row.



Even though the row timestamp looks like a column in the Studio viewer, it is not an actual column in the row and does not appear in the schema. The row timestamp may not be explicitly modified by a CCL query. However, queries may be written in such a way as to take action based on row timestamp values. In addition, the row timestamp values of output rows may be implicitly controlled by the results of certain kinds of CCL queries (specifically joins and output-regulated queries).

Note: In addition to the row timestamp, which is required by Sybase CEP Engine, other timestamps may also be included in a row as actual data columns. These timestamps may be used in any expression in a CCL query, and may even be explicitly modified as the row moves through a sequence of queries. These timestamps are not used in any way by Sybase CEP to control the sequencing of events.

Setting Row Timestamp Values

A description of how to set row timestamp values with Sybase CEP Engine and adapters.

For streams that receive data from external sources (in other words, via adapters), the row timestamp can be set in several different ways:

- The adapter may itself set the row timestamp value and send it to the Engine with each row. For example, a stock trade row might include the actual time of the trade taken from the original source of the data as its row timestamp, or the adapter itself may set the row

timestamp to the current system time, so that the row timestamp value reflects when the adapter sends the row to the Engine.

Note: One problem that can occur is latency in delivery from the adapter which causes rows to arrive at the Engine late or out of sequence. Sybase CEP has advanced capabilities for handling these situations when data rates are high or when network latency impacts the ability to deliver rows from the adapter to the Server or to the adapter from the data source at precise times.

- The row timestamp for a row from an external source may also be set by the Sybase CEP Engine itself. This results in the row timestamp reflecting when the row arrives in the Engine, rather than an external time value. When this approach is used, the row timestamps chosen by the Engine will meet all of Sybase CEP's sequencing and timing requirements, but it may not be possible to synchronize timing in the Engine with the real events that are represented by the data.

For streams that do not receive external data, the row timestamp is always set internally by the Sybase CEP Engine.

Timestamp Values

All timestamp values in Sybase CEP, including row timestamps, are measured and stored internally in absolute microseconds.

These timestamps are converted and interpreted as actual dates and times relative to a standard base (also called "the epoch") which, in the Sybase CEP Engine, is January 1, 1970. A timestamp of 0 represents 12:00 midnight UTC (Coordinated Universal Time, also widely known as Greenwich Mean Time or GMT) on the epoch date.

Microsecond timestamp values may be positive or negative. Positive timestamps reflect times after the epoch, while negative timestamp values reflect times prior to the epoch. Thus, a timestamp value of 100 represents 100 microseconds past midnight on January 1, 1970, while a timestamp value of -60,000,000 (60 seconds in microseconds) represents 1 minute prior to midnight on December 31, 1969.

Sybase CEP does not interpret or infer the actual meaning of row timestamps, and uses them solely to synchronize all timed events within the system. These timestamps may designate a variety of events, depending on their usage in the application. Proper use of row timestamps in queries is dependent upon understanding the origin and meaning of a timestamp.

For instance, if the row timestamp is being set by Sybase CEP Engine, queries that use the row timestamp need to be aware that the timestamp doesn't necessarily indicate when the event represented by the data row actually took place. The actual time the event took place could differ from the system-generated row timestamp by several seconds, or even minutes or hours. Row timestamps supplied by the Sybase CEP Engine always indicate only the exact time when the row entered Sybase CEP Engine, or when it was created by a query.

Sybase CEP can also use adapter-supplied row timestamp values "as is". It is then the adapter's responsibility to ensure that the assigned row timestamp value has the appropriate meaning for the application, depending upon the actual data source.

Rows With Duplicate Timestamp Values

Sybase CEP uniquely distinguishes every row, even where multiple rows have the same row timestamp value.

Every row arriving in the Sybase CEP Engine is processed as if it arrived at a unique moment in time, even if its row timestamp appears to indicate that it arrived at the same time as other rows. The sequence in which the rows are processed is determined by the order in which the rows actually arrive in the Sybase CEP Engine. This applies to rows that arrive in the same data stream, as well as rows that arrive in different data streams of the same application.

Communicating with External Sources

External data enters Sybase CEP Engine through input adapters, which translates data from an external format into a format usable by Sybase CEP Engine.

Adapters may be in-process or out-of-process. In-process adapters are executed within the Sybase CEP server process while out-of-process adapters execute in their own processes, possibly even on different machines than the Sybase CEP Engine, and communicate with the Sybase CEP server via interprocess communication mechanisms (pipes, sockets or streams).

Every external data source that provides data to Sybase CEP must be connected to its own input adapter, which must in turn be connected to a Sybase CEP data stream.

Adapters connected to data streams are also used to send Sybase CEP data to external destinations. A CCL query places its results in a data stream, which may be connected to an output adapter. The output adapter translates the data into the required external format and sends it to the external destination.

Since Sybase CEP receives data from many types of sources, Sybase CEP Engine includes an extensive adapter architecture to translate data to and from external formats. Some of these adapters ship with Sybase CEP. Others can be created using a variety of adapter Software Development Kits (SDKs) that are available for different programming languages, such as C, C++, Java 1.5 and .NET3.

For a complete discussion of the available adapters, SDKs, and their usages, please see the *Sybase CEP Integration Guide*.

How Windows Work in Sybase CEP

A description of how CCL windows maintain row states in Sybase CEP Engine.

A CCL query can only process a row in a data stream once it arrives in the Sybase CEP Engine. A query that subscribes to a data stream has access only to the most recent arrival in the stream. However, CCL queries frequently require that the state of previously-received rows be maintained, in order to perform analysis or computations on multiple rows. For example, a query that calculates the moving average of the number of shares of stock traded over the previous hour, requires that the data for all trades (or at least the volume data) for the past hour be retained in some way to calculate the average. In such cases, CCL windows are used to

maintain the state of previously arrived rows. CCL queries subscribing to these windows may then access all the rows in the window.

Since data streams and windows can both act as data sources and destinations for queries, the terms data source and destination are often used in Sybase CEP documentation to refer to either a stream or a window. Thus, the basic relationship between queries and data streams can be expanded to include data streams or windows.

Window Policies

A description of windows in Sybase CEP and their policies.

The window definition states the policy that determines what rows in a stream should be retained and for how long. CCL windows come in several varieties:

- *Count-based windows* have a single count-based policy and retain rows up to a specified maximum number. Alternately, count-based policies may be created to retain a maximum of a specified number of *row groups*, called *buckets*. For more information about buckets, please see the KEEP Clause section of the Sybase CEP *CCL Reference*.
- *Time-based windows* have a single time-based policy and retain rows for a specified interval of time.
- *Multi-policy windows* have one count-based policy and one time-based policy, and retain rows according to both sets of criteria.

Both count-based and time-based windows may be sliding or jumping (and multi-policy windows may be both).

- *Sliding windows* accumulate and expire rows, (or groups of rows called buckets), one at a time. Count-based sliding windows remove rows or buckets from the window as new rows arrive or new buckets are created; time-based sliding windows remove rows from the window as time passes and rows age and expire.
- *Jumping windows* accumulate and expire rows or buckets either up to a specified number (in the case of count-based windows) or for a specified time interval (for time-based windows) then remove all the rows or buckets from the window at once and start accumulating again from an empty window.

Count-based windows may also be grouped. These windows may be thought of as implementing a separate count-based policy *for every combination of unique values* in one or more key columns. For example, instead of simply retaining the last five rows to arrive in the stream, a grouped window analyzing stock trades might retain the last five rows for each stock symbol.

Both count-based and time-based windows may be ordered. *Ordered windows* order the rows retained by the window based on the values of one or more columns.

Important: As further described in *Windows and Row Propagation* on page 23, window policies retain row state, but do not affect how quickly rows are propagated to other data streams or windows.

Named and Unnamed Windows

A description of named and unnamed windows and their behaviour in Sybase CEP.

Unnamed windows are created within a query and associated with one data stream. This type of window uses the same schema as its stream, and receives the same rows as the data stream (although the window saves the state of the rows while the stream merely lets the rows pass through). Once a window is defined in a query, it replaces its related stream as the data source for the query. A window defined within a query can only be used by that query.

Named windows are defined independently of the query, in a separate CCL statement. Named windows have their own schema, and rows can be inserted into a named window from more than one query. Named windows may also be accessed by more than one query.

Introducing Schemas

All data streams and windows have schemas that specify their structure. A schema specifies the number and names of the columns in the stream or window and indicates each column's datatype. Schemas may be created explicitly by the user and associated with a particular stream or window, or, in some cases, a schema for an output stream may be inferred by Sybase CEP from the contents of the query.

CCL column names

Every column in a schema must have a column name.

Column names are constructed according to the following rules:

- Column names have no maximum length.
- Column names must begin with an alphabetic character: A-Z or a-z, or \177 to \377 in octal representation, or 7F-FF in hex representation.
- Column names must be made up entirely of alphabetic characters, numeric characters (0-9) and/or underscores (_).
- Column names are not case sensitive (in other words, avgprice and AVGPrice are the same name).
- Within a schema, no two columns can have the same name.
- Column names cannot be a CCL reserved word.
- The prefix C8_ is reserved for Sybase CEP names.
- CCL contains other words that are not reserved but that still have special meanings. In particular, it is not recommended to use the names of CCL functions as column names.

Datatypes

Each column in a schema must have a datatype. The datatype defines the kind of value that each column may contain.

The following table lists the datatypes available in Sybase CEP.

Datatype	Description
Boolean	A value of TRUE or FALSE.
Integer	A numeric value representing a signed 32-bit binary integer. Values range from -2,147,483,648 to 2,147,483,647.
Long	A numeric value representing a signed 64-bit binary integer. Values range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
Float	A numeric value representing a signed 64-bit IEEE floating-point number. Values range approximately between $-10^{+/-308}$ to $+10^{+/-308}$.
String	A character string value representing a variable-length string. Maximum string length is system-dependent, but is always at least 65,535 characters (and may be much greater).
Timestamp	A 64-bit time value representing a date (year, month, day) and time (hours, minutes, seconds and microseconds). This datatype is useful for storing precise time values. Row timestamps are always implicitly assigned the TIMESTAMP datatype.
Interval	A time value representing a 64-bit integer that contains a number of microseconds. This number is typically used to store the difference in time between two TIMESTAMPs.
XML	A data item representing an XML tree or forest. An XML column value must contain correctly constructed XML, but is not verified against any specific XML DTD or XML Schema.
Blob	A Binary Large Object representing a long unconstrained binary string of data, which may include, in addition to regular string values, ASCII NULLs, control characters and other binary values.

For a full listing of available datatypes and their limitations and requirements, please see the Sybase CEP CCL Reference.

Datatype Conversions

All columns handled by Sybase CEP have a datatype. Use conversion functions to convert data to the correct datatype.

Different datatypes support different operations and are treated differently by Sybase CEP. For example, some functions, such as AVG (average) or SUM (total) can be performed on numeric columns (such as INTEGER or FLOAT) but not on STRING columns.

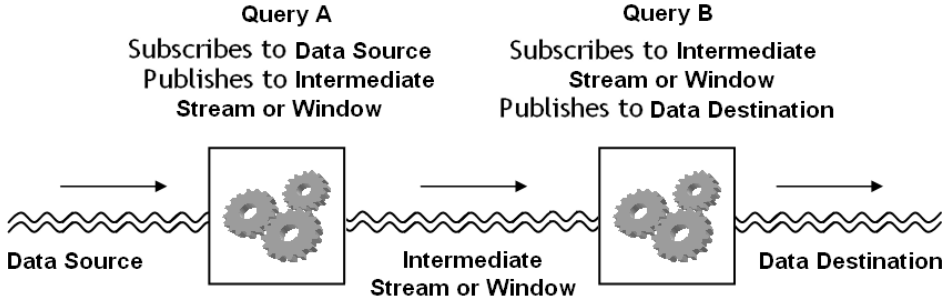
Conversion functions can be used to convert data to the correct datatype. When data of one datatype is placed in a column of another datatype without an explicit conversion function, Syabse CEP attempts to convert the incoming data to the datatype of the receiving column automatically. For example, values of type INTEGER can be automatically converted to type LONG. If Sybase CEP cannot convert the incoming data to the column's datatype, it reports an error.

Each value subsequently placed in a column assumes the datatype of that column. For example, if you insert the value "1994-01-01" into a TIMESTAMP column, Sybase CEP treats the 1994-01-01 character string as a TIMESTAMP value, after verifying that it translates to a valid date.

Query Paths and Query Pipelining

Inside Sybase CEP Engine, streams, windows, and queries can combine to route data along a single path, separate it along multiple paths, or merge it from multiple paths into one. Multiple CCL queries can be pipelined together, in which each successive query subscribes to the data stream published by the previous query.

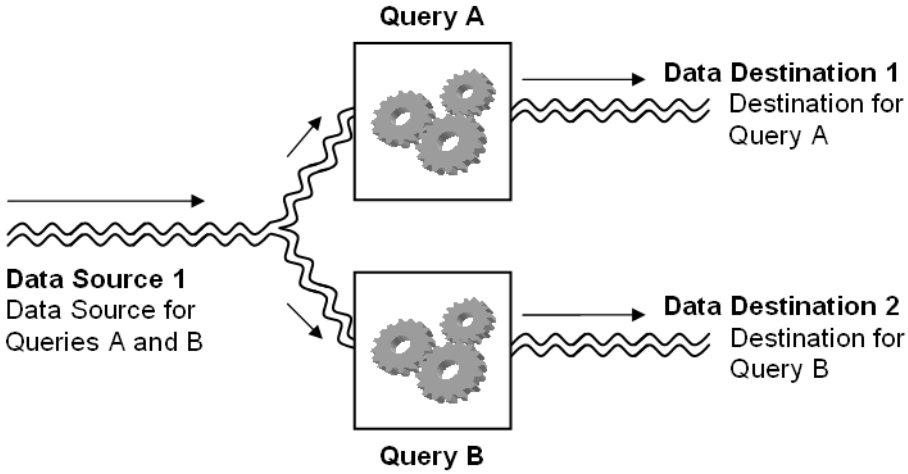
The following figure shows a sequential pipeline of two queries:



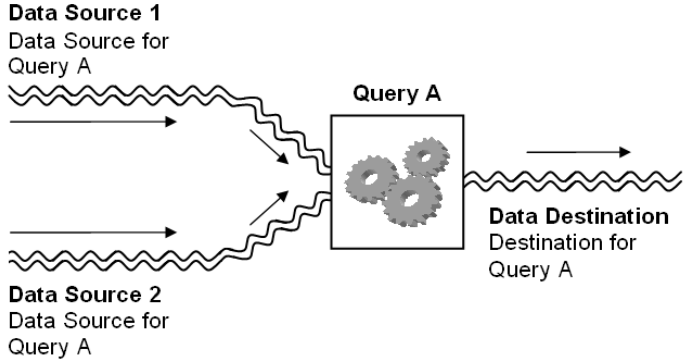
Windows and Row Propagation

In cases where a CCL window is used as a data source and destination in a query chain, the window policy has no effect on how quickly rows are propagated down the chain. Windows retain the state of rows, but do not actually "hold on" to the rows themselves. Regardless of the window policy, rows travel down the chain as quickly as the system can accommodate.

Several queries can also subscribe to the same data source and then publish their results to different streams, thus creating a branch in the pipeline. Each branch may then go on to be processed independently by other queries:



Finally, a single query can subscribe to data from multiple sources and publish the results to a single stream. This very common occurrence in CCL queries is shown in the following figure:



About Query Modules

CCL queries are stored in query modules. A query module is a reusable group of one or more queries, all of which are compiled and executed together.

To write or edit queries in a query module, you must first either create a new .ccl file, or open an existing file.

Note: None of the queries in a query module run until the module is started. For instructions on starting and stopping query modules, see the Sybase CEP Studio Guide. Once the module is

started, however, the queries in it run simultaneously and continuously, until the module is stopped, or the Sybase CEP Engine is shut down.

Query modules are also used to keep track of data streams. Every data stream that supplies or receives data from a query must be attached to the query module where that query resides. Data streams can be attached to query modules using the Sybase CEP Studio interface.

For instructions on how to add a data stream to a query module, please see the Sybase CEP Studio Guide.

Putting the Basic Elements Together

A description of tasks you must complete before you can successfully execute CCL queries.

This chapter has discussed data streams and their schemas, adapters, windows, and query modules. If you are running Sybase CEP Studio, please refer to the Sybase CEP Studio Guide for instructions on how to complete the following tasks:

1. Create or open a query module.
2. Add uniquely named data streams to the module.
3. Create a schema, or assign an existing schema, for each stream.
4. Add adapters to streams that either subscribe or publish to external sources.
5. Write CCL queries.
6. Run the query module.

CCL Queries

The Sybase CEP Continuous Computation Language (CCL) is a language based on the ANSI SQL Standard (1998). It is used to manipulate, analyze and process data within the Sybase CEP Engine.

Like SQL, CCL syntax is structured around commands called statements. One or more CCL statements reside in a query module. When a query module is started, all the CCL statements in it are executed and run continuously until the module is stopped or the Sybase CEP Engine is shut down.

Note: SQL users should note that CCL is not identical to SQL. Whereas SQL is designed to query data from static tables, CCL is adapted to the needs of processing and analyzing dynamic data. CCL syntax and implementation diverge from the ANSI SQL standard when the requirements of continuous stream and event processing call for such a divergence.

CCL has several types of statements. This chapter introduces you to the simplest forms of the Query Statement. A Query Statement takes data from one or more data sources, processes it and then publishes it to a destination. This is, by far, the most common statement in CCL and can take many complex forms. Other sections of this guide provide more detail on the Query Statement. For complete Query Statement syntax, please see the Sybase CEP CCL Reference.

Writing a CCL Query

Provides an example of a CCL query statement that subscribes to all of the data in the *StockTrades* data stream and publishes it, row by row, to a stream with an identical schema called *StockTradesAll*.

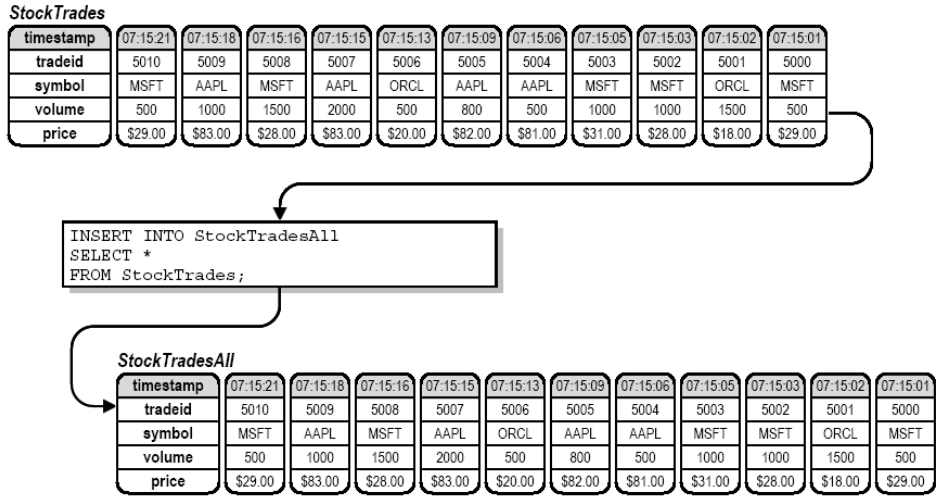
Here is an example of the simplest possible Query Statement.

```
INSERT INTO StockTradesAll
SELECT *
FROM StockTrades;
```

In practical terms, this query is not very useful, since it simply takes data from one data stream and puts it into another stream. However, it illustrates some important CCL syntax and introductory concepts:

- The INSERT INTO clause, which is always the first clause in a Query Statement, indicates the destination of the query output. This query publishes its output to the *StockTradesAll* stream. All Query Statements provide a single destination to which the query results are be published.
- All Query Statements must have a SELECT clause immediately after the INSERT INTO clause. The SELECT clause defines the contents of the query output. This query uses the special asterisk syntax (SELECT *), which indicates that all columns should be selected from the source data stream.
- The FROM clause follows the SELECT clause and specifies one or more data sources to which the query subscribes. In this example, the data source is the *StockTrades* data stream.
- A semicolon (;) is used to terminate every CCL statement.

Here is an illustration of data flowing from the *StockTrades* stream to the *StockTradesAll* stream after passing through this basic query:



Selecting Rows and Columns

One of the most important functions of a CCL query is to select which rows and columns are passed from the source stream to the destination stream. Columns are selected by specifying the required columns in the Select list.

Rows are selected by using one or more filter or selection conditions. A row is selected if all of the filter conditions are true.

Let's look at a slightly more complex query which illustrates column and row selection:

```
INSERT INTO StockTradesMicrosoft
SELECT volume, price
FROM StockTrades
WHERE symbol = 'MSFT';
```

In this example, only the volume and price data is needed, so the destination stream *StockTradesMicrosoft* has the following schema:

Column	Datatype	Description
shares	Integer	The number of shares exchanged for this trade.
price	Float	The price of each share exchanged for this trade.

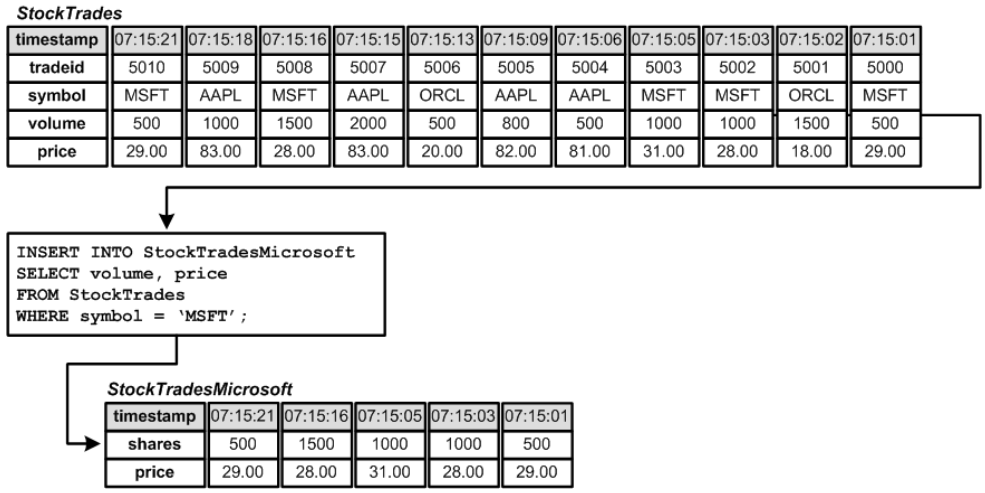
The query selects the volume and price columns (SELECT volume, price) from the *StockTrades* stream for all Microsoft stock trades (WHERE symbol = 'MSFT'). Notice the following syntax in this query:

- Because the *StockTrades* and *StockTradesMicrosoft* streams do not have identical schemas, the query can not simply select all columns in *StockTrades* using the asterisk. In

fact, using the asterisk notation would result in a query compilation error stating that there are more columns in the SELECT list than there are in the destination stream.

- The SELECT clause lists the specific *StockTrades* columns which should be published to the *StockTradesMicrosoft* stream. When there is more than one column in the SELECT clause, the column names are separated by a comma.
- The WHERE clause follows the FROM clause. The WHERE clause specifies a filter condition for the query. The filter condition is an expression which must evaluate to TRUE or FALSE. It restricts the rows from the data source which are published into the destination stream to those rows that meet the expression's condition.
- Even though the volume column in *StockTrades* (volume) doesn't have the same name in *StockTradesMicrosoft* (shares), there is no problem because, by default, columns are copied from source to destination positionally. That is, the first item in the SELECT list publishes to the first column of the destination stream, the second item in the select list publishes to the second column, and so on. This default order can be overridden, as explained in a later example.

Here is an example of what the input and output streams would look like for this query:



CCL Expressions

CCL expressions are combinations of literals, column names, operators, and functions that evaluate to a single value.

Here are some examples of possible expressions:

```
price > 80.00
symbol = 'DELL'
volume * price
SQRT(StockTrades.volume)
(temperature - 32) * 5 / 9
volume IS NOT NULL
```

Literals

Sybase CEP expressions can include literals of various datatypes.

Datatype	Examples
Boolean	TRUE, FALSE
Integer	1, -4455, 1234567899999L (long integer)
Float	1.2345, -999.999, 543.21e+3 (exponential format)
String	'Jane', "Dell Computers", "" (the empty string)
Timestamp	TIMESTAMP '2003-11-30 15:04'
Interval	INTERVAL '3 days 02:30', 3 days 2 hours 30 minutes
Null	NULL

String literals may be enclosed in either single quotes or double quotes. Timestamp literals always consist of the word TIMESTAMP followed by a quoted string which represents the date and optionally the time for the timestamp. Interval literals can be in either a quoted form preceded by the word INTERVAL or in a segmented form without quotes, specifying the DAYS, HOURS, MINUTES and SECONDS for the interval. The null literal represents the absence of a value and is always just the word NULL.

For a complete explanation of literals used with CCL, please see the Sybase CEP CCL Reference.

Column Names

Sybase CEP column names in expressions can be either unqualified, or qualified with a stream name.

An example of an unqualified stream name is `symbol`, where the qualified stream name would be `StockTrades.symbol`. When unqualified column names are used, the usage must be unambiguous within the query.

If two or more streams have the same column name, then the use of that column name in an expression will generally require a qualified name (it is not legal for the same schema to have two columns with the same name). Column names are case-insensitive, so `symbol` and `Symbol` are treated as the same column name in Sybase CEP.

For a complete explanation of how column names are used with CCL, please see the *Sybase CEP CCL Reference*.

Operators

Sybase CEP expressions support a variety of numeric, non-numeric, and logical operator types.

Operator Type	Description
Arithmetic	addition (+), subtraction (-), multiplication (*), division (/), exponentiation (^), modulo (mod)
String	concatenation operator (||)
Comparison	equality (=), inequality (!= or <>), greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), wildcard string comparisons (LIKE, REGEXP_LIKE), null comparisons (IS NULL, IS NOT NULL)
Logical	AND, OR, XOR, NOT
Grouping	(...)

CCL operators are executed following a defined precedence order which is similar to the order used in ANSI SQL (for instance, multiplication and division are always performed before addition and subtraction, and all arithmetic is performed before any comparisons). Parentheses can always be used to group the operations in an expression in a different order (operators inside parentheses are performed before operators outside the parentheses).

For a complete explanation of operators and the precedence order used by CCL, please see Sybase CEP CCL Reference.

Scalar Functions

Sybase CEP supports several categories of scalar functions.

Function Type	Description
Numeric	Numeric functions are used with numeric values (INTEGER, LONG and FLOAT). Some numeric functions can also be used with INTERVAL and TIMESTAMP values. Examples of numeric functions include ROUND (rounds a float value to the nearest integer), SQRT (returns the square root of a value), and MIN (returns the smallest value from a list of values).
String	String functions are used with STRING values and usually (but not always) return a STRING value. Examples of string functions include LEFT (returns the left-most characters of a string), RTRIM (removes trailing spaces from a string), and REPLACE (replaces occurrences of one substring with another substring).

Function Type	Description
Conversion	Conversion functions convert data values of various datatypes to the datatype specified by the function name. For example, the TO_STRING function converts a value to the STRING datatype. Not all conversions between different data types are permitted in Sybase CEP. The rules governing valid datatype conversions are discussed in Sybase CEP CCL Reference.
XML	There are special scalar functions which are designed to correctly handle XML data. For more information on these functions, please see Sybase CEP CCL Reference .
Miscellaneous	There are several miscellaneous functions including the COALESCE function, which returns the first non-null expression in a list of expressions, and the NOW function, which returns a timestamp corresponding to the current time.

Scalar functions take one or more expression values as arguments and return a single result value for each row of data processed by a query. These functions can appear in most expressions, and are used most often in SELECT clauses and WHERE clauses.

For a complete explanation of the scalar functions used by CCL, please see Sybase CEP CCL Reference.

User-Defined Functions

In addition to the built-in functions provided by Sybase CEP, you can define your own functions. Please see the Sybase CEP *CCL Reference Guide* and the Sybase CEP *Integration Guide* for more information about creating and using user-defined functions.

CCL Expressions in the SELECT Clause

The SELECT clause may contain a list of one or more columns, which are published to the destination stream specified in the INSERT INTO clause. Each of the items in the select list may also be an expression.

While select list expressions typically reference column names, they are not limited to column names, and may not contain column names at all.

Select list expressions may include one or more of the expression elements listed in the previous sections. An item in the select list may be a simple literal, column name or a complex expression containing a mix of literals, column names, operators and functions.

Here is an example of an expression used in a select list. This query publishes the total dollar amount for each trade, and the trade identification number associated with the trade, to the *TradeValues* stream:

```
INSERT INTO TradeValues(id, tradevalue)
```

```
SELECT tradeid, volume*price
FROM StockTrades;
```

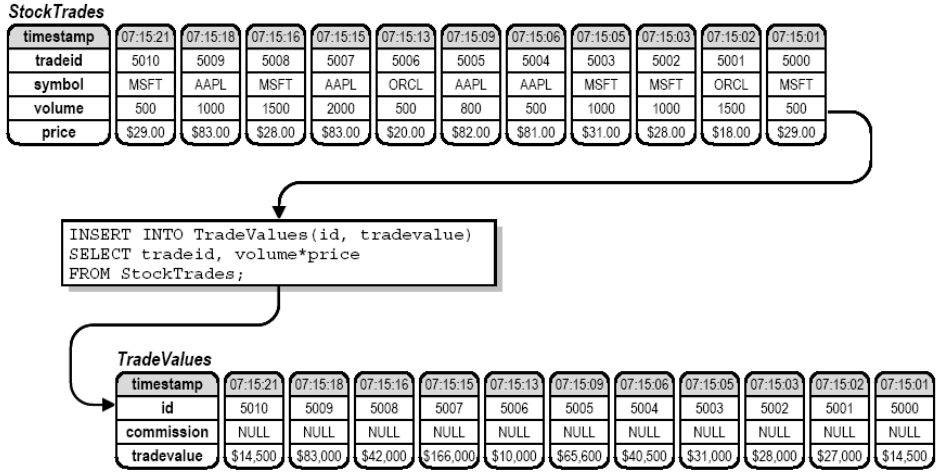
The *TradeValues* stream has the following schema:

Column	Datatype	Description
id	String	A unique sequence number that identifies each trade.
commission	Float	The commission value of this trade.
tradevalue	Float	The total value of this trade (volume * price).

Note the following important points about this query:

- The first item in the select list selects the tradeid column of *StockTrades* and publishes it to the id column of *TradeValues*.
- The second item in the select list of the query is an expression that multiplies the volume and price columns of the *StockTrades* stream to calculate the total dollar amount of the stock trade. This total value is written to the tradevalue column of the *TradeValues* stream.
- The INSERT INTO clause can include a list of columns after the stream name enclosed in parentheses, which explicitly matches the items in the select list with corresponding columns in the *TradeValues* stream. This overrides the default order which publishes items in the select list to the destination stream left to right.
- The *TradeValues* schema has an extra column (commission) which is not referenced in the query. Ignoring for the moment why this column might be present in this schema, note that if there are extra columns in the schema, they do not need to be referenced by the query **as long as the query uses the explicit column list in the INSERT INTO clause.** If there are extra unreferenced columns in the schema (such as commission), the query fills those columns with the NULL value.

Here is an example of what the input and output streams would look like for this query:



CCL Expressions in the WHERE Clause

The WHERE clause uses expressions to restrict the data captured by the SELECT clause.

Consider the following change to the previous query:

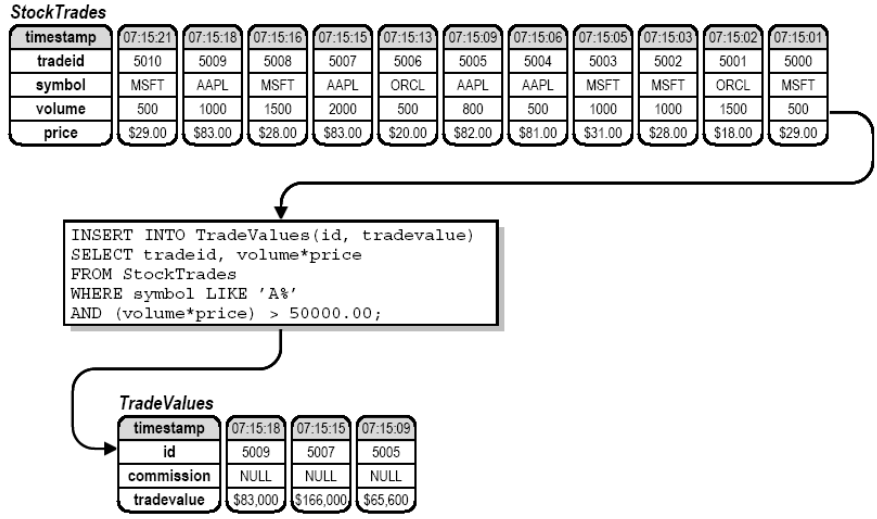
```
INSERT INTO TradeValues(id, tradevalue)
SELECT tradeid, volume*price
FROM StockTrades
WHERE symbol LIKE 'A%' AND (volume*price) > 50000.00
```

The WHERE clause has a complex expression which filters rows for symbols that start with the letter 'A', but only if the total trade value is also greater than $50,000. The LIKE comparison operator is used to support "wildcard" comparisons for strings. The percent sign (%) in the literal 'A%' means to match any character values after the initial letter 'A'. In the small subset of data in the example, this would only find the symbol 'AAPL'.

A WHERE clause expression may refer to columns that are not included in the select list of the SELECT clause as shown in this example (symbol), although the columns must still be present in the data sources listed in the FROM clause.

Also note the use of the parentheses to group the multiplication operation. Although this isn't strictly needed in this example, using parentheses helps to make it clear to anyone reading the query what calculations are actually being executed.

Here is an example of what the input and output streams would look like for this query:



The use of expressions in the WHERE clause is somewhat more restricted than their use in the SELECT clause. The WHERE clause must always evaluate to a value of TRUE or FALSE. This is usually accomplished using comparison and logical operator expressions, although it

is possible for other expressions, even those without a comparison or logical operator, to evaluate to TRUE or FALSE.

Using Windows

This chapter teaches you how to create basic CCL windows and use them in queries. Windows permit you to maintain the state of rows that have been previously processed from a stream.

Using Windows in a FROM Clause

CCL windows are typically defined by the use of a KEEP clause. This section explains how to define basic windows by adding a KEEP clause to the FROM clause of your query. See the *Sybase CEP CCL Reference* for complete KEEP clause syntax.

Defining Count-Based Windows

Count-based windows have a KEEP clause that defines the maximum number of rows the window keeps, and whether or not these rows or buckets expire from the window one at a time, or all in a group.

Sliding Count-Based Windows

A sliding count-based window retains a fixed number of rows or buckets.

Once the window is full, each new row that arrives or each new bucket that is created in the window displaces the oldest row or bucket, which is then removed from the window.

The basic KEEP clause syntax for a sliding count-based window is:

```
KEEP number ROWS
```

where `number` specifies the number of rows the window should keep.

Earlier, there was an example that created the StockTradesMicrosoft stream which consisted of all stock trades for Microsoft. Here is an example of a sliding count-based window that is used to calculate the average price of the preceding three Microsoft trades on a continual basis using the StockTradesMicrosoft stream as a data source:

```
INSERT INTO StockTradesMicrosoft
SELECT volume, price
FROM StockTrades
WHERE symbol = 'MSFT';

INSERT INTO AvgPriceMicrosoft
SELECT AVG(price)
FROM StockTradesMicrosoft KEEP 3 ROWS;
```

This example illustrates the use of two queries in a pipeline. The output stream from the first query (StockTradesMicrosoft) is subscribed to by the second query. Each time a new row is

added to the window in the second query, the oldest row in that window is also removed, and the second query is executed which calculates the average price for all of the rows in the window.

The output of the second query (in this case, just the expression AVG(price)), which calculates the average price of the most recent three Microsoft trades, is sent to the AvgPriceMicrosoft stream:

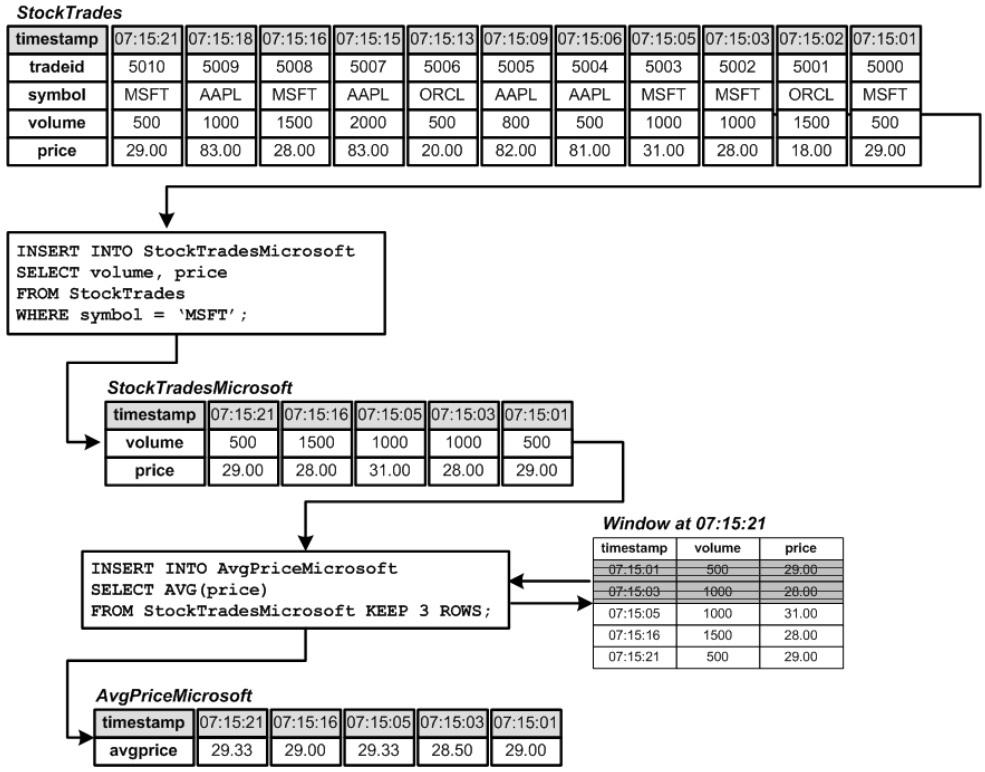
Column	Datatype	Description
avgprice	Float	The average price for the most recent set of trades.

The AVG function in the second query is not a scalar function. It is an aggregator function which operates on more than one row at a time. The AVG function takes the sum of the values for the specified column, and divides the sum by the number of rows that are in the group of rows, which in this case is the same as the number of rows in the window.

Note: If a row has a value of NULL for the column being aggregated, the value is not included in either the sum or the count of the number of rows. This is different from a value of 0, which would be added to the sum (although having no effect on the total) and would be included in the count, thereby reducing the aggregated value of the average. Thus, values 0, 2 and 4 would yield an average of 2 ((0+2+4)/3 = 6/3 = 2) but value of NULL, 2 and 4 would yield an average of 3 ((2+4)/2 = 6/2 = 3).

In addition to AVG, there are several other aggregator functions (such as SUM, MIN, MAX, COUNT), which, like AVG, are almost always used in conjunction with data in windows.

Here is an example of how the data flows from StockTrades through the first query and then into the second query, creating the window which generates the ultimate average price information. Keep in mind that the average price is constantly recalculated every time a new row enters the window from the StockTradesMicrosoft stream:



Jumping Count-Based Windows

A jumping count-based window adds rows or buckets to a window until it is full and then expires all of the old rows at the same time upon the arrival of the next row, or creation of the next bucket.

The basic syntax for a jumping count-based window is:

```
KEEP EVERY integer ROWS
```

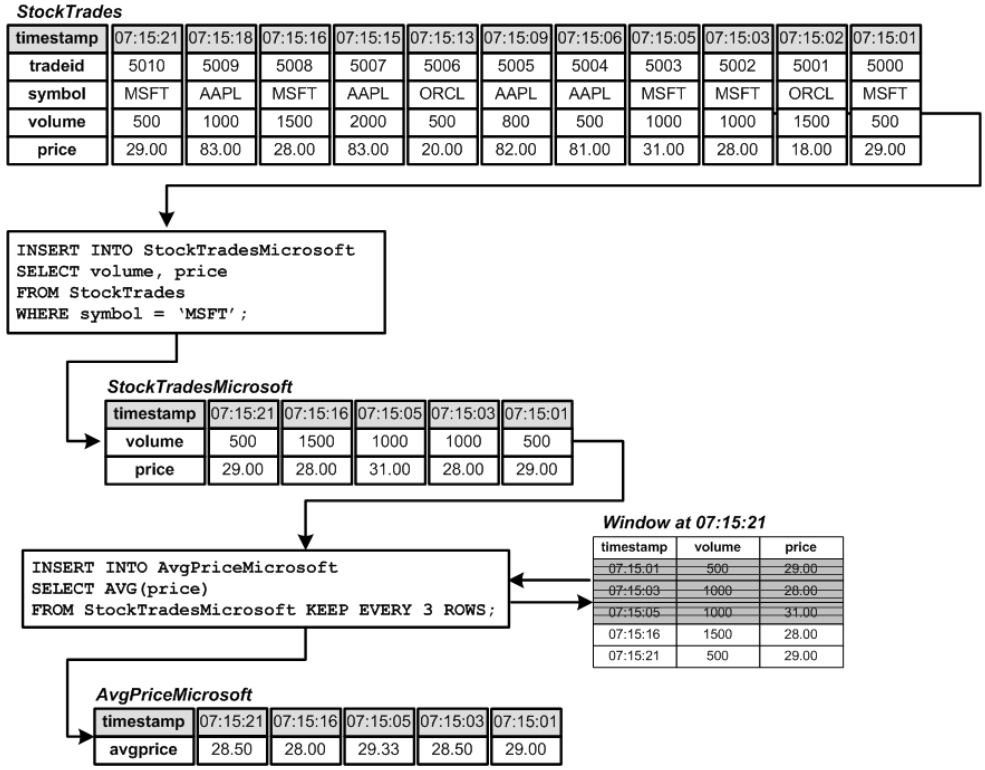
Consider the difference in the output if the previous example were written with a jumping count-based window like this:

```
INSERT INTO StockTradesMicrosoft
SELECT volume, price
FROM StockTrades
WHERE symbol = 'MSFT';

INSERT INTO AvgPriceMicrosoft
SELECT AVG(price)
FROM StockTradesMicrosoft KEEP EVERY 3 ROWS;
```

In this example, a jumping window of three rows would be created and after the first three rows were stored in the window, resulting in three output values for the average price being published, all the rows would expire and the next three rows would create a new set of data in the window and a new set of average price results, completely independent of the values from the previous three rows.

Contrast the output of this jumping count-based window query with the output from the sliding count-based window query:



Other Count-Based Windows

A description of the KEEP LAST ROW and KEEP ALL count-based windows.

A KEEP LAST ROW window keeps only the most recent row from the stream. It is functionally identical to a KEEP 1 ROW window. This becomes very important when two or more data sources are joined together in a single query as you will see in a later example.

A KEEP ALL window keeps every row that arrives. Rows never expire. This is useful for certain kinds of historical calculations.

Note: 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. Use this type of window only when you have allocated sufficient resources for it or when you know that the window can't grow too large. Alternatively, you may use a named window with KEEP ALL and provide a way of explicitly deleting unneeded rows as necessary.

Defining Time-Based Windows

Time-based windows have a KEEP clause that defines how long the window will keep rows, and whether or not these rows will expire from the window one at a time, or all in a group.

Sliding Time-Based Windows

A sliding time-based window retains a variable number of rows, depending on how many rows have arrived within the specified interval.

As time passes, rows that have been in the window longer than the specified interval expire from the window. Expiration of rows happens regardless of whether new rows arrive in the window or not.

Note: This is significantly different from count-based windows. In count-based windows, rows only expire when new rows arrive.

Because rows can arrive and expire at different times, the output of a time-based window query will often differ dramatically from a count-based window.

The basic KEEP clause syntax for a sliding time-based window is:

```
KEEP interval-literal
```

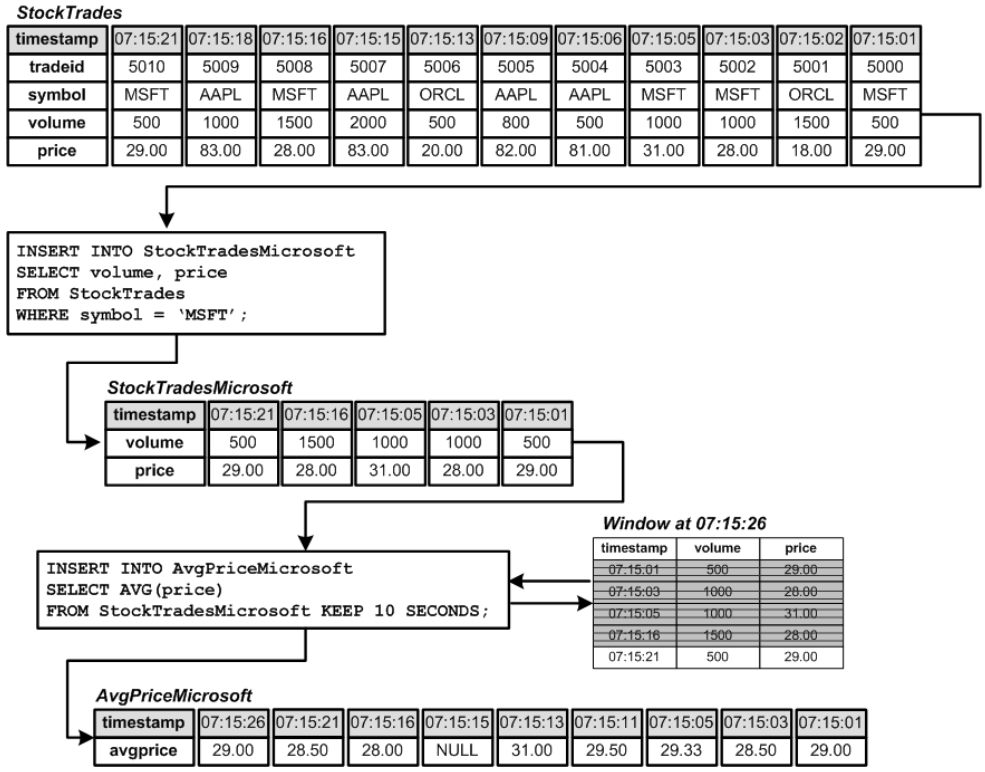
where `interval-literal` specifies the length of time a row will stay in the window before it expires.

Let's rewrite the sliding count-based window example used in the previous section. Instead of retaining 5 rows in the window, an interval of 10 seconds is used:

```
INSERT INTO StockTradesMicrosoft
SELECT volume, price FROM StockTrades
WHERE symbol = 'MSFT';

INSERT INTO AvgPriceMicrosoft
SELECT AVG(price)
FROM StockTradesMicrosoft KEEP 10 SECONDS;
```

In this example, a new average price is calculated every time a new row enters the window, just as before. However, additional average prices will be calculated each time a row expires from the window as well, because whenever a row expires, the average price for the preceding 10 seconds will change:



In this example (unlike in count-based examples), more rows come out of the query than go in. This is caused by the changing value of the average price each time a row expires in the window. These expirations occur at 07:15:11, 07:15:13, 07:15:15, and 07:15:26.

Specifically notice that when the row which entered at 07:15:05 expires at 07:15:15, the output row for that timestamp shows a value of NULL, because at the moment, there are no other rows in the window to produce an average value.

Jumping Time-Based Windows

A jumping time-based window adds rows to a window for a fixed interval and then expires all of the old rows at the same time when the end of the interval is reached. Provides the syntax for a jumping time-based window.

The syntax for a jumping time-based window is:

```
KEEP EVERY interval-literal
```

Consider the difference in the output if the previous example were written with a jumping time-based window like this:

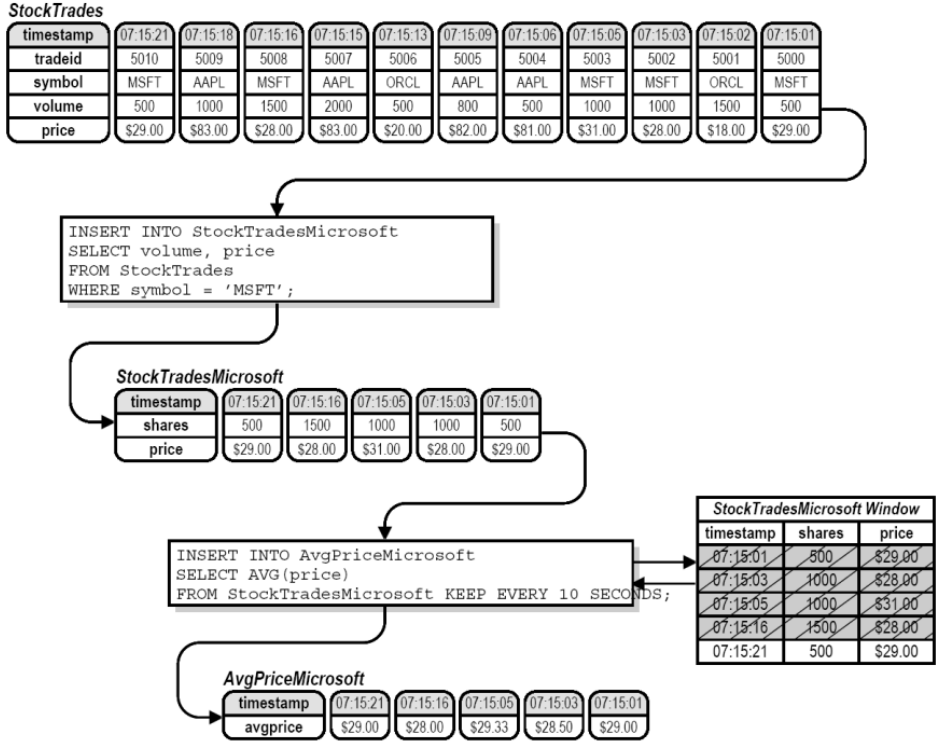
```
INSERT INTO StockTradesMicrosoft
```

```
SELECT volume, price
FROM StockTrades
WHERE symbol = 'MSFT';

INSERT INTO AvgPriceMicrosoft
SELECT AVG(price)
FROM StockTradesMicrosoft KEEP EVERY 10 seconds;
```

In this example, a jumping window of 10 seconds would be created and after the first 10 seconds of data was stored in the window, resulting in output values for each new row, all of the rows would expire and the next interval would begin, completely independent of the values from the previous interval.

Contrast the output of this jumping time-based window query with the output from the sliding time-based window query:



Note that unlike the sliding time-based window, no row is produced when the interval expires. In sliding time-based windows, it is important to generate these rows because the output values change at indeterminate moments in time based on when each individual row expires. Since the intervals expire at predetermined points, and since the output of each interval expiration would always be the same when the window is empty, there is no compelling reason to create an additional row with the average price set to NULL when the interval expires.

Alternate Intervals for Jumping Time-Based Windows

Describes how to offset the beginning of the interval for a time-based window to another time than the default.

By default, a jumping time-based window begins its intervals offset from January 1, 1970 at midnight. In the previous example, this means that the window intervals always begin and end at 0:00, 0:10, 0:20, and so on. If you want to offset the beginning of the interval for your window from the default (for instance, so that the intervals can begin and end at 0:05, 0:15, and so on), add the STARTING AT subclause to the window definition, using the following syntax:

```
KEEP EVERY interval-literal STARTING AT interval-literal
```

Here is the previous example again, shown with the additional STARTING AT subclause, which causes the window intervals to be offset by 5 seconds:

```
INSERT INTO StockTradesMicrosoft
SELECT volume, price
FROM StockTrades
WHERE symbol = 'MSFT';

INSERT INTO AvgPriceMicrosoft
SELECT AVG(price)
FROM StockTradesMicrosoft
KEEP EVERY 10 seconds
STARTING AT 5 seconds;
```

Defining Named Windows

In addition to creating unnamed windows inside a FROM clause, you can also create reusable window definitions that can be shared by multiple queries.

These window definitions are called *named windows* and are defined with the CCL Create Window Statement. The following table illustrates the main differences between an unnamed and named window:

Unnamed Window	Named Window
Is defined inside a Query Statement.	Is defined in a separate Create Window statement.
Does not have a name.	Has a name, which must be unique to the query module.
May be used only by the statement where it is defined.	May be used by several other types of CCL statements, including any Query Statements in the query module.

Unnamed Window	Named Window
Has one or more window policies (KEEP clauses), defined inside the Query Statement's FROM clause. The policies are linked to a data stream that serves as one of the query's data sources and apply only to that data stream.	Has one or more window policies (KEEP clauses), defined by the Create Window statement. The policies are not permanently linked to a particular data stream and are applied to whatever data enters the window.
Inherits a schema from the data stream to which it is linked.	Includes a SCHEMA clause that provides a schema definition for the window.
Inherits data from the data stream to which it is linked.	Does not contain any data when it is first created. Named windows are populated by being used as destinations by other CCL queries.

Here is an example of a previous example rewritten to incorporate named windows.

```
CREATE WINDOW WindowTradesMicrosoft
SCHEMA 'WindowTradesMSFT.ccs'
KEEP 3 ROWS;

INSERT INTO WindowTradesMicrosoft
SELECT volume, price
FROM StockTrades
WHERE symbol = 'MSFT';

INSERT INTO AvgPriceMicrosoft
SELECT AVG(price)
FROM WindowTradesMicrosoft;
```

The first statement in this example creates a named window called WindowTradesMicrosoft. The schema for this window is contained inside the WindowTradesMSFT.ccs schema file and the window's policy keeps up to three most-recent rows that enter the window. The second query supplies the window with data and the third uses it as its data source. Neither of the subsequent queries determines the window's policy or its schema.

For more information about the Create Window statement and its clauses, please see the Sybase CEP *CCL Reference*.

Grouping a Window

Describes how Sybase CEP partitions windows into one or more more column values.

All of the windows explained so far consist of a single group of rows from the stream. However, Sybase CEP has a way of taking a window and partitioning it by one or more column values. For instance, if the original StockTrades stream is used as the input to a window, and a KEEP 2 ROWS policy is specified, only the most recent 2 rows for the entire stream will be kept in the window. An alternative requirement might be to keep the most recent two rows for each company (symbol) whose stock is traded. This can be done by using the GROUP BY clause with a windowed query:

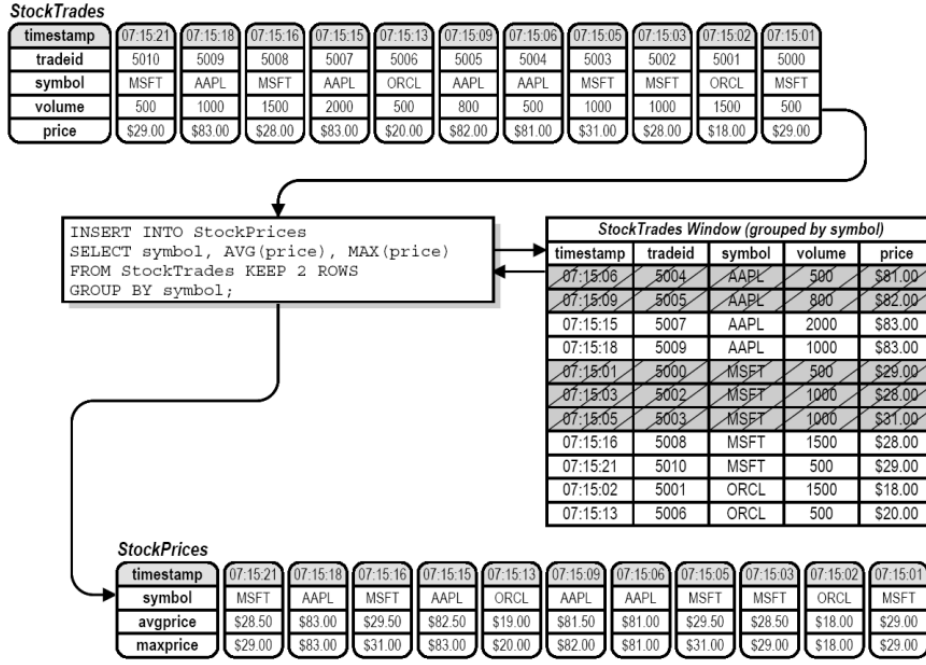
```
INSERT INTO StockPrices
SELECT symbol, AVG(price), MAX(price)
FROM StockTrades
KEEP 2 ROWS
GROUP BY symbol;
```

Here is the schema for the stream *StockPrices*:

Column	Datatype	Description
symbol	String	The symbol for this set of trades.
avgprice	Float	The average price for the most recent set of trades.
maxprice	Float	The maximum price for the most recent set of trades.

In this query, the GROUP BY clause creates a two-row subwindow for each symbol. Each subwindow is managed independently and output is created for each new row that arrives for each symbol. The average price and maximum price that is calculated is for the current and previous rows for that symbol.

Here is an illustration of the window and stream data for this query:



The window illustration shows the rows grouped (and sorted) by their symbol. Notice that rows expire in the window based on their symbol. Only two rows (the most recent row and the previous row) for each symbol are kept in the window.

Joining Data Sources

A query may use data from multiple data sources, though it must always publish data to a single data source. Using multiple data sources in a single query is handled by an operation called a join.

In this section, the following topics will be covered:

- Stream/Window Joins.
- Using Stream Aliases.
- Window/Window Joins.
- Joins with Subqueries.
- Self-Joins.

Joining a Stream to a Window

A description of how to create an inquiry stream which receives a row whenever an inquiry is required. The row contained in this stream can be joined to a window which holds the desired data.

Here is an example of a simple join query that publishes the most recent trade for a stock whenever an inquiry is made for that stock symbol:

```
INSERT INTO TradeResults
SELECT StockTrades.*
FROM TradeInquiry, StockTrades
KEEP LAST WHERE StockTrades.symbol = TradeInquiry.symbol
GROUP BY StockTrades.symbol;
```

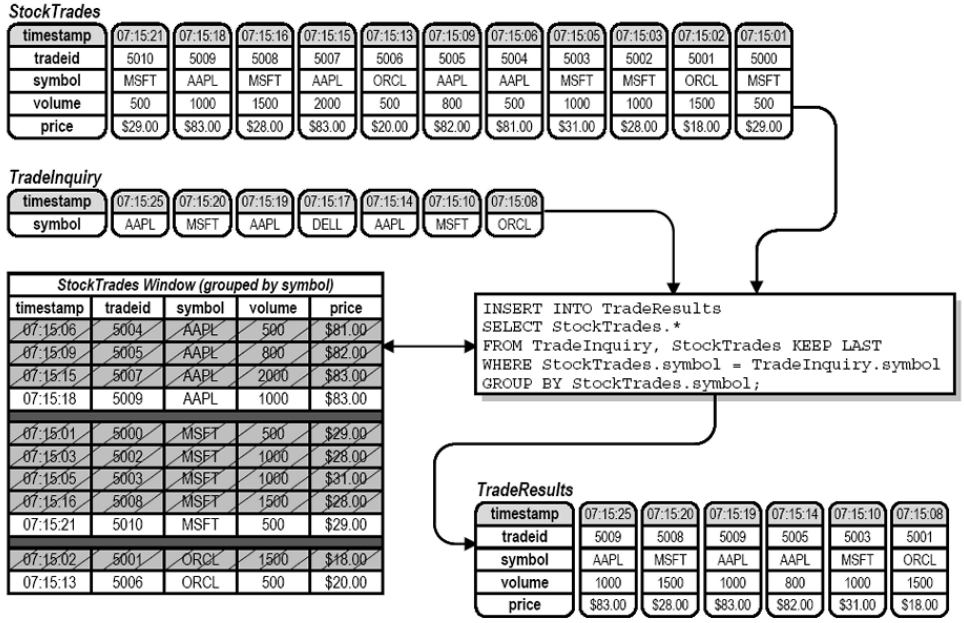
The *TradeInquiry* stream is a single column stream that receives a row with a stock symbol whenever someone needs to view the latest trade for that symbol. It can be thought of as a simple stream coming from a terminal, web page or other application. Here is the schema for the *TradeInquiry* stream:

Column	Datatype	Description
symbol	String	The symbol of the requested stock.

The *StockTrades* stream is directed into a KEEP LAST window which, because of the GROUP BY clause, keeps the last row for each symbol. (Without the GROUP BY clause, only the most recent trade for the entire *StockTrades* stream would be kept.)

The output stream schema for *TradeResults* is the same as the schema for *StockTrades*. The use of the qualified term StockTrades.* puts all of the columns from *StockTrades*, but not the symbol column from *TradeInquiry*, into the output stream,

Here is an illustration of how this simple join query works:



Here are some important things to notice about this simple join query:

- The FROM clause of the query lists the data sources used for the join. Multiple data sources in a join are separated by commas in the FROM clause.
- When two or more data sources are used in a join query, only one of the sources can be a stream. The other sources must be windows.

  Note: This is a very important point to understand. A query can only process the most recent row from a stream, and only at precisely the moment when the row enters the stream. Since rows from different streams will almost never "exist" in a query at the same moment in time, it is virtually impossible to join two streams. Only one of the streams will have data at any given moment, so there will be nothing to join to in the other stream. Fortunately, a window can easily be used to maintain the state of the data from a stream so that when it needs to be joined to another stream, there will be relevant data in both the stream and the window to accomplish the join operation.
- The rows in the stream and the window need to be matched up using a join condition. This is provided in the WHERE clause (WHERE StockTrades.symbol =

TradeInquiry.symbol). The join condition in this query makes sure that the incoming row in *TradeInquiry* is matched only to rows in the *StockTrades* window with the same symbol.

- Notice that at 07:15:19, the row with tradeid 5009 is inserted into the *TradeResults* stream. At 07:15:20, the row with tradeid 5008 is inserted into the *TradeResults* stream, even though it appeared in the *StockTrades* stream prior to tradeid 5009. The effects of the window can result in an output stream which has rows in a completely different sequence than the rows in any of the data sources.
- Also notice that at 07:15:25, the row with tradeid 5009 is repeated in the *TradeResults* stream. Since no other trade for AAPL had occurred when the next inquiry for that stock arrived, the query simply publishes the same row it published for the previous AAPL inquiry at 07:15:19.
- Because both the *TradeInquiry* and *StockTrades* schemas have a column called symbol, it is incorrect to use the column name symbol by itself. The CCL compiler would not know which column named symbol in the query to use. Instead, the query must use qualified names TradeInquiry.symbol and StockTrades.symbol. Preceding the column name with the name of the stream or window (separated from the column name by a period) makes the reference to the different symbol columns unambiguous.
- There can be only one row timestamp for any row. Since a join merges two or more rows into a single row, the row timestamp associated with the output row of a stream/window join is taken from the row in the stream.

Using Stream Aliases

When using qualified column names in any query, the names may become very long. Use the AS keyword in the FROM clause to assign shorter aliases to data source names.

To assign shorter aliases to data source names, using the syntax FROM data-source AS alias. For example, the *StockTrades* and *TradeInquiry* data streams in the above example, may be given the alias ST and SI, respectively. The same query would then look like this:

```
INSERT INTO TradeResults
SELECT S.*
FROM TradeInquiry AS T, StockTrades AS S
KEEP LAST
WHERE T.symbol = S.symbol
GROUP BY S.symbol;
```

How Rows are Joined to a Window

When a data stream is joined to a window, rows are published by the query only when the data stream receives data. When rows arrive in the window, the rows are immediately placed in the window but no output rows are produced. When a row arrives in the data stream, it is conceptually joined to all the rows in the window.

The join produces a Cartesian product. The resulting set of rows is then processed against the conditions of the WHERE clause, and all rows that do not match the selection criteria are filtered out.

Join behavior is identical when a data stream is joined to several windows. Rows arriving in the various windows are added to the respective windows but no output is published. The join query executes only when a row arrives in the stream and produces a Cartesian product of the stream row with all rows in every window.

Note: If the windows are very large and depending upon the selection criteria used to join the data sources, a stream to multiple window join may take considerable execution time to execute for each stream row. Sybase CEP tries to optimize this process by automatically generating indexes for the rows in the windows, and using specialized matching algorithms to minimize the time it takes to correlate the rows.

Aggregators in a Stream/Window Join

An example of a join query that publishes the symbol and the volume weighted average price (VWAP) for a stock over a limited period. Describes the *InquiryPrice* stream and schema.

```
INSERT INTO InquiryPrice
SELECT StockInquiry.symbol, SUM(volume*price)/SUM(volume)
FROM StockInquiry, StockTrades
KEEP 10 SECONDS
WHERE StockTrades.symbol = StockInquiry.symbol
GROUP BY StockTrades.symbol;
```

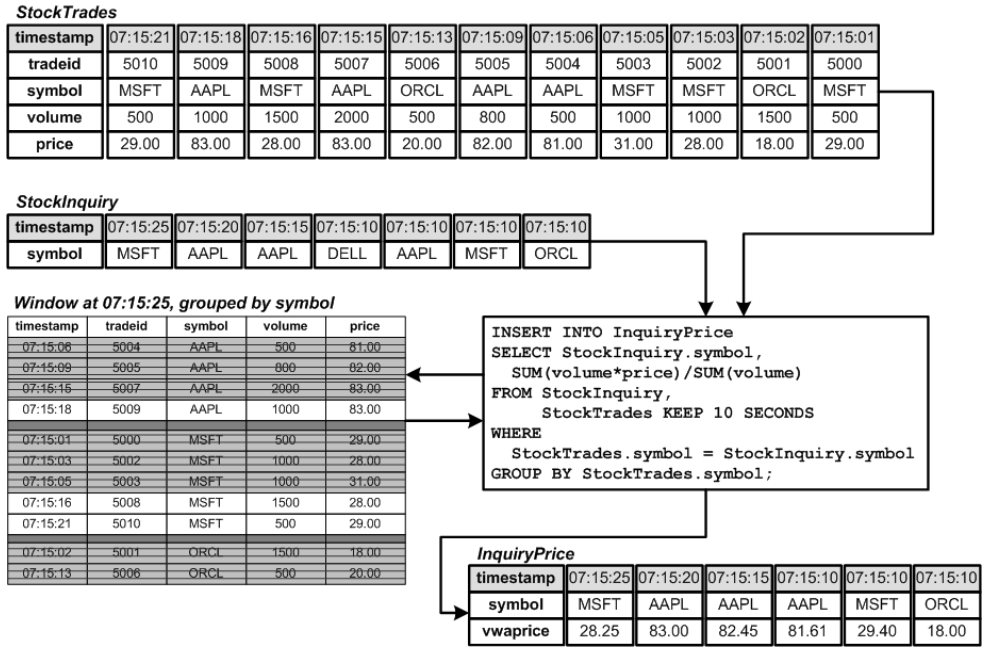
The output of the query goes to the *InquiryPrice* stream which has the following schema:

Column	Datatype	Description
symbol	String	The symbol for this set of trades.
vwaprice	Float	The volume-weighted average price for the most recent set of trades.

In this example, the stock trades are accumulated in a sliding time-based window for ten seconds and grouped by symbol using the GROUP BY clause.

Note: Obviously, ten seconds is an artificially small window period which is chosen to match the small amount of data in the illustrative data stream. Typically, the window period will be much greater for most kinds of data, and can be any reasonable value, from a few minutes to many hours or even days, depending on the volume of data that will be retained and the available memory in your system.

Here is an illustration of how this join query works:



Joining Multiple Windows

Joins can also be performed with two or more windows. Unlike stream to window joins, window-to-window joins produce data any time a row enters any of the windows.

Assume that you wanted to compare the average total sale (volume times price) for Oracle with the average sale for Microsoft over the past ten trades every time a trade occurs for either stock.

Here is one way to do this:

```
INSERT INTO OrclAvgSale
SELECT AVG(volume*price)
FROM StockTrades
KEEP 10 ROWS
WHERE symbol = 'ORCL';

INSERT INTO MsftAvgSale
SELECT AVG(volume*price)
FROM StockTrades
KEEP 10 ROWS
WHERE symbol = 'MSFT';

INSERT INTO AvgSaleComparison
SELECT ORCL.avgsale, MSFT.avgsale
FROM
  OrclAvgSale AS ORCL KEEP LAST,
  MsftAvgSale AS MSFT KEEP LAST;
```

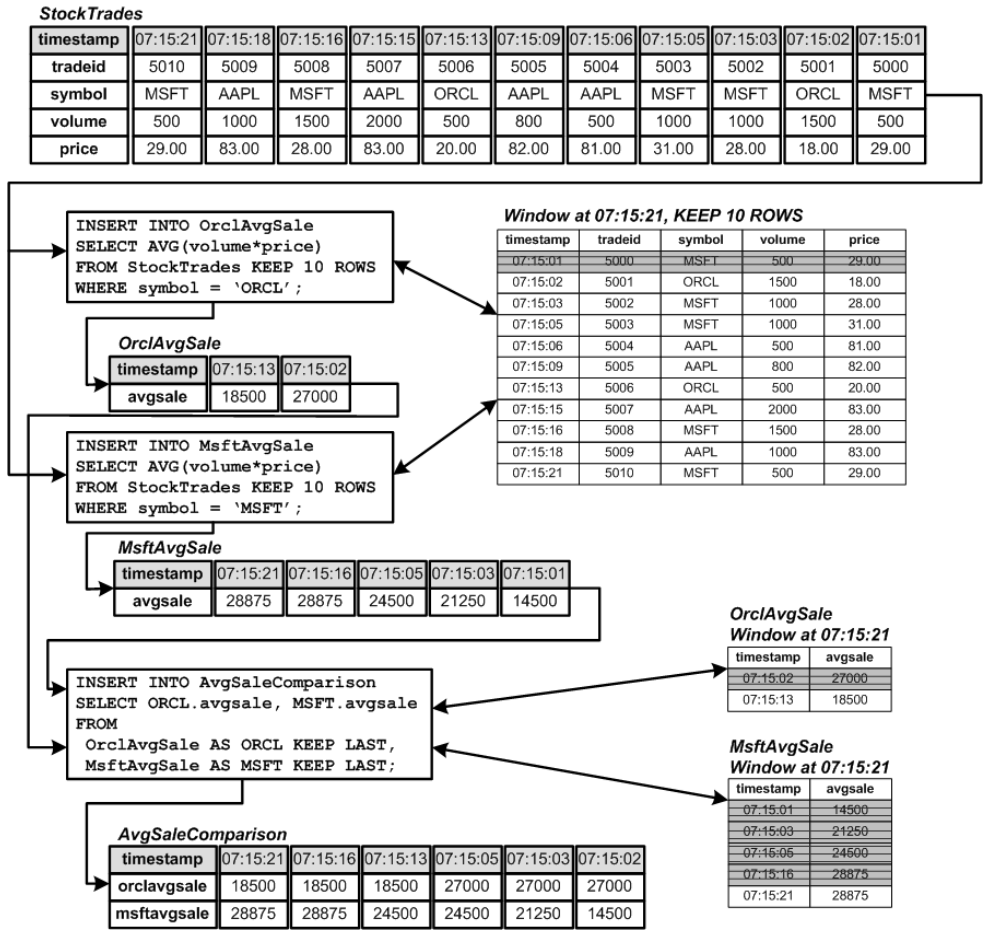
The first query calculates the average sale amount over the previous 10 rows for Oracle stock and publishes the results to the OrclAvgSale stream, while the second query does exactly the same thing for all Microsoft stock and publishes the results to the MsftAvgSale stream. Both of these streams have an identical schema which contains the average sale value:

Column	Datatype	Description
avgsale	Float	The average sale over the preceding period.

The third query joins the contents of MsftAvgSale and OrclAvgSale and publishes the results to the combined AvgSaleComparison stream, which has the following schema:

Column	Datatype	Description
orclavgsale	Float	The average sale for Oracle.
msftavgsale	Float	The average sale for Microsoft.

This query publishes a row whenever a row arrives either from MsftAvgSale, or from OrclAvgSale. Here is an illustration of how this join query works:



Notice that the StockTrades window in the first two queries are illustrated as being shared. If two or more windows in a project are based on the same stream and have identical retention policies, Sybase CEP does not need to store two copies of the window. Sybase CEP Engine optimizes this situation by sharing the common window data between two or more queries.

Notice that the join in this query doesn't have a join condition (in fact, it doesn't have a WHERE clause at all). Whenever there is no WHERE clause with a join condition, the result of executing the query will be a Cartesian product of the two data sources.

However, also note that in this query, both data sources are KEEP LAST windows. This means that at any moment in time, the windows for each of the data sources will only have the most recent average sale value for the particular stock (a KEEP LAST window is also the same as a KEEP 1 ROW window, which means to keep only the most recent row that has arrived).

The Cartesian product of two one-row windows is a single row so there is no need for a join condition at all in this example.

Joining Data Sources Using Subqueries

Use subqueries to feed data to a subsequent query.

Here is how the previous example could be restructured using subqueries:

```
INSERT INTO AvgSaleComparison
SELECT ORCL.avgsale, MSFT.avgsale
FROM
(SELECT AVG(volume*price) as avgsale
 FROM StockTrades KEEP 10 ROWS
 WHERE symbol = 'ORCL') AS ORCL KEEP LAST,
(SELECT AVG(volume*price) as avgsale
 FROM StockTrades KEEP 10 ROWS
 WHERE symbol = 'MSFT') AS MSFT KEEP LAST;
```

Subquery expressions, enclosed in parentheses, take the place of stream names in the FROM clause and are essentially the same as any other query except:

- No INSERT INTO clause is used before the SELECT clause.
- If any columns in the SELECT list are expressions, an AS clause is needed for each column in order to provide a name which can be used in the outer query.

If the data from the subquery is going to be used in a window, the window definition must be placed outside the subquery as shown above. Notice that subqueries themselves can have windows but the output of a subquery is always a stream unless it is converted to a window by a KEEP clause in the outer query.

The query above is functionally identical to the previous example, except that it is accomplished in a single query with two subqueries instead of using three separate queries and two additional data streams. In actuality, Sybase CEP constructs the same intermediate data streams between the subqueries and the outer queries, but the user isn't required to do anything other than write the subquery.

The use of subqueries is principally designed to minimize the need to define additional intermediate data streams. Any query with subqueries can always be restructured as a query pipeline with explicit intermediate data streams. In some cases, this can be helpful, particularly when debugging a Sybase CEP application, because intermediate data streams can be observed in the Studio and adapters can be temporarily attached to intermediate streams to capture the stream data. This is not currently possible with subqueries.

Joining a Data Source to Itself (Self-Join)

The same data source may be referenced more than once in the FROM clause by using different aliases for each copy of the same data source.

Each mention of the data source under a different alias may include its own KEEP clause, allowing you to base more than one type of window on the same data stream. References to the different aliases in the rest of the query are then treated as references to different data sources although the data sources actually have the same incoming data.

Consider the need to compare the incoming volume for a specific trade with the cumulative volume for the preceding 10 seconds (again, this example period is for illustration with our limited data stream, but this period could be 15 minutes, an hour or longer). You would like to see this comparison every time a new trade arrives for any stock symbol.

Here is an example that joins individual rows of data in the StockTrades stream with a window which is also based on the StockTrades stream:

```
INSERT INTO VolumeComparison
SELECT StockTrades.symbol,
 StockTrades.volume, SUM(STWindow.volume)
FROM
 StockTrades,
 StockTrades AS STWindow KEEP 10 SECONDS
WHERE StockTrades.symbol = STWindow.symbol
GROUP BY StockTrades.symbol;
```

The VolumeComparison stream has the following schema:

Column	Datatype	Description
symbol	String	The symbol for this trade.
volume	Integer	The volume for this trade.
vol10sec	Integer	The volume over the previous 10 seconds for this stock.

In order to perform the volume calculation, a window is required on the StockTrades stream with a window policy that retains rows for 10 seconds. Though the two data sources refer to the same stream and the same columns, they are now treated as different and distinct. Rows that enter the StockTrades data stream are also routed to StockTradesWindow where they are retained for 10 seconds.

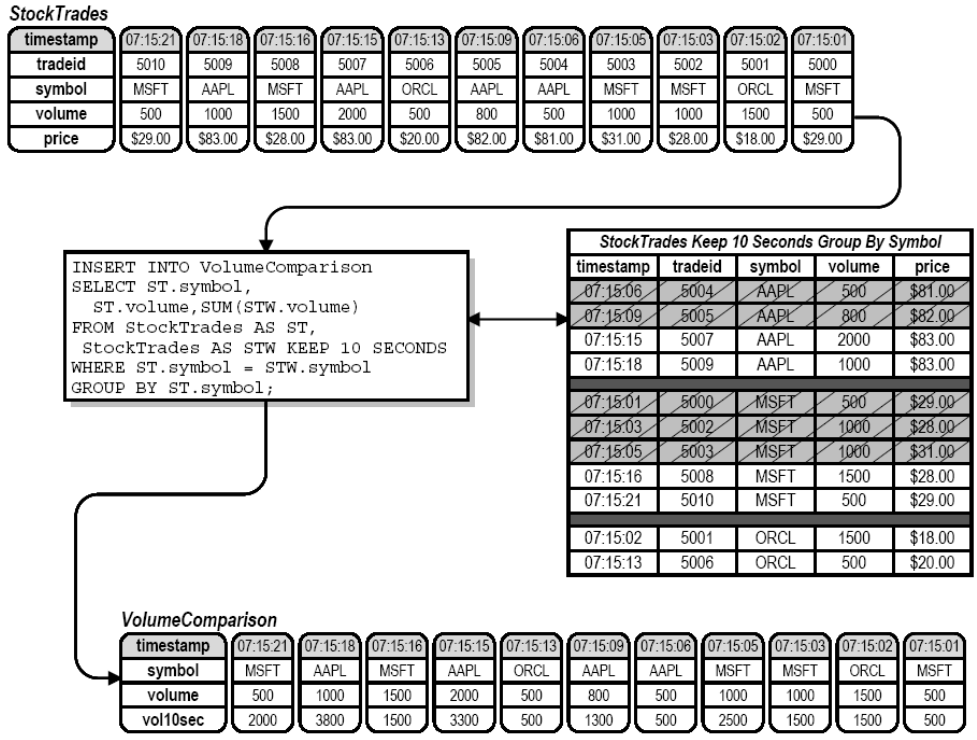
The GROUP BY clause groups the resulting output by stock symbol. Also, as noted in the Window chapter, it breaks the window into subwindows, each containing the data for one stock symbol.

The WHERE clause defines a join condition that matches the row in the StockTrades stream with the same group of rows in the window.

The SELECT clause calculates the sum of the volumes for the rows in the window and aggregates down to a single output row for each incoming stock trade.

Note: Since the result of the join operation ensures that the symbol from both the StockTrades stream and the window is the same for any given row, it doesn't matter which data source is used as the source of the symbol column, but it is mandatory that one of the sources be chosen to qualify the column name symbol. Since both StockTrades and StockTradesWindow contain the symbol column, a compile error would result if the unqualified name symbol were used in the query.

Here is an illustration of how this self-join query works:



Controlling Output Timing

Use ouput clauses to delay or regulate the processing of output from a query.

Normally, you can expect output from a query to be delivered immediately upon execution. When a row comes into an input stream, all queries that act upon that stream will execute and, if appropriate, produce output based on that incoming row. Output rows that are produced as a result of query execution will have the timestamp of the incoming row and those output rows, if directed to other queries, will be acted upon immediately. Thus, all actions that act on the incoming row or any of its derived results will appear to occur simultaneously.

However, it may be appropriate to delay or regulate the processing of output from a query, so that "downstream" actions on the results of a query do not occur at the same time. Sybase CEP has a number of ways to control and regulate the processing of query output:

- OUTPUT AFTER.
- OUTPUT EVERY.
- OUTPUT ALL EVERY.

- OUTPUT FIRST WITHIN.

Using OUTPUT AFTER

The OUTPUT AFTER clause can be used to delay the output either by a specific time interval or by a specific number of rows.

Interval Delays With OUTPUT AFTER

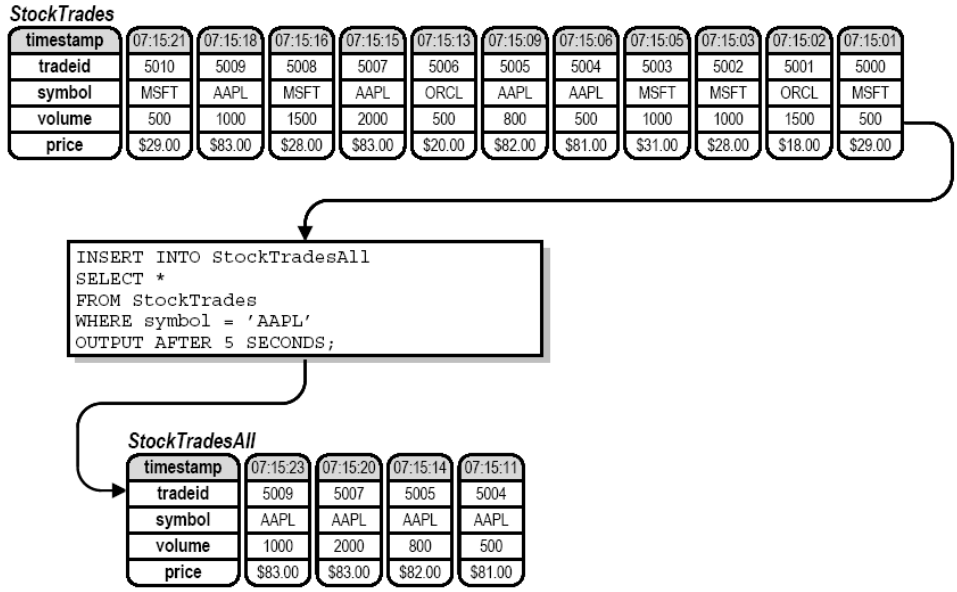
Delay simple filter queries with the OUTPUT AFTER clause so that all rows leaving the query will leave five seconds after they enter.

```
INSERT INTO StockTradesAll
SELECT *
FROM StockTrades
WHERE symbol = 'AAPL'
OUTPUT AFTER 5 SECONDS;
```

This query is very similar to one of the queries in an earlier chapter except for the addition of the OUTPUT AFTER clause. In the original query, the output rows came out of the query at the same time as they went in, with the original timestamp intact. In this query, the output rows will be delayed by five seconds and will have new timestamps based on the delay.

The delay that is introduced will cause these rows to be processed by any subsequent queries in the query pipeline of the module five seconds after the rows arrived in the original query.

Here is an illustration of how this delayed output works:

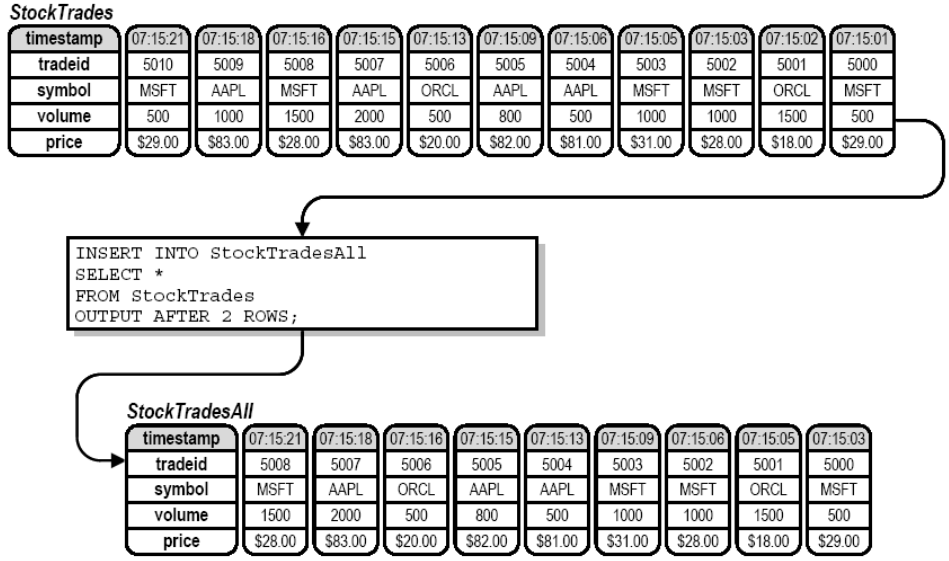


Row Count Delays With OUTPUT AFTER

Delay output rows by a specific number of rows. A description of how to enable row delay, and what syntax to use.

```
INSERT INTO StockTradesAll
SELECT *
FROM StockTrades
OUTPUT AFTER 2 ROWS;
```

Here is an illustration of how row delay works:

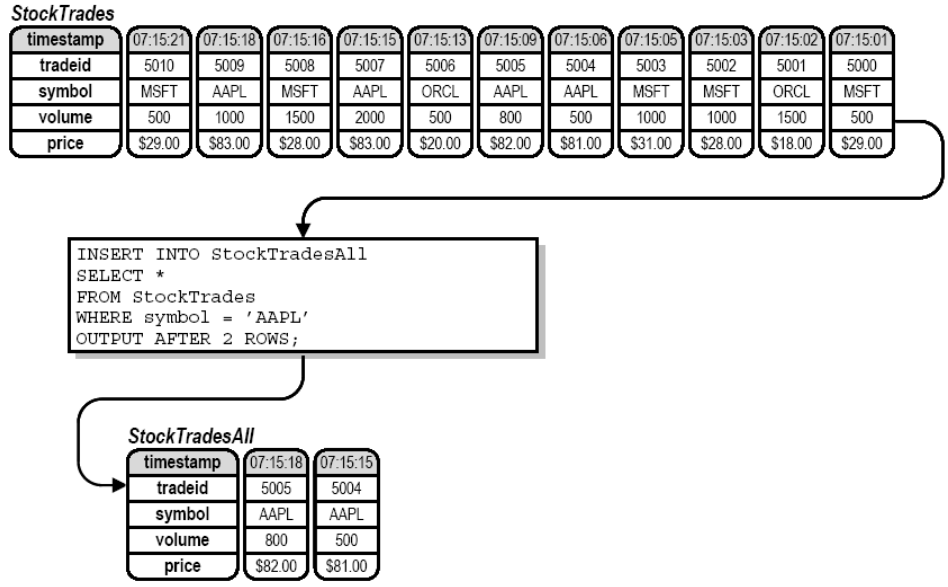


In the above query, the first row in the input stream was published to the output stream when the third row in the input stream arrived and was sent to be published (delaying the first row by two rows). Note that after the row with tradeid 5008 is published (when tradeid 5010 arrives), no other output appears to be published. Since the example StockTrades data only has 11 events, the last two events won't be published until more rows arrive, permitting the OUTPUT AFTER 2 ROWS clause to be satisfied.

Note: This does not happen with interval-based delays, because regardless of whether other rows arrive, time continues to be monitored and when Sybase CEP's internal clock reaches the point when the rows should be published, the rows will be published.

When a row count is used with OUTPUT AFTER, the row count delay is actually based on the number of rows being published to the output stream, not the rows arriving in the input stream. When there is no filter in the query, as in the above example, these events are equivalent (since all rows arriving in the input stream will be simultaneously ready for the output stream).

The introduction of a WHERE clause that filters output causes a change in the timing of output publication:



Note that when a WHERE clause is used, the first row that is published (the row with tradeid 5004) is delayed until the row with tradeid 5007 (not tradeid 5006) is processed. This is because the row with tradeid 5006 is not published at all (it doesn't meet the filter condition) and is not counted when delaying the output rows.

Using OUTPUT EVERY

The OUTPUT EVERY clause permits the regulation of row publication at specific intervals, based either on time or row counts. This reduces the number of rows that are published.

Interval Timing With OUTPUT EVERY

Use OUTPUT EVERY to limit the number of rows generated by a query and regulate the timing of the output so that the result rows arrive at a predictable interval.

Consider the situation in a stock trading application where it is desired to publish the total number of shares traded on a regular basis during the day (many different kinds of data might be desired, such as average volume, average price, and so on, but what is published is not necessarily critical to understanding the concept of output timing, only that some value or set of values is published). Previously, an aggregator was shown accumulating the total number of shares over a specific interval, but the output stream contained a result row for every incoming row that was processed with a continuously changing value for total number of shares.

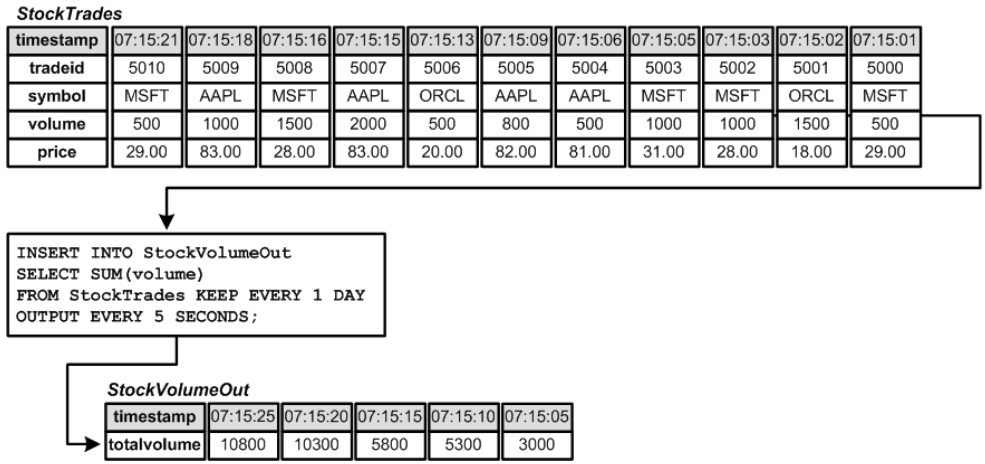
Use the OUTPUT EVERY clause as follows:

```
INSERT INTO StockVolumeOut
SELECT SUM(volume)
FROM StockTrades KEEP EVERY 1 DAY
OUTPUT EVERY 5 SECONDS;
```

The output of the query (in this case, the expression SUM(volume)), which calculates the total volume of the current day's trades (KEEP EVERY 1 DAY), is sent to the *StockVolumeOut* stream:

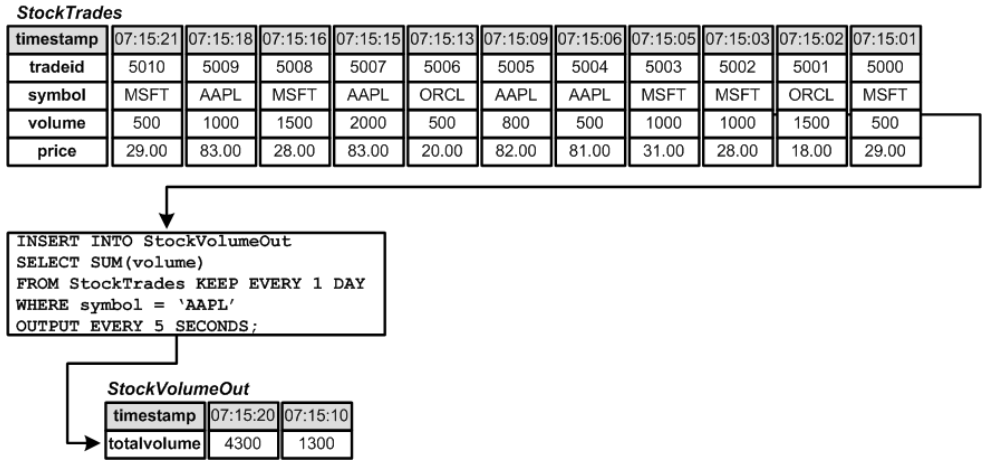
Column	Datatype	Description
totalvolume	Integer	The total volume for the current day's trading.

Here is an example of how this query generates output rows every five seconds, regardless of when data arrives:



Notice that the output row timestamps are not directly related to the input row timestamps (although some output rows do occur at the same moment that input rows arrive). Instead, the output rows are exactly five seconds apart. This is the regulating mechanism of OUTPUT EVERY which allows you to specify precisely when output rows will be generated.

Let's look at the same example with a WHERE clause that filters out only the AAPL rows:

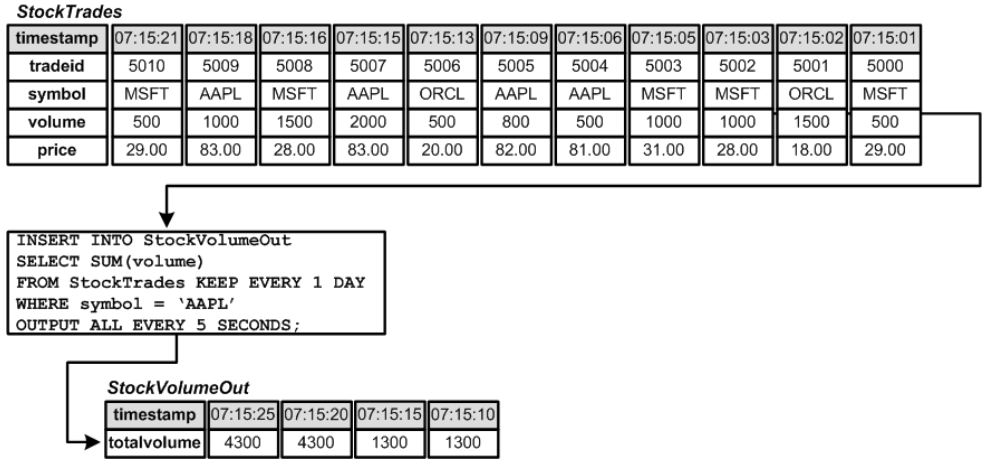


In this example, only the AAPL volume data is summed. Note that because there are no rows at all prior to 07:15:10, there is no output row for 07:15:05. However, also note that there is no row for 07:15:15. This is because there are no AAPL rows that arrive between 07:15:10 and 07:15:15.

Publishing Rows For Every Interval

If you want to regulate the publication of rows so that a row is always generated for every interval, even if no rows arrive during that interval, use the OUTPUT ALL EVERY option.

This forces Sybase to publish the last valid result from an OUTPUT EVERY clause even if no new rows have arrived since the last interval:

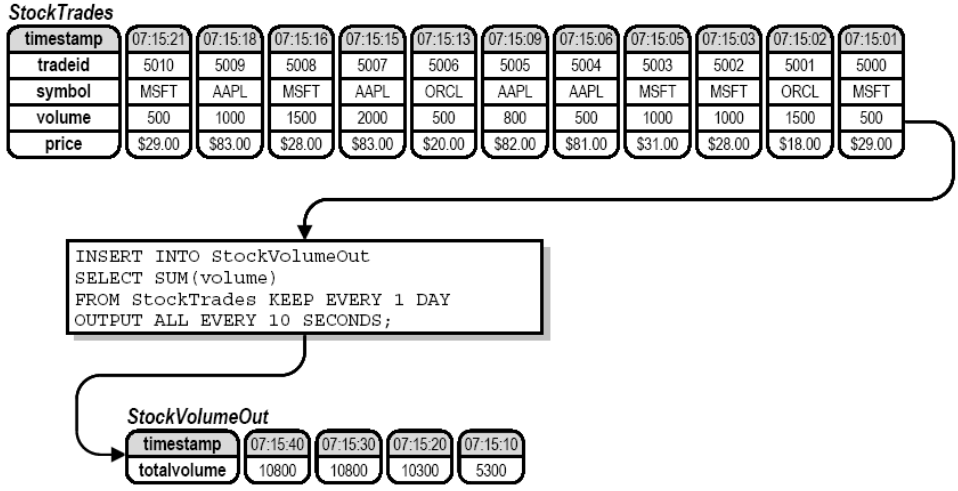


Notice that now there is a row at 07:15:15, even though there was no AAPL row during that interval. In addition, notice that with OUTPUT ALL EVERY, publication of the last valid row continues repeating every interval even if no new rows arrive.

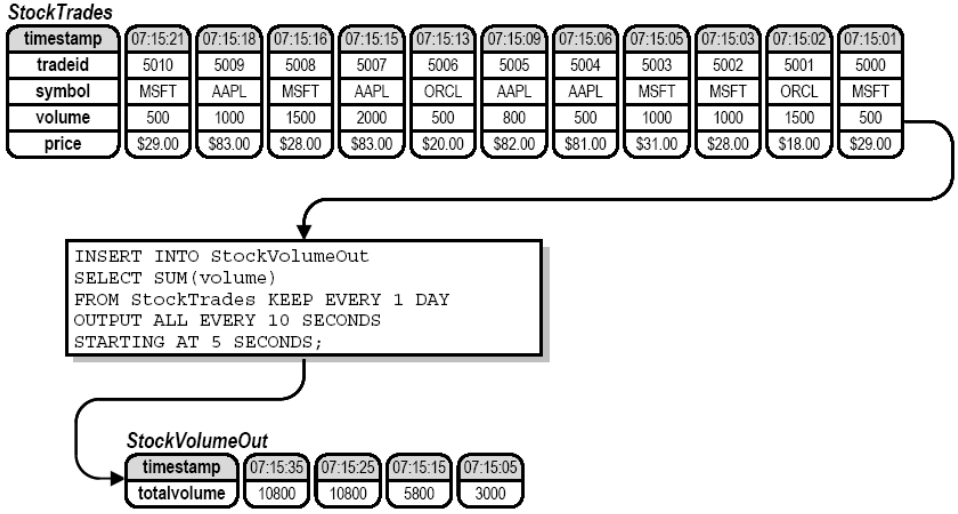
Modifying the OUTPUT EVERY Interval

Use the STARTING AT option with the OUTPUT EVERY or OUTPUT ALL EVERY clause to offset row publication.

Consider the following example which illustrates publishing total volume data every 10 seconds (compared to every 5 seconds as in previous examples):



In this example, rows are published every ten seconds, precisely on the ten second interval. Similar to the way windows behave with intervals, the intervals used for regulating output publication also start at the "epoch" (January 1, 1970) and the intervals are calculated as equal increments from that point. If an application requires a different interval starting point, you can use the STARTING AT option with the OUTPUT EVERY or OUTPUT ALL EVERY clause to create an offset. The value provided with the STARTING AT option is used to adjust the starting point from the "epoch" so that all intervals are adjusted respectively. In the following example, the interval size will still be ten seconds (KEEP EVERY 10 SECONDS), but will occur on five second boundaries (in other words, :05, :15, :25, and so on) because of the STARTING AT 5 SECONDS clause:



Notice that the first two rows published at 07:15:05 and 07:15:15 have different values than in the previous example, because the intervals, although the same size, now start and end at different times.

Grouping OUTPUT EVERY Rows

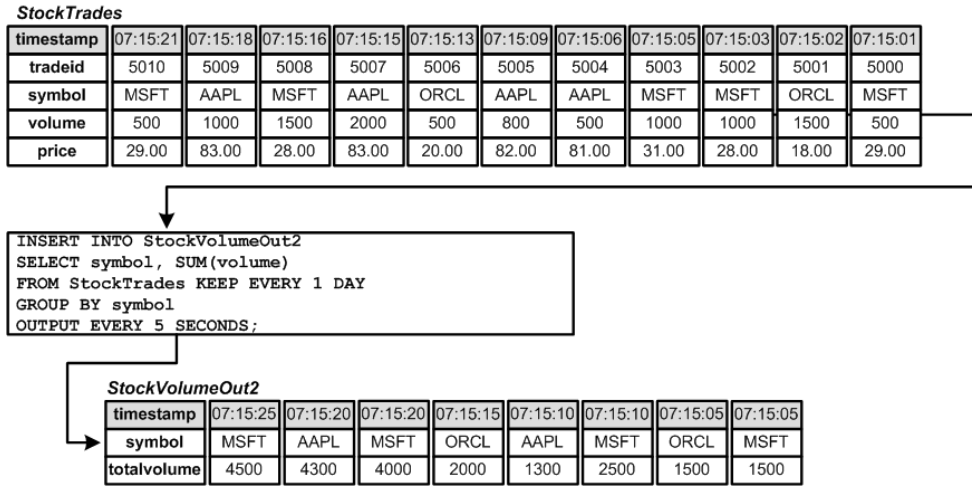
Use the GROUP BY clause to establish a grouping for an OUTPUT EVERY clause.

The grouping causes all output rows with the same grouping value to be treated as a separate set of rows from all other groups. Each group is published separately but all groups are published at the same interval.

The output schema should be modified to include the grouped column (symbol). Sybase CEP will not prevent the publication of grouped values without the grouping columns, but there will be no way to identify which rows in the output stream belong to which group. The output will be sent to the StockVolumeOut2 stream:

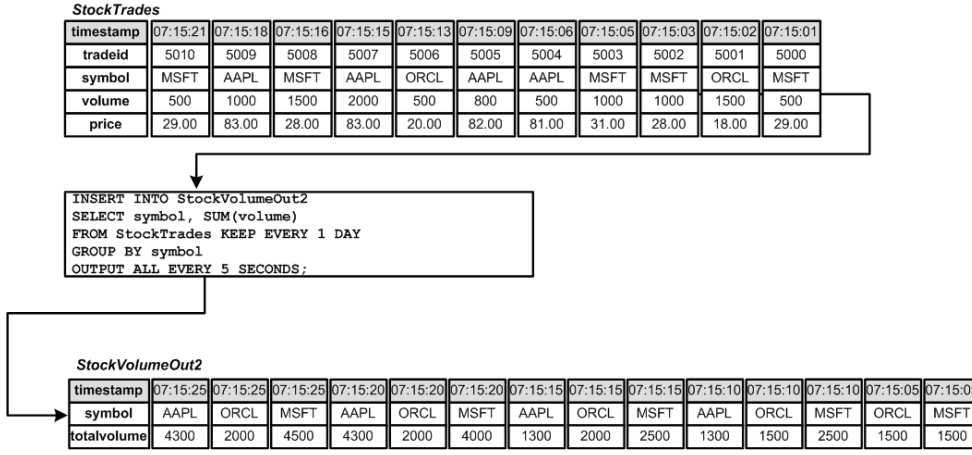
Column	Datatype	Description
symbol	String	The value of the stock symbol for each group.
totalvolume	Integer	The total volume for the current day's trading for this stock symbol.

First, here is an example of OUTPUT EVERY with a grouping by symbol:



Notice that each interval may have more than one row, depending on what rows entered the window during each interval. Only groups that have new data for an interval are published at the end of the interval. This is why the interval ending at 07:15:15 only has an ORCL row, because no other symbols occurred between 07:15:10 and 07:15:15. Also notice, however, that the output for the interval is still based on the window, which is larger than the interval, so that data from previous intervals is accumulated (the window keeps data for one day).

Here is a similar example using OUTPUT ALL EVERY. As might be expected, this publishes values for each group for every interval (once there is data from the group in the window at all), regardless of whether any new rows have entered the window during the preceding interval:

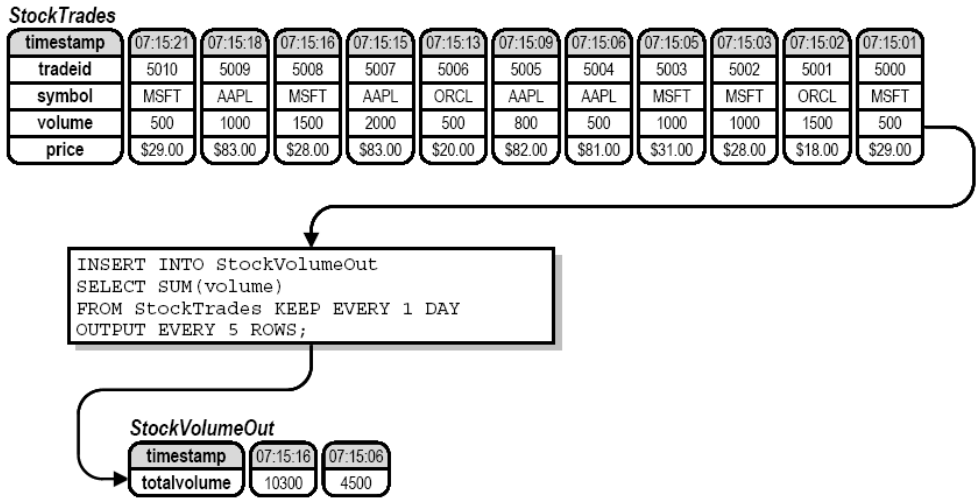


Row Count With OUTPUT EVERY

Use the OUTPUT EVERY clause with a row count. Provides an example of how to apply the OUTPUT EVERY clause to a row count, as well as, the syntax rules for the clause.

```
INSERT INTO StockVolumeOut
SELECT SUM(volume)
FROM StockTrades KEEP EVERY 1 DAY
OUTPUT EVERY 5 ROWS;
```

Here is an example of how this query generates output rows every 5 rows, regardless of when data arrives:



Notice that the output row timestamp is the value from the 5th and 10th input rows respectively.

The OUTPUT ALL EVERY syntax is permitted for row count intervals, but it has no impact on the query, since there is no way rows can be missing from a row count interval (by definition, the interval is defined based on the very fact that rows have been published).

Grouped queries are also permitted with OUTPUT EVERY row count syntax. As with timed intervals, the output is broken into groups. However, unlike timed intervals, where all groups are published at each interval, each individual group is published based on the number of rows specified in the row count for that group.

Using OUTPUT FIRST

Use the OUTPUT FIRST WITHIN clause to return values at the start of timed or row count intervals.

```
INSERT INTO StockVolumeOut
```

```
SELECT SUM(volume)
FROM StockTrades
KEEP EVERY 1 DAY
OUTPUT FIRST WITHIN 10 SECONDS;
```

With OUTPUT FIRST WITHIN, the first row generated by the query is published and, concurrently, an interval starts. Both timed and row count intervals are permitted.

For timed intervals, all rows generated after the published row which starts the interval are then ignored, until the end of the interval. For row count intervals, the next N subsequent rows generated by the query are ignored where N is the number of rows specified in the OUTPUT FIRST WITHIN clause.

Once the interval has completed, the next row to be generated by the query is published, regardless of when it arrives. This starts a new interval and interval processing repeats as described above.

Finding Patterns In Data

Continuous Computational Language (CCL) queries can use a special pattern matching syntax that enables a query to detect the occurrence of certain patterns of events.

When a MATCHING clause is added to a query, the query detects a match on the pattern by looking at two or more data sources within a specified interval. Rows are published from the query for each successfully detected match.

Finding a Sequence of Events

Run a pattern-matching query to isolate a sequence of events in your data.

Here is a simple pattern-matching query:

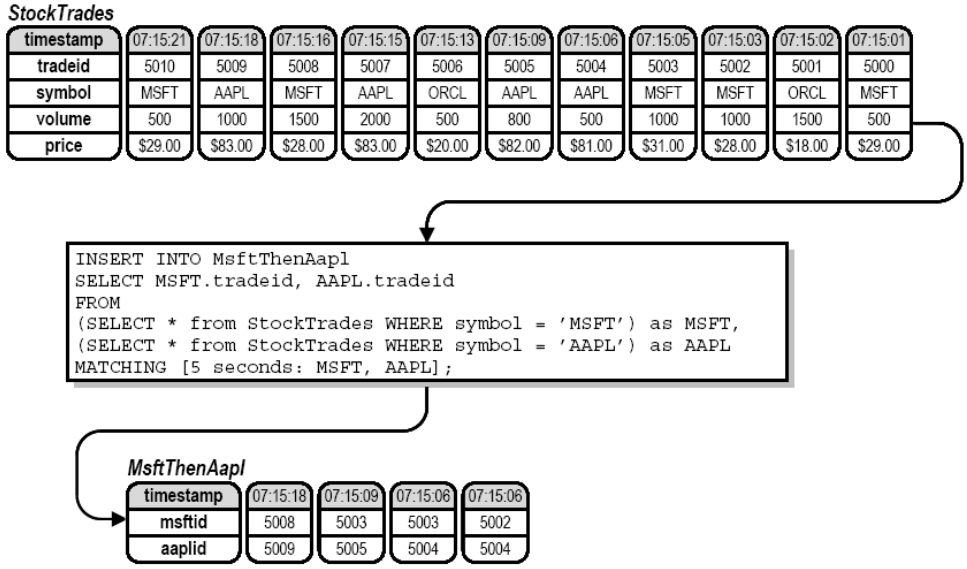
```
INSERT INTO MsftThenAapl
SELECT MSFT.tradeid, AAPL.tradeid
FROM
   (SELECT * from StockTrades WHERE symbol = 'MSFT') as MSFT,
   (SELECT * from StockTrades WHERE symbol = 'AAPL') as AAPL
MATCHING [5 seconds: MSFT, AAPL];
```

Note the following important points:

- The FROM clause must contain all the data sources mentioned in the matching clause. Note that in this query, the subquery and alias technique (`subquery AS alias`) is applied to provide two different streams from the *StockTrades* stream, one with the trades for MSFT and one with the trades for AAPL.
- None of the data sources mentioned in the FROM clause may be windows or window expressions.
- The matching condition starts with the word MATCHING and is enclosed in square brackets.

- The interval in the MATCHING clause specifies a sliding time period within which the desired pattern must be detected. The time period is triggered by the arrival of the first event in the sequence.
- The desired pattern is expressed by giving the names of the data sources after the interval, in the order desired and possibly using special pattern matching operators to modify the type of pattern search that will be performed.
- The comma operator, also called the sequence operator, means that an event from the first data source must be followed by an event from the second data source within the specified interval.

The query finds any trade for MSFT which is then followed by a trade for AAPL within five seconds. Here is an example of the output from this query:



Note that the timestamp associated with the output from the pattern match is the timestamp of the final event. Also notice that it is possible to get more than one match from an event, in which case the event will appear in the output stream multiple times. If two events in the first stream occur before two matching events in the second stream (and all four events occur within the interval), you will get four outputs: $MSFT^1$-$AAPL^1$, $MSFT^1$-$AAPL^2$, $MSFT^2$ - $AAPL^1$, and $MSFT^2$ -$AAPL^2$.

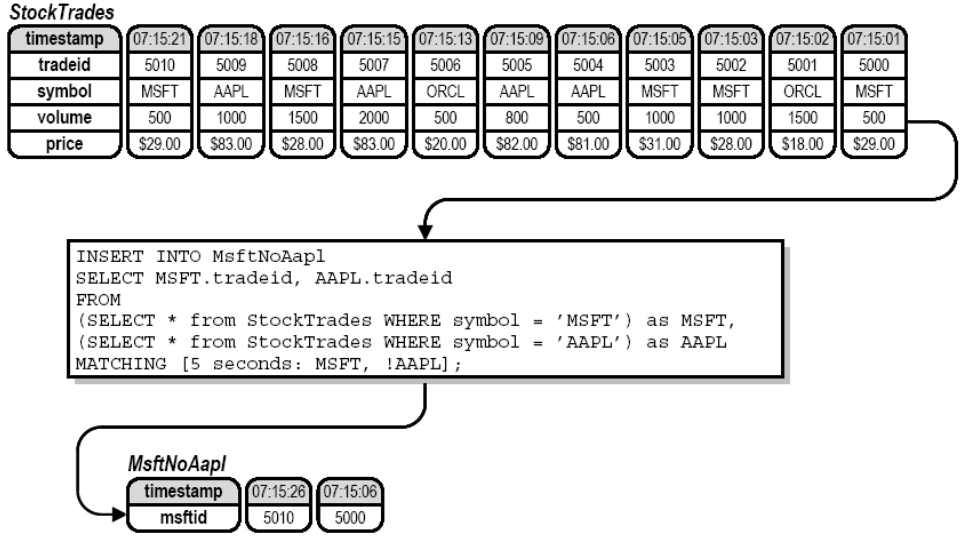
Detecting Non-Events

An important pattern-matching operator is the "!" or "Not" operator.

In the previous query, a sequence of events was detected (a stock trade for MSFT followed by a stock trade for AAPL). Use the "Not" to detect a "non-event", that is an event in one stream that is *not* followed by an event in another stream. This permits the detection of missing events from a variety of sequences:

```
INSERT INTO MsftNoAapl
SELECT MSFT.tradeid, AAPL.tradeid
FROM
 (SELECT * from StockTrades WHERE symbol = 'MSFT') as MSFT,
 (SELECT * from StockTrades WHERE symbol = 'AAPL') as AAPL
MATCHING [5 seconds: MSFT, !AAPL];
```

This query finds any trade for MSFT which is not followed by a trade for AAPL within five seconds. Here is an example of the output from this query:



Note that the timestamp associated with the output from pattern matching with non-events will be the timestamp when the matching interval expires after the initial event is detected.

Other Pattern-Matching Operators

There are a number of pattern-matching operators that you can use inside a MATCHING clause.

Operator	Symbol	Description
Sequence	,	The events separated with the sequence operator must appear within the event interval in the sequence specified in the MATCHING clause. A,B means that event B must follow A in order to match. Sequences may not contain two non-events. Thus A, !B, C is valid but !A, B, !C is invalid.
Negation	!	The non-event may not appear in the event interval.

Operator	Symbol	Description
Conjunction ("and")	&&	Both events must appear in the event interval but either event may occur first.
Disjunction ("or")	||	Either event (or both) may appear in the event interval.

Matching Associated Events

Use the stream alias and self-join techniques to join rows from two different events with an associated key value.

It is possible that two sources of events may contain rows that need to be related by an associated key value. For instance, if you want to detect stock trades in which the price of the stock has gone up by more than $1 within the past 5 seconds, you might consider writing the following matching query:

```
INSERT INTO StockGains
SELECT First.tradeid, Second.tradeid
FROM StockTrades as First,
   StockTrades as Second
MATCHING [5 seconds: First, Second]
WHERE (Second.price - First.price) > 1.0;
```

This uses the stream alias and self-join techniques to use a single stream as if it were two unique streams. This allows you to compare the events in a stream with itself. The above query will generate results, but it will match every row in StockTrades. A more likely desired result is to only compare trades that have the same symbol. This can be done using the ON clause to specify the matching condition between the two streams:

```
INSERT INTO StockGains
SELECT First.tradeid, Second.tradeid
FROM StockTrades as First,
   StockTrades as Second
MATCHING [5 seconds: First, Second]
ON First.symbol = Second.symbol
WHERE (Second.price - First.price) > 1.0;
```

This looks very similar to earlier join conditions using the WHERE clause. However, note that in an ON clause, only the equal sign may be used to compare columns. Because the ON clause is specifically limited to providing join conditions, it can be optimized in ways that a regular WHERE clause can't.

If more than two events need to be matched, the ON clause permits a shorthand notation for associating columns in the data sources:

```
INSERT INTO MultipleTrades
SELECT First.tradeid, Second.tradeid, Third.tradeid
FROM StockTrades as First,
   StockTrades as Second,
```

```
   StockTrades as Third
MATCHING [5 seconds: First, Second, Third]
ON First.symbol = Second.symbol = Third.symbol;
```

This query detects three consecutive trades for any given stock within the interval.

Accessing External Databases

Sybase CEP Engine can communicate with external data sources via input and output adapters.

These external data sources are diverse and may include relational databases. However, in the special case of relational databases, Sybase CEP provides additional means of communication with the database directly from within CCL queries, without using input or output adapters. This chapter teaches you how to write CCL statements that exchange data with external relational databases.

Sybase CEP supports CCL connections to external databases that use a standard OBDC driver, or, for use with the Oracle relational database, the Oracle native driver. To use the features described in this chapter, you must first obtain and install the appropriate driver on your machine and then configure the Sybase CEP connection with the external database. For more information on installing and configuring your database connection, please refer to the *Sybase CEP Installation Guide*.

Database Subqueries

A CCL query statement can incorporate data directly from an external database by means of a database subquery, which is nested in the CCL query as part of the FROM clause.

When the CCL query is running, the database subquery is passed to an external database server, such as Oracle. The external database executes the subquery, selects the appropriate data, and passes the data back to the CCL query. Once the Oracle data is returned to Sybase CEP and incorporated into the output data published by the query to its output stream, it can be processed by other CCL queries in the same manner as all other Sybase CEP data.

A database subquery is always joined to exactly one Sybase CEP data stream (not to a window). Database subqueries are quite different from regular subqueries (which are simply CCL queries nested within other CCL statements). Database subqueries use special syntax and follow very restricted rules.

A database subquery is continuous and conceptually executes every time its associated Sybase CEP query executes. However, as database information is often relatively static and may not require repetitious execution, performance can be improved by reducing the frequency of database subquery execution by the use of caching. For a discussion of caching external database data, please see the *Sybase CEP Installation Guide*.

Let's assume that there is an external SQL database called *StockDatabase* which has a table of historical stock prices called *StockHistory* with the following schema:

Column	Datatype	Description
symbol	String	The symbol of the stock.
avgprice	Float	The average historical price of the stock.

Note: The type of SQL database is not important to this discussion because the SQL query involved will be common to all normal databases such as Oracle, MySQL, and so on. Whatever database is available at your installation can be accessed with a database subquery, as long as it has the necessary interface driver.

Here is an example of a simple database subquery using the StockTrades stream to pick up historical data from the StockHistory table:

```
INSERT INTO StockWithHistory
SELECT current.symbol, current.price, history.avgprice
FROM StockTrades AS current,
(DATABASE 'StockDatabase' SCHEMA (avgprice FLOAT)
[[SELECT avgprice FROM StockHistory
 WHERE symbol = ?current.symbol]]) AS history;
```

Notice the following things about a CCL query that contains a database subquery:

- The database subquery is contained inside parentheses like any other subquery, but it starts with the word DATABASE followed by the name of the database as it is known to the interface software (for example, ODBC or, in the case of Oracle, the Oracle driver). The appropriate security for the external database must be provided in the interface software (see the Sybase CEP Administration Guide for details).
- The schema of the returned data must be provided as the next part of the database subquery. The schema definition starts with the word SCHEMA and is followed either by a Sybase CEP schema filename or an internal schema definition enclosed in parentheses. The internal schema definition is simply a set of column names and data types separated by commas. In the example, only one column, avgprice, is returned from the historical database and so only one column is named in the schema.
- The appropriate SQL statement that is needed to extract the data from the external database is enclosed in doubled square brackets. The syntax used here is standard SQL and may contain any valid SQL for the database that is being accessed.

  Note: Sybase CEP does no syntax checking of the SQL statement. If there are any errors in the statement, that will not be detected until the query is actually executed.
- In order to provide the necessary retrieval condition for the data in the database (in this case, the value of the stock symbol to be looked up), values from the Sybase CEP data stream can be used inside the SQL query. The value of current.symbol (where current is the alias for the *StockTrades* stream) is passed to the subquery and it is inserted in the WHERE clause of the SQL statement. Substituted columns from the data stream must be preceded with a question mark (?) in order to be identified properly. Sybase CEP substitutes the

value at run-time from the current row in the data stream before invoking the SQL query in the external database.
- The database subquery is given an alias in the same way as other data streams and windows, using the AS clause, so that its schema columns can be referenced in the CCL query.
- If the datatypes used by the external database do not exactly match the datatypes used by Sybase CEP, you may need to modify the datatypes of Sybase CEP data and/or external database data, before retrieving the database from a CCL query. For more specific information about CCL datatype conversion between CCL and other external databases, please see the Sybase CEP Reference.

Modifying Data in External Databases

Use the Database statement to send instructions to modify external relational databases directly from Sybase CEP. The Database statement can execute any valid SQL Update, Insert, or Delete statement in the associated external database.

Assume that a Sybase CEP query is creating ongoing average price data from incoming stock trades. This query sends its information to a stream called StockAvgPrice. Here is an example of how the Database statement can read the StockAvgPrice stream and update the external database:

```
EXECUTE STATEMENT DATABASE 'StockDatabase'
[[UPDATE StockHistory
 SET StockHistory.avgprice = ?avgpr
 WHERE StockHistory.symbol = ?symb]]
SELECT symbol AS symb, avgprice AS avgpr
FROM StockAvgPrice
```

The syntax of the Database statement is very similar to the database subquery, except that it is a complete statement of its own which is not embedded in an INSERT/SELECT statement. Note the following differences and important points:
- No schema is needed with the Database statement, because Sybase CEP simply sends the data to the external database and does not get any results back.
- The SQL inside the square brackets can be any valid SQL for the external database but it can not return a result. Typically, this includes UPDATE, INSERT and DELETE statements, but it can also be used to execute stored procedures as long as the stored procedure doesn't return a result set or a cursor.
- The SELECT clause using after the SQL statement follows almost all Sybase CEP SELECT clause rules although there are some exceptions. The principal exception is that every column used in the SELECT list must be given an AS column name alias so that it can be uniquely identified in the body of the SQL statement. This is true even if the names are already unambiguous.
- Also, because names must be given to each column, SELECT * is not permitted. For other minor differences, see Sybase CEP CCL Reference.

For every row that is published by the SELECT clause, a call is made to the external database, passing the entire SQL statement contained in the square brackets after all ? references are

converted to their actual Sybase CEP data values. The external database is responsible for executing the SQL statement. Errors are reflected back to the Sybase CEP server and can be found in the Sybase CEP log files.

Index

A

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W

X