RapidsDB

RapidsDB Streaming Guide

Release 3.0

Beta

RapidsDB Streaming Guide

Table of Contents

1. What is RapidsDB?

1.1 What is RapidsDB?

RapidsDB is a fully parallel, distributed, in-memory federated query system that is designed to support complex analytical SQL queries running against a set of different data stores. The diagram below shows the major components of the RapidsDB system:

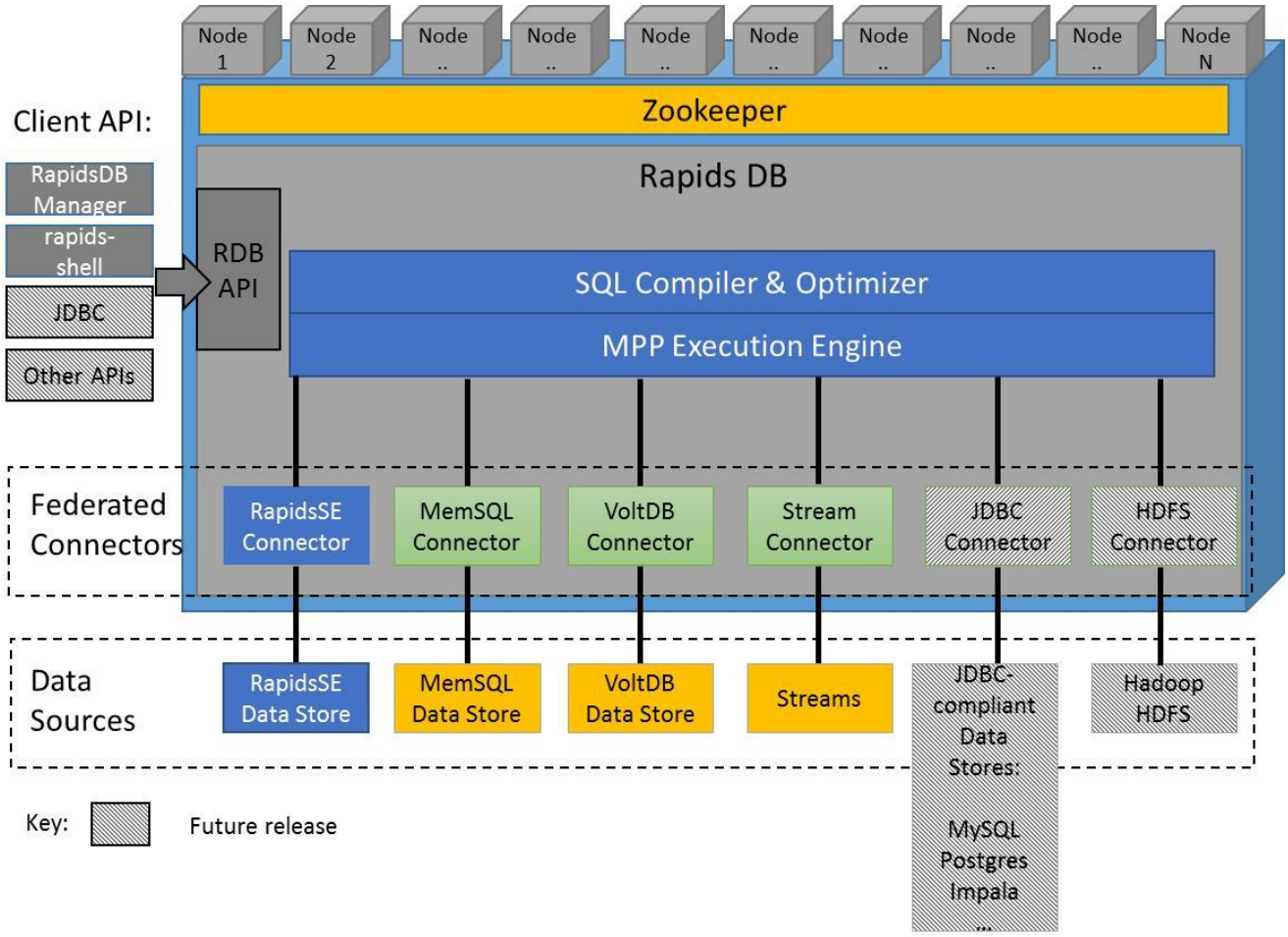


Figure 1. RapidsDB architecture

RapidsDB provides unified SQL access to a wide variety of data sources, which can include relational and non-relational data sources. Data can be joined across all of the data sources. For Release 3.0 Beta, RapidsDB supports access to its own internal in-memory data store, RapidsSE, the MemSQL (http://www.memsql.com/) and VoltDB (https://voltdb.com/) in-memory data stores and also supports access to streaming data sources. For the next release, support will be provided for any data sources that support a JDBC interface, as well as supporting access to data that is stored in Hadoop HDFS.

1.2 RapidsDB Components

1.2.1 SQL Compiler and Optimizer

RapidsDB has an advanced SQL Compiler & Optimizer that is responsible for taking a user's SQL query and building a query plan that will take full advantage of the native SQL capabilities of the underlying data source . The generated query plan will push down those parts of the query plan that can be executed directly by the data source and then execute the remainder of the plan using the RapidsDB MPP Execution Engine by only pulling up the data from the underlying data sources that is needed to complete the execution of the query.

1.2.2 MPP Execution Engine

RapidsDB has its own fully parallel, MPP Execution Engine that is responsible for execution of the query plan generated by the RapidsDB SQL Compiler & Optimizer. The MPP Execution Engine will access the underlying data sources using the Federated Connectors.

1.2.3 Federated Connectors

RapidsDB supports a plug-in Data Connector technology for interfacing with the underlying data sources. The Connectors provide a standard ANSI 3-part naming interface (catalog.schema.table) to the data that is managed by the data sources. For those data sources that do not support ANSI 3-part naming, the Connector will provide the missing parts of the ANSI 3-part name. For example, for a streaming data source the Stream Connector will provide a catalog and schema name. During query execution, the Connectors are responsible for executing those parts of the plan that the Optimizer has generated for the associated data store, and then passing up the results of the query execution to the RapidsDB MPP Execution Engine.

1.2.4 RapidsSE

RapidsSE is an in-memory storage engine that is tightly integrated with the RapidsDB Execution Engine. The data managed by RapidsSE is stored in a shared memory segment that can be accessed directly by the RapidsDB Execution Engine. For the Beta release, RapidsSE can run on a single node in the RapidsDB cluster. At this time, RapidsSE has no query processing capabilities, all query execution is handled by the RapidsDB Execution Engine, with RapidsSE purely providing in-memory storage for the data (in a csv format). RapidsDB is playing a similar role to Hive in Hadoop when accessing data stored in HDFS files, the difference being with RapidsSE the data is stored in memory.

1.2.5 Client API

RapidsDB provides a web-based management console, the RapidsDB Manager, for configuring and managing the RapidsDB cluster.

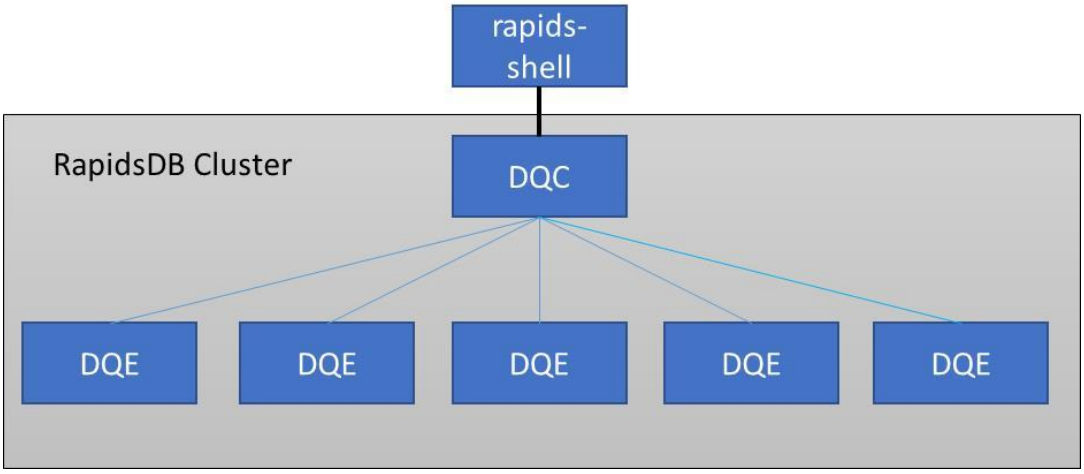
RapidsDB also provides a command line interface via the rapids-shell, and also provides a programmatic interface for Java-based clients. In the next release support will be provided for a JDBC interface and an SDK will be provided that will allow third parties and customers to build native interfaces to other languages, such as C++, python, etc.

1.2.6 Zookeeper

RapidsDB uses Zookeeper (version 3.4.6 or later) for configuration management across the RapidsDB cluster.

1.3 RapidsDB Cluster Topology

The diagram below shows the topology of a RapidsDB Cluster:



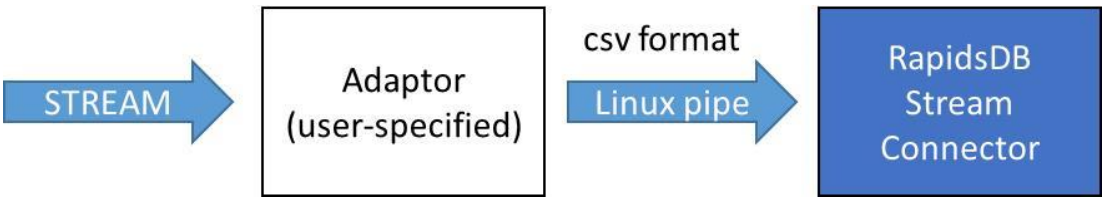
- rapids-shell – provides the command line interface for sending queries to the RapidsDB Cluster
- DQC (Distributed Query Coordinator) – within the RapidsDB Cluster there is one node that must be specified as the DQC. This is the node that the rapids-shell will send queries to. The DQC node can also behave as a DQE node (see below).
- DQE (Distributed Query Executor) – the remaining nodes in the RapidsDB Cluster will perform the query execution and are used to connect to remote data sources via the RapidsDB Connectors (see section 2 for more details on Connectors).

2. RapidsDB Streaming

2.1 Overview

The RapidsDB Stream Connector supports the reading of continuous data streams and as part of the RapidsDB Federation it allows the user to query the streams using SQL and to join the data from a stream with the data from other Connectors. The user can also query a data stream and use the results of the query to create a new data stream, which allows the user to build a complex series of transformations to support CEP processing.

The diagram below shows the RapidsDB Streaming architecture:



The RapidsDB Stream Connector requires that the stream is delivered in a csv format (the field separator must be a comma character ‘,’) over a Linux named pipe. The user is required to transform the stream from its original format into a csv format and to then write the csv data to a pre-existing Linux named pipe.

2.2 Windowing

The Stream Connector will process the data stream using time-based windowing. The stream data will be accumulated for a configurable time window, and once the current time window has expired the query accessing the stream will be completed using the data accumulated for that time window, and then the data will be deleted and a new time window will be started. If there is no active query then no data will be read from the data pipe. The time window is configured when configuring the Stream Connector (see 2.6).

2.3 Stream Metadata

When configuring a Stream Connector, the user must specify the table name to be associated with that data stream along with the schema definition for that stream. The schema definition maps each column in the table to the corresponding field in the csv data that was generated from the Adaptor (see 2.1). The table below shows the column data types supported:

Integer
Float
Decimal
Varchar
Timestamp

Standard SQL exclusions:

1. Varchars cannot have a length specification.
2. Decimals cannot have a precision or scale specification.
3. Null, not null designations are not supported
4. Primary keys are not supported

2.4 Configuring Stream Connectors

To configure a Stream Connector you can either use the Rapids Manager or you can use the CREATE CONNECTOR command from the rapids-shell as described below. Refer to the RapidsDB Installation and Management Guide for details on using the Rapids Manager.

To add a Stream Connector use the following command:

CREATE CONNECTOR <name> TYPE STREAM [WITH window = '<milliseconds>']**
  NODE <node name> [NODE <node name>] [<further node names>]
  CATALOG [* | STREAM]
  SCHEMA [* | PUBLIC]
  TABLE <table name> WITH datapipe = '<pipe name>' [, window = '<milliseconds>'] <table ddl>
  [TABLE <table name> WITH datapipe = '<pipe name>' [, window = '<milliseconds>'] <table ddl>]
  [<further stream tables>];

Where,
<milliseconds>: <integer>

The integer value must be > 0 and < 65536. The default value is 1000. Represents how long to continue trying to read data from the stream (in milliseconds) before issuing an end-of-query. If the Stream Connector is running on multiple nodes, then the window must be applied to all of the nodes by including the WITH window clause after the STREAM keyword (see ** above). The window can also be specified at the table level, in which case there can be different values specified for each table.

NODE <node name>:

The RapidsDB node(s) where the Linux pipe(s) for the streams managed by this Connector are set up. When the streams managed by this Connector are split across multiple nodes, each node must be listed.

CATALOG [* | STREAM]:

The catalog name must be specified and can be either the wildcard character, or "STREAM" (each Stream Connector only has one catalog named "STREAM").

SCHEMA [* | PUBLIC]:

The schema name must be specified and can be either the wildcard character, or "PUBLIC" (each Stream Connector only has one schema named "PUBLIC").

<table name>: the name of the table associated with this stream

<pipe name>: the path to the Linux pipe that will be used to deliver the data for this stream

<table ddl>: USING (<column definition>, …)

column_definition:
  <column name> <data type>

data_type:
  INTEGER
 | FLOAT
 | DECIMAL
 | TIMESTAMP
 | VARCHAR

NOTES:

1. For the Stream Connector the user must provide at least one table name, and that table name cannot be the wildcard ("*") character. Along with the table name, the user must also provide the list of columns and their associated data types in the USING clause.
2. The following standard SQL exclusions apply:
   - Varchars cannot have a length specification.
   - Null, not null designations are not supported
   - Primary keys are not supported

Example commands:

1. CREATE CONNECTOR STREAM1 TYPE STREAM
   NODE NODE2 WITH window = '5000'
   CATALOG *
   SCHEMA *
   TABLE SENSORS WITH datapipe='/home/stdata/sensors'
   USING (sensor_type varchar,
   sensor_ts timestamp,
   sensor_reading integer);

   This command would create a Stream Connector that would handle a single stream on NODE2, with a query window of 5 seconds. Note the use of the wildcard character for the catalog and schema names. Even though for a Stream Connector there is only one catalog and one schema, you must still specify a CATALOG and SCHEMA in the command. You could also specify "STREAM" for the catalog name, and "PUBLIC" for the schema name.

2. CREATE CONNECTOR STREAM1 TYPE STREAM
   NODE NODE2
   NODE NODE3
   CATALOG *
   SCHEMA *
   TABLE SENSOR1 WITH datapipe='/home/stdata/sensor1', window = '5000'
   USING (sensor_type varchar,
   sensor_ts timestamp,

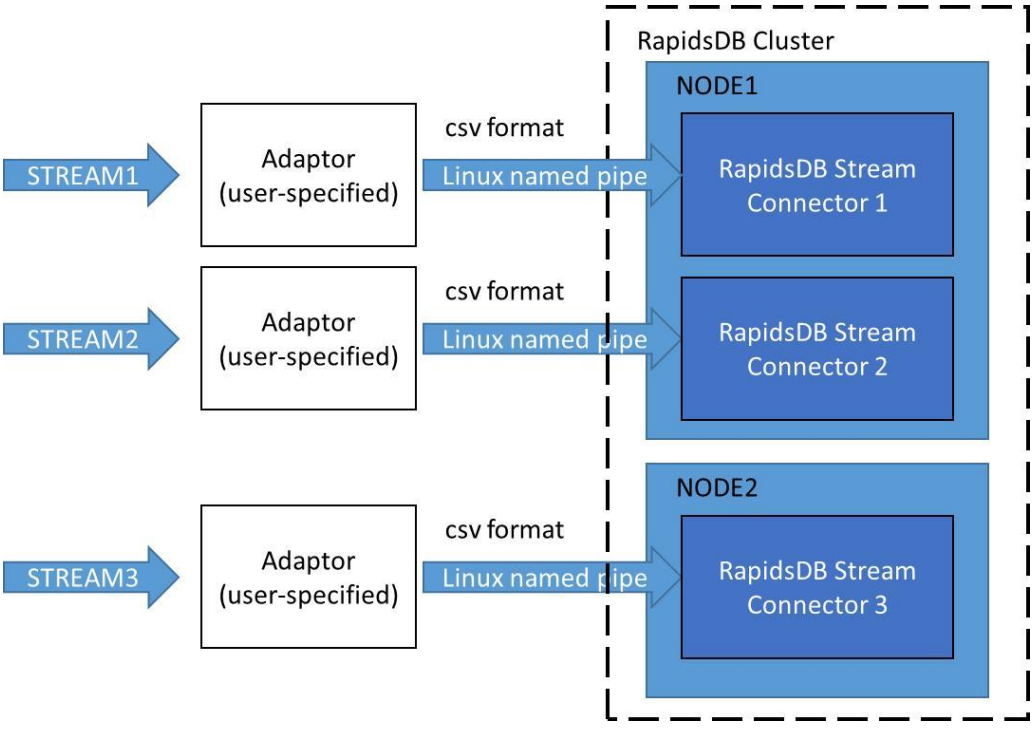
```
                    sensor_reading integer)
TABLE SENSOR2 WITH datapipe='/home/stdata/sensor2'
          USING ( sensor_type varchar,
                    sensor_ts timestamp,
                    sensor_reading integer);
```

This command would create a Stream Connector that would handle two streams that would be split across nodes NODE2 and NODE3. Each stream table has the same 3 columns. The window for the table "SENSOR1" is set to 5000 milliseconds, with table "SENSOR2" using the default of 1000 seconds.

2.5 Stream Topologies

2.5.1 Single Node

A RapidsDB Stream Connector can be configured to run on any node in a RapidsDB Cluster, and there can be multiple Stream Connectors running on one or more nodes in the RapidsDB Cluster, with each Stream Connector supporting a different stream as shown in the diagram below:



The following are the sample commands for creating the Stream Connectors shown above:

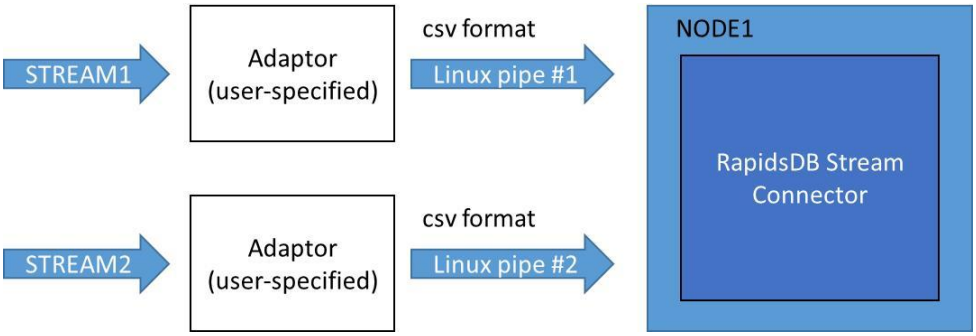
```
CREATE CONNECTOR STREAM1 TYPE STREAM
NODE NODE1
CATALOG *
SCHEMA *
TABLE SENSOR1 WITH datapipe='/home/stdata/sensor1'
        USING ( col1  varchar,
                col2  timestamp);

CREATE CONNECTOR STREAM2 TYPE STREAM
NODE NODE1
CATALOG *
SCHEMA *
TABLE SENSOR2 WITH datapipe='/home/stdata/sensor2'
        USING ( col1  varchar,
                col2  timestamp);

CREATE CONNECTOR STREAM3 TYPE STREAM
NODE NODE2
CATALOG *
SCHEMA *
TABLE SENSOR3 WITH datapipe='/home/stdata/sensor3'
        USING ( col1  varchar,
                col2  timestamp);
```

NOTES:

1. To create a Linux pipe use the command mkfifo, for example:
   mkfifo /home/stdata/sensor3

2.5.2 Multiple Streams Per Connector

A RapidsDB Stream Connector can also be configured (see 2.6) to support multiple data streams with each data stream being delivered over a separate Linux named pipe as shown in the diagram below:

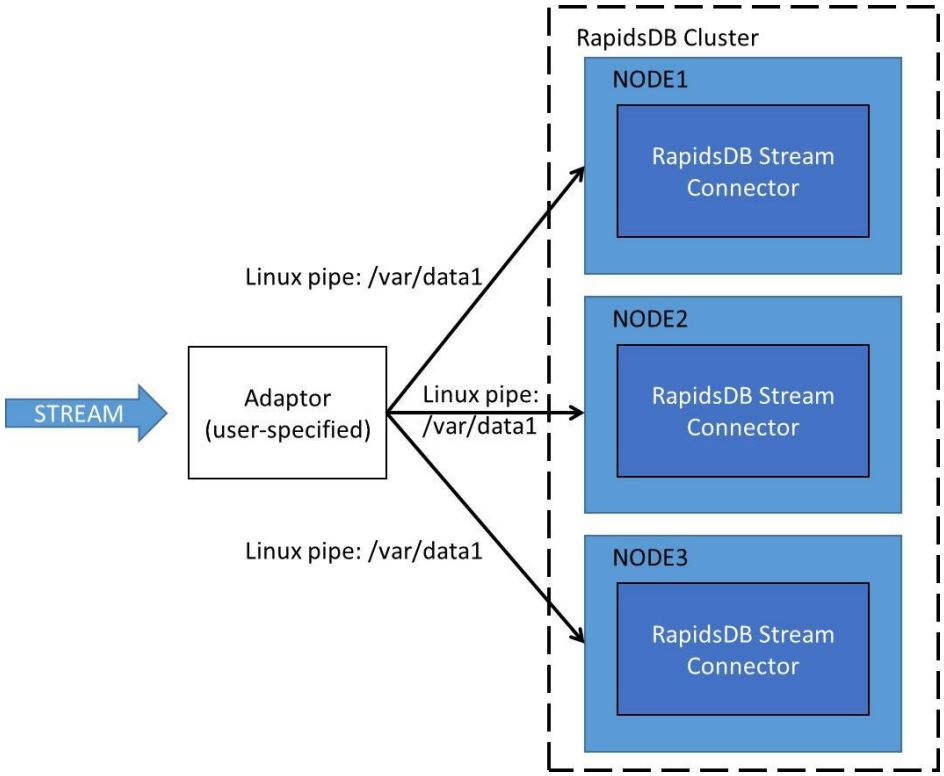


The following is the sample command for creating the Stream Connector shown above:

```
CREATE CONNECTOR STREAMS TYPE STREAM
NODE NODE1
CATALOG *
SCHEMA *
TABLE SENSOR1 WITH datapipe='/home/stdata/sensor1'
        USING ( col1  varchar,
                col2  timestamp)
TABLE SENSOR2 WITH datapipe='/home/stdata/sensor2'
        USING ( col1  varchar,
                col2  timestamp);
```

2.5.3 Multi-Node Support

A RapidsDB Stream Connector can also be configured to run on multiple nodes in a RapidsDB Cluster and to ingest a single stream across all of the nodes. In this case, each Stream Connector will ingest a portion of the stream and then the portions of the stream will be combined back to a single stream when the stream is queried. The source data stream Adaptor (see 2.1) is responsible for splitting up the source data stream into sub-streams (eg using round-robin) and delivering each sub-stream to the Stream Connector on each node via the configured Linux pipe. The diagram below shows a single stream being split across 3 Stream Connectors, with each part of the stream being delivered using a Linux named pipe named /var/data1:



The following is the sample command for creating the Stream Connector shown above:

```
CREATE CONNECTOR STREAM1 TYPE STREAM
NODE NODE1 NODE NODE2 NODE NODE3
CATALOG *
SCHEMA *
TABLE SENSOR1 WITH datapipe='/var/data1'
        USING ( col1  varchar,
                col2  timestamp);
```

2.6 Querying Streams

Only a single query can be run against a stream table at any time. Concurrent queries are not supported in this release. If concurrent queries are executed against the same stream table, then each query will get a subset of the data and the query will return incorrect results. The user can treat the stream table the same as any other table in the system and as such can use any SELECT query against that stream table including joins.

2.6.1 Stream Table Naming

As was described earlier (see 2.3), when configuring a Stream Connector the user provides a table name to be associated with the stream. As for all tables in RapidsDB, stream tables are identified by a 3-part name, which for a Stream Connector will be stream.public.<table name>. When querying a stream table, the catalog (stream) and schema (public) names are optional and are only required to disambiguate the table name.

2.6.2 Static Query

When querying a stream table, if the stream has not been activated, then the query will behave as though the stream table has no data. Once the data stream has been started the query will then return data. As described earlier (see 2.2), a query against a stream table will operate on the data read during the current time window, and the query will not return until the time window has expired. The time window starts when the query is submitted.

Example:

```
rapids>  select count(*) from st_lineitem join orders on l_orderkey = o_orderkey;
            ?1
            --
           277

          1 row(s) returned
rapids >
```

2.6.3 Continuous Query

Using the rapids-shell, the user can also submit what is called a "Continuous Query" and in this case the query will be executed continuously reporting the query result every time the time window completes. To specify a Continuous Query the user must prefix the query with the keyword "stream:". Use Ctl^c to terminate a Continuous Query.

Example:

```
rapids>  stream: select count(*) from st_lineitem join orders on l_orderkey = o_orderkey;
          ?1
          --
         277

        1 row(s) returned

          ?1
          --
         292

        1 row(s) returned
```

2.6.4 Creating New Streams From Query Results

It is possible to generate a new stream using the results of a query using the rapids-shell. This would normally be done using the results from a continuous query. The syntax for generating the new stream is shown below:

INTO PIPE <pipe>: [stream:] SELECT <select query>;

where, <pipe> is the fully qualified path name for the Linux named pipe that has been configured for a Stream Connector for the newly created stream. In this case the rapids-shell must be running on the same RapidsDB node that the target stream was configured for.

Below is a simple example that demonstrates how to create new streams from existing streams. In this example the input stream table is named ST_LINEITEM, and from that stream a new stream table named ST_AIR will be created which will include all of the rows from the ST_LINEITEM stream where the column l_shipmode is set to 'AIR'.

In this example we have created two Stream Connectors named STREAM1 and STREAM2, with STREAM1 handling the input stream for the ST_LINEITEM table and STREAM2 handling the stream that was generated from the ST_LINEITEM table. It should be noted that we could also have created one Stream Connector to handle both streams. Below are the commands to create the two Stream Connectors (see Section 2.5 for more details on creating Stream Connectors):

```
CREATE CONNECTOR STREAM1 TYPE STREAM NODE NODE2
CATALOG STREAM
  SCHEMA PUBLIC
        TABLE ST_LINEITEM WITH DATAPIPE='/home/rapids/davec/streamdata/st_lineitemData'
            USING ( L_ORDERKEY INTEGER, L_PARTKEY INTEGER, L_SUPPKEY INTEGER,
            L_LINENUMBER INTEGER, L_QUANTITY INTEGER, L_EXTENDEDPRICE FLOAT,
            L_DISCOUNT DECIMAL, L_TAX DECIMAL, L_RETURNFLAG VARCHAR, L_LINESTATUS
            VARCHAR, L_SHIPDATE TIMESTAMP, L_COMMITDATE TIMESTAMP, L_RECEIPTDATE
            TIMESTAMP, L_SHIPINSTRUCT VARCHAR, L_SHIPMODE VARCHAR, L_COMMENT
            VARCHAR ) ;

CREATE CONNECTOR STREAM2 TYPE STREAM NODE NODE2
CATALOG STREAM
  SCHEMA PUBLIC
        TABLE ST_AIR WITH DATAPIPE='/home/rapids/davec/streamdata/st_air'
            USING ( L_ORDERKEY INTEGER, L_PARTKEY INTEGER, L_SUPPKEY INTEGER,
            L_LINENUMBER INTEGER, L_QUANTITY INTEGER, L_EXTENDEDPRICE FLOAT,
            L_DISCOUNT DECIMAL, L_TAX DECIMAL, L_RETURNFLAG VARCHAR, L_LINESTATUS
            VARCHAR, L_SHIPDATE VARCHAR, L_COMMITDATE VARCHAR, L_RECEIPTDATE
            VARCHAR, L_SHIPINSTRUCT VARCHAR, L_SHIPMODE VARCHAR, L_COMMENT VARCHAR
            ) ;
```

Below are the queries that were run from the rapids-shell to generate the new ST_AIR table and then query that table. The row counts reflect the number of rows that satisfy the predicate "l_shipmode – 'AIR'" every second (the default window size). It is assumed that data is being written to the Linux pipe for STREAM1, '/home/rapids/davec/streamdata/st_lineitemData'.

```
rapids > into pipe /home/rapids/davec/streamdata/st_air: stream: select * from st_lineitem where l_shipmode = 'AIR';
11,212 row(s) written into pipes /home/rapids/davec/streamdata/st_air ...
11,484 row(s) written into pipes /home/rapids/davec/streamdata/st_air ...
11,461 row(s) written into pipes /home/rapids/davec/streamdata/st_air ...
11,059 row(s) written into pipes /home/rapids/davec/streamdata/st_air ...
11,474 row(s) written into pipes /home/rapids/davec/streamdata/st_air ...
11,505 row(s) written into pipes /home/rapids/davec/streamdata/st_air ...
11,073 row(s) written into pipes /home/rapids/davec/streamdata/st_air ...
11,645 row(s) written into pipes /home/rapids/davec/streamdata/st_air ...
11,138 row(s) written into pipes /home/rapids/davec/streamdata/st_air ...
11,426 row(s) written into pipes /home/rapids/davec/streamdata/st_air ...
11,534 row(s) written into pipes /home/rapids/davec/streamdata/st_air ...
11,425 row(s) written into pipes /home/rapids/davec/streamdata/st_air ...
14,923 row(s) written into pipes /home/rapids/davec/streamdata/st_air ...
11,695 row(s) written into pipes /home/rapids/davec/streamdata/st_air ...
```

```
rapids >stream: select count(*) from st_air;

1 row(s) returned
   ?1
   --
 11212

1 row(s) returned
   ?1
   --
 11484

1 row(s) returned
   ?1
   --
 11461

1 row(s) returned
   ?1
   --
 11059
```