STREAM: The Stanford Stream Data Manager

User Guide and Design Document

Contents

1 Introduction **3**

2 Functionality **3**
2.1 Output . . . 3
2.2 CQL Restrictions . . . 4
2.3 Named Queries (Views) . . . 5

3 Installation **6**
3.1 Building the system . . . 6
3.2 Testing the system . . . 6

4 Using STREAM **7**
4.1 gen_client . . . 7
4.2 STREAM library . . . 7
4.3 Configuring the STREAM server . . . 8
4.4 Script files . . . 8
4.4.1 Examples . . . 8
4.4.2 Input Streams and Relations . . . 8
4.4.3 Source files . . . 9
4.4.4 Queries . . . 9

5 Interface **11**
5.1 `Server` class . . . 11
5.2 `TableSource` class . . . 14
5.2.1 Input Tuple Encoding . . . 15
5.3 `QueryOutput` class . . . 16

6 STREAM Architecture **18**
6.1 Planning Subsystem . . . 18
6.1.1 Parser . . . 18
6.1.2 Semantic Interpreter . . . 18
6.1.3 Logical Plan Generator . . . 19
6.1.4 Physical Plan Generator . . . 19
6.1.5 Plan Manager . . . 20
6.1.6 Table Manager . . . 20
6.1.7 Query Manager . . . 20
6.2 Execution Subsystem . . . 21
6.2.1 Data . . . 21
6.2.2 Low-Level Operational Units . . . 23
6.2.3 High-Level Operational Units . . . 23
6.2.4 Global Operational Units . . . 27

1 Introduction

This document describes the architecture and design of a general purpose prototype *Data Stream Management System (DSMS)* called STREAM, for *STanford stReam datA Manager*. A more comprehensive introduction to DSMSes and the motivation for building one can be found in [MW+03].

2 Functionality

STREAM supports declarative *continuous queries* over two types of inputs: *streams* and *relations*. A continuous query is simply a long-running query, which produces output in a continuous fashion as the input arrives. The queries are expressed in a language called *CQL*, which is described in [ABW03]. The input types—streams and relations—are defined using some ordered *time domain*, which may or may not be related to wall-clock time.

Definition 2.1 (**Stream**) A *stream* is a sequence of timestamped tuples. There could be more than one tuple with the same timestamp. The tuples of an input stream are required to arrive at the system in the order of increasing timestamps. A stream has an associated schema consisting of a set of named attributes, and all tuples of the stream conform to the schema. □

Definition 2.2 (**Relation**) A relation is time-varying bag of tuples. Here "time" refers to an instant in the time domain. Input relations are presented to the system as a sequence of timestamped updates which capture how the relation changes over time. An update is either a tuple insertion or a tuple deletion. The updates are required to arrive at the system in the order of increasing timestamps. Like streams, relations have a fixed schema to which all tuples conform. □

Note that the timestamp ordering requirement is specific to one stream or a relation. For example, tuples of different streams could be arbitrarily interleaved.

2.1 Output

The output of a CQL query is a stream or relation depending on the query. The output is produced in a continuous fashion as described below:

- If the output is a stream, the tuples of the stream are produced in the order of increasing timestamps. The tuples with timestamp τ are produced once all the input stream tuples and relation updates with timestamps $\leq \tau$ have arrived.
- If the output is a relation, the relation is represented as a sequence of timestamped updates (just like the input relations). The updates are produced in the order of increasing timestamps, and updates with timestamp τ are produced once all input stream tuples and relation updates with timestamps $\leq \tau$ have arrived.

Some additional clarifications:

1. Note that the system can infer that all the tuples of a stream (resp. updates of a relation) with timestamps $\leq \tau$ have arrived only when the first tuple (resp. first update) with timestamp $> \tau$ arrives. So the output tuples with timestamp τ are produced only when at least one tuple or update with timestamp $> \tau$ has arrived on every input stream and relation.

2. Currently there is no way of providing information (called *heartbeats* in [ABW03, SW04, TMSF03]) to the system about the progress of time in input streams and relations. This could cause problems for applications with an input stream that is mostly "silent".

3. The representation of a relation as a sequence of timestamped updates is not unique, since tuple insertions and tuple deletions with the same timestamp and the same tuple value cancel each other. The system does not specify which of the several possible representations of a relation is produced.

2.2 CQL Restrictions

STREAM currently does not support all the features of CQL specified in [ABW03]. In this section, we mention the important features omitted in the current implementation of STREAM. In the next section, we describe how queries which require these features can be specified in an alternate fashion using *named intermediate queries* (or *views*). The important omissions are:

1. Subqueries are not allowed in the `Where` clause. For example the following query is not supported:

```
Select *
From S
Where S.A in (Select R.A
              From R)
```

2. The `Having` clause is not supported, but `Group By` clause is supported. For example, the following query is not supported:

```
Select A, SUM(B)
From S
Group By A
Having MAX(B) > 50
```

3. Expressions in the `Project` clause involving aggregations are not supported. For example, the query:

```
Select A, (MAX(B) + MIN(B))/2
From S
Group By A
```

   is not supported. However, non-aggregated attributes can participate in arbitrary arithmetic expressions in the project clause and the where clause. For example, the following query is supported:

```
Select (A + B)/2
From   S
Where  (A - B) * (A - B) > 25
```

4. Attributes can have one of four types: `Integer`, `Float`, `Char(n)`, and `Byte`. Variable length strings (`Varchar(n)`) are not supported. We do not currently support any casting from one type to another.

5. Windows with the *slide* parameter are not supported.

6. The binary operations `Union` and `Except` is supported, but `Intersect` is not.

2.3 Named Queries (Views)

A CQL query can be assigned a name and a schema to allow its result to be referenced by other queries. This feature allows us to express some of the esoteric, omitted features of CQL mentioned in Section 2.2. The following query is an example of a CQL view (it produces a relation):

```
AggS (A, Sum_B, Max_B):

  Select A, SUM(B), MAX(B)
  From S
  Group By A
```

It can be used in a different query just like an input relation:

```
  Select A, Sum_B
  From AggS
  Where Max_B > 50
```

Note that the combination of these two queries produces the same output as the query with a `Having` clause that we mentioned in Section 2.2 (item # 2).

3 Installation

The latest version of the code can be downloaded from `http://www-db.stanford.edu/stream/code/`. The code has been packaged using the standard GNU tools, so the usual technique for installing such packages works. We assume, for illustration, that the code will be extracted, built, and installed in the following set of directories:

- Directory where the code is extracted: `/home/user/stream-0.5.0`
- Directory where the binaries are installed: `/home/user/stream/bin`
- Directory where the libraries are installed: `/home/user/stream/lib`
- Directory where the headers are installed: `/home/user/stream/include`

3.1 Building the system

1. `cd /home/user/stream-0.5.0/`
2. `./configure --bindir=/home/user/stream/bin/ --libdir=/home/user/stream/lib/ \ --includedir=/home/user/stream/include`
3. `make`
4. `make install`

These four steps generate a command-line client `gen_client` at `/home/user/stream/bin`, the stream library at `/home/user/stream/lib` and the header files for use with the library at `/home/user/stream/include`. Details of using the command-line interface and the library are provided in Section 4.

3.2 Testing the system

We have provided a collection of test scripts to check if the system has been built properly. To run the tests:

1. `cd home/user/stream-0.5.0/test`
2. `./test.sh`

To remove the temporary files produced while testing:

1. `cd home/user/stream-0.5.0/test`
2. `./cleanup.sh`

4 Using STREAM

Currently, there are two ways of using STREAM:

1. Using the command-line client `gen_client`.
2. Linking the STREAM library directly into your C++ application.

In the next release, we are planning to include a standalone server that talks to clients over the network and a GUI client.

4.1 gen_client

`gen_client` dynamically loads the STREAM library, so `LD_LIBRARY_PATH` should be set to include the path to the STREAM library as follows:

```
export LD_LIBRARY_PATH=/home/user/stream/lib/:\$LD_LIBRARY_PATH
```

The usage syntax of the `gen_client` program is:

```
gen_client -l [log-file] -c [config-file] [script-file]
```

`[log-file]` is the output file where the execution log of the program is written. `[config-file]` is an input file which specifies values of various server configuration parameters (e.g., available memory, duration of execution). Finally `[script-file]` is an input file which contains the queries to be executed and the streams and relations involved in the queries. All three arguments `[log-file]`, `[config-file]`, and `[script-file]` are necessary.

We have include examples of script files and configuration files in `/home/user/stream-0.5.0/examples`. You can run the example script files as follows:

1. `export PATH=$PATH:/home/user/stream/bin/`
2. `cd /home/user/stream-0.5.0`
3. `gen_client -l log -c examples/config examples/scripts/script1`

(The example scripts assume that the `gen_client` program is being run from `/home/user/stream-0.5.0` directory; running the program from a different location will cause an error.) Details of writing script files and configuration files can be found in Section 4.4 and Section 4.3 respectively.

4.2 STREAM library

The reader can look at the source of `gen_client` (`/home/user/stream-0.5.0/gen_client/generic_client.cc`), which is less than 150 lines of code, for an illustration of using the STREAM library. Briefly, the steps involved in using the STREAM as an embedded library are within an application are:

1. Create a new `Server` object.
2. Configure the server by providing a configuration file.
3. Register the input streams and relations.

4. Register the queries (named or unnamed).

5. Generate a query plan.

6. Start the server.

The details of the `Server` interface that accomplish these tasks is discussed in Section 5. In order to input a stream or a relation, the application passes an object implementing the `TableSource` interface (`<interface/table_source.h>`) to the server at the time of registering the stream or a relation. Similarly, in order to get the output of a query, the application passes an object implementing the `QueryOutput` interface (`<interface/query_output.h>`) at the time of registering the query.

4.3 Configuring the STREAM server

See `/home/user/stream-0.5.0/examples/config` for an example configuration file. The file also explains the meaning of all the configuration parameters, and provides reasonable default values. If you are unsure about what configuration parameters to use, you can use an empty (!) configuration file, and the system will configure itself with default values. But note that the default value of memory size is 32MB.

4.4 Script files

This section describes the details of writing script files input to the `gen_client` program. The script file is used to specify:

1. The input streams and relations and their file locations, and

2. One or more queries over the streams and relations.

White spaces can be freely added anywhere in a script file. Also lines starting with # are comment-lines and are not interpreted by the `gen_client` program.

4.4.1 Examples

Example script files can be found in `/home/user/stream-0.5.0/examples/scripts/` and `/home/user/stream-0.5.0/test/scripts/`. These scripts are probably more useful than the (semi-) formal descriptions below.

4.4.2 Input Streams and Relations

An input stream and a relation (generically called a table) is specified using two lines. The first line specifies the schema and the second the source file. The following two lines specify that `S` is a stream with 4 attributes `A, B, C,` and `D` of types `integer`, `byte`, `float` and `char(n)`, respectively, and that the source of the stream is the file S.dat.

```
table : register stream S (A integer, B byte, C float, C char(4));

source : /home/user/stream-0.5.0/examples/data/S.dat
```

(Note the semi-colon at the end of the first line.) Example of a specifying a relation:

```
table : register relation R (A integer, B byte, C float, C char(4));

source : /home/arvind/stream/examples/data/R.dat
```

The two lines that specify a table (stream or relation) should appear consecutively in the script file after ignoring empty lines and comment lines.

4.4.3 Source files

See `/home/user/stream-0.5.0/examples/data/*.dat` for examples of source files. The first line of the source file contains the augmented schema of the stream or relation. For a stream, the first attribute in the augmented schema is the timestamp (integer) attribute. The remaining attributes are the data attributes that are specified in the `register stream ...` script-line. For example, the schema line for the stream `S` above looks like:

```
i,i,b,f,c4
```

Here `'i'` stands for integer, `'b'` for byte, `'f'` for float, and `'c4'` for `char(4)`. Note that there are two `'i'`s. The first `'i'` corresponds to the timestamp attribute, and the second to the data attribute `A` of `S`.

For a relation, the first attribute in the augmented schema is again a timestamp (integer). The second attribute is a 'sign' (byte) attribute. For example, the schema line for the relation R looks like;

```
i,b,i,b,f,c4
```

All the remaining lines of the source files encode data tuples, one per line. The attribute values of a tuple are comma separated. For streams, the first attribute is the timestamp attribute, and this has to be non-decreasing. For relation, the first attribute is the non-decreasing timestamp, and the second is the sign attribute. The sign takes only two values `'+'` and `'-'`. A `'+'` indicates that the tuple was inserted into the relation at the specified timestamp and a `'-'` indicates that the tuple was deleted from the relation at the specified timestamp. For example, the line `1,+,1,1.1,a,abc` indicates that the tuple $\langle 1, 1.1, a, abc \rangle$ was inserted into relation at time 1. A `'-'` tuple should correspond to some preceding `'+'` tuple.

4.4.4 Queries

As described in Section 2, queries are specified in CQL. Example CQL queries can be found in `/home/user/stream-0.5.0/examples/cql-queries`.

Recall from Section 2 that queries can be named or unnamed. An unnamed query produces an external output, while a named query produces an internal output that can be referenced by other queries. The two lines:

```
query : select * from S;

dest : outfile
```

specify an unnamed query whose output is written to file `outfile`. The query refers to a stream `S`, which could be an input stream or the output of a named query. The following two lines illustrate named queries:

```
vquery : select A, B from S Where A = 5;

vtable : register stream FilteredS (A integer, B byte);
```

This query filters stream `S` on attribute `A` and projects out attributes `A` and `B`. The second line assigns names to this query and its attributes. `FilteredS` can be referenced in other named/unnamed queries. Example:

```
query : select * from FilteredS;

dest : ...
```

All names should be "registered" before they are referenced. For example, the `register stream FilteredS ...` line should occur before the `select * from FilteredS` line. Also, currently, it is an error to specify a named query that is not used by any other query.

5 Interface

This section describes the three classes:

1. `Server`
2. `TableSource`
3. `QueryOutput`

which constitute the external interface to STREAM.

5.1 `Server` class

Synopsis

```
#include <stream/interface/server.h>

class Server {
public:
  static Server *newServer (std::ostream &);
  int setConfigFile (...);
  int beginAppSpecification();
  int registerBaseTable(...);
  int registerQuery(...);
  int registerView(...);
  int endAppSpecification();
  int beginExecution();
};
```

Description

The STREAM server interface. It contains methods to configure the server, register new streams and relations, register new continuous queries, and run the continuous queries. The server operates in two phases: in the first phase all the registering (of CQ, streams, and relations) is done, and in the second phase the continuous queries registered in the first phase are executed. The reason for separating out these two phases is performance: this design lets us perform some optimizations across queries before generating the final plan.

Member functions

1. `static Server* newServer(std::ostream &log)`

   Construct and return a new STREAM server.

   Parameters

   `log`: The `log` parameter specifies the output stream to which the server log entries are written.

2. `int setConfigFile (const char *configFile)`

   Specify the location of the configuration file. Section 4.3 gives details of configuration files. This method should be called before any of the other methods of `Server` class.

Parameters

`configFile`: The location of the configuration file.

Returns

zero on success, non-zero on error.

3. `int beginAppSpecification()`

   This method is called before the actual specification of an application (which is done using the `registerQuery()` and `registerTable()` methods).

   Returns

   zero on success, non-zero on error.

4. ```
   int registerBaseTable (const char *tableInfo,
                          unsigned int tableInfoLen,
                          Interface::TableSource *input)
   ```

   Register an input stream or a relation (generically called a table) with the server by specifying (1) the name-related information about the table and (2) the `TableSource` object that provides the input tuples. The name-related information consists of the table type (stream or a relation), the table name and the names of the attributes of the table. An input table should be registered before it is referenced in any query.

   Parameters

   `tableInfo`: String encoding the name-related information about the table. For example, the string "`register stream S (A integer, B char(4))`" is used to register a stream `S` with two attributes `A` and `B`. Similarly, the string "`register relation R (A integer, C float)`" is used to register a relation `R` with two attributes `A` and `C`.

   `tableInfoLen`: The length of `tableInfo` parameter.

   `input`: The `TableSource` object that provides the input data tuples of the stream or relation.

   Returns

   zero on success, non-zero on error.

5. ```
   int registerQuery(const char *querySpec,
                     unsigned int querySpecLen,
                     Interface::QueryOutput *output,
                     unsigned int &queryId)
   ```

   Register a query (named or unnamed) with the server. If the output of the query is desired, then the application should pass an implementation of `QueryOutput` interface. Note that the output of the query may not always be required externally—a query could be registered with the sole purpose of providing input to other queries. In this case, the query is a named query (see Section 2.3), and the naming information of the query output should be specified using the `registerView` method.

   The method returns a `queryId` parameter which is used to reference this query in the `registerView` method.

   Parameters

`querySpec`: The specification of the query in CQL.

`querySpecLen`: Length of the query specification.

`output`: An object that implements the `QueryOutput` interface if the output of the query is required by the application, or 0 if the output is not required.

`queryId`: Output parameter set by the method on termination.

Returns

zero on success, non-zero on error.

6. `int registerView(unsigned int queryId,`
   `const char *tableInfo,`
   `unsigned int tableInfoLen)`

   Register a name to the output of a previously registered query so that the output can be referenced in other future queries. The name-relation information consists of the table type (stream or a relation), the table name and the names of the attributes of the table (just like in the `registerBaseTableMethod`).

   Parameters

   `queryId`: The identifier for the query whose output is being named. The identifier is the value returned by the `registerQuery` method when the query was registered.

   `tableInfo`: String encoding the name-related information of the query output. Same syntax as `tableInfo` parameter of `registerBaseTable` method.

   `tableInfoLen`: The length of the `tableInfo` parameter.

   Returns

   zero on success, non-zero on error.

7. `int endAppSpecification()`

   This method is called after all the queries and inputs have been registered, and after this method call no future queries and inputs can be registered.

8. `int beginExecution()`

   Begin the execution of the continuous queries registered earlier. The duration of execution is specified within the configuration file.

5.2 `TableSource` class

Synopsis

```
#include <stream/interface/table_source.h>

class TableSource {
public:
  int start();
  int getNext(...);
  int end();
};
```

Description

The interface that the server uses to get input stream or relation tuples. The application should provide an object that implements this interface along with each input stream and relation (see `Server::registerQuery()` method). Different inputs could have different implementations, e.g., one reading from a file, and the other from a network.

The main method is `getNext()`, which the server uses to pull the next tuple of the stream or relation whenever it desires. Before consuming any input tuples, the server invokes the `start()` method, which can be used to perform various kinds of initializations. Similarly, the `end()` method is called when the server is not going to invoke any more `getNext()`s.

Member functions

1. `int start()`

   This method is invoked by the server (exactly) once before any `getNext()` calls are invoked.

2. `int getNext(char *& tuple, unsigned int& len)`

   This method is invoked by the server to input the next tuple of the stream or relation that the `TableSource` object handles. If the next tuple is available, on return the parameter `tuple` should point to the encoding of the next tuple. Otherwise, the parameter `tuple` should point to `null` (0). The memory for the location pointed to by `tuple` (if it is non-null) is allocated and owned by the `TableSource` object, which means that the server does not deallocate the memory. The `TableSource` object should ensure that the contents of this memory location is unchanged until the next `getNext()` call.

   Setting the `tuple` parameter to `null` only indicates that there is no input tuple availabe currently. In particular, it does not signify the end of the stream. An input tuple might be available later, and the server keeps polling periodically.

   Details of encoding the input tuples are described in Section 5.2.1.

   Parameters

   `tuple`: On returning points to the encoding of the input tuple if the next input tuple is available; 0 otherwise.

   `len`: Set by the function to the length of the tuple encoding.

   Returns

zero on success, non-zero on error.

3. `int end()`

   Called by the server to indicate that it is not going to invoke any more `getNext()` calls. Currently, never invoked.

5.2.1 Input Tuple Encoding

A stream tuple consists of a timestamp and the values for the data attributes. A relation tuple consists of a timestamp, a sign, and values for the data attributes, which represents a timestamped update to the relation as indicated in Definition 2.2. The sign is either a `+`, which indicates an insertion, or a `-`, which indicates a deletion.

The encoding of a stream tuple contains the encoding of the timestamp followed by the encoding of the attribute values. Attribute values are encoded in the order in which they are declared in the stream schema. Timestamps are encoded as `unsigned int`s, integer attributes as `int`s, floating point values as `float`s, byte values as `char`s, and `char(n)` values as a sequences of n `char`s. For example,

```
unsigned int timestamp;
int a_val;
float b_val;

// Get attribute values
// ...

// Encode a tuple of stream S (A integer, B float, ...)
memcpy (tupleBuf, &timestamp, sizeof(unsigned int));
memcpy (tupleBuf + sizeof(unsigned int), &a_val, sizeof(int));
memcpy (tupleBuf + sizeof(unsigned int) + sizeof(int), &b_val, sizeof(float));

//  ...

// Set tuple to point to tupleBuf
tuple = tupleBuf;
```

The encoding of a relation tuple contains the encoding of the timestamp followed by the encoding of the sign followed by the encoding of the attribute values (in the order in which they appear in the schema). Timestamps and attribute values are exactly as in the case of stream tuples. The sign is encoded using a single `char`.

5.3 `QueryOutput` class

Synopsis

```
#include <stream/interface/query_output.h>

class QueryOutput {
public:
  int setNumAttrs(...);
  int setAttrInfo(...);
  int start();
  int putNext(...);
  int end();
};
```

Description

Interface that the server uses to produce query output. An application should provide an object that implements this interface for each query whose output it desires (see `Server::registerQuery()` method). Different queries could have different implementations, e.g., the output of one could go to a file, the other to a remote location on the network.

The main method is the `putNext()` method which is used by the server to push out the next output tuple of the query. Before pushing any tuples to the output, the server first indicates the schema of the output tuples using `setNumAttrs()` and `setAttrInfo()` method calls. The method `start()` is called before the first `putNext()` call: this can be used by the object to perform various initializations, resources allocation etc.

Member functions

1. `int setNumAttrs(unsigned int numAttrs);`

   Set the number of attributes in the output schema of the query. This method is called (exactly once) by the server before any other method.

   Parameters

   `numAttrs`: the number of attributes in the output schema of the query.

   Returns

   zero on success, non-zero on error.

2. ```
   virtual int setAttrInfo(unsigned int attrPos,
                           Type attrType,
                           unsigned int attrLen)
   ```

   The server calls this method to specify the type information of a particular attribute in the output schema of the query. The server calls this method (once for each attribute in the schema) after the `setNumAttrs()` method and before the `start()` method.

   Parameters

   `attrPos`: The position of the attribute for which the type info is being specified. This is a value between 0 and $numAttrs - 1$, where $numAttrs$ is the number of attributes in the output schema of the query.
   ```

`attrType`: The type of the attribute. `Type` is an `enum` defined in `<stream/common/types.h>`.

```
enum Type {
  INT,                // integer
  FLOAT,              // floating point
  BYTE,               // 1 byte characters
  CHAR                // fixed length strings.
};
```

`attrLen`:

3. `virtual int start()`

   Called by the server to signal that it is ready to invoke the `putNext()` calls. The object can use this method to perform various initializations.

4. `int putNext(const char *tuple, unsigned int len)`

   Called by the server to push the next output tuple of the query.

   Parameters

   `tuple`: Encoding of the next output tuple. The memory of the location pointed by `tuple` is allocated and owned by the server. The encoding is almost identical to the encoding of input tuples as described in Section 5.2.1. The only difference is that there exists a "sign" column irrespective of whether the output of the query is a stream or a relation. The sign column is always `+` if the output of the query is a stream while it could be `+` or `-` if the output of the query is a relation.

   Returns

   zero on success, non-zero on error.

5. `int end()`

   Called by the server to indicate that it is not going to invoke any more `putNext()` calls. Currently, never invoked.

6 STREAM Architecture

This section briefly describes the architecture of the STREAM DSMS prototype. The STREAM architecture is made up of two broad components:

1. *Planning subsystem*, which stores metadata and generates query plans, and
2. *Execution engine*, which executes the continuous queries.

6.1 Planning Subsystem

Figure 6.1 shows the main components of the planning subsystem. The components shown with double-bordered rectangles are stateful—they contain the system metadata. The other components are stateless, functional units, which are used to transform a query to its final plan. The solid arrows indicate the path of a query along these components.

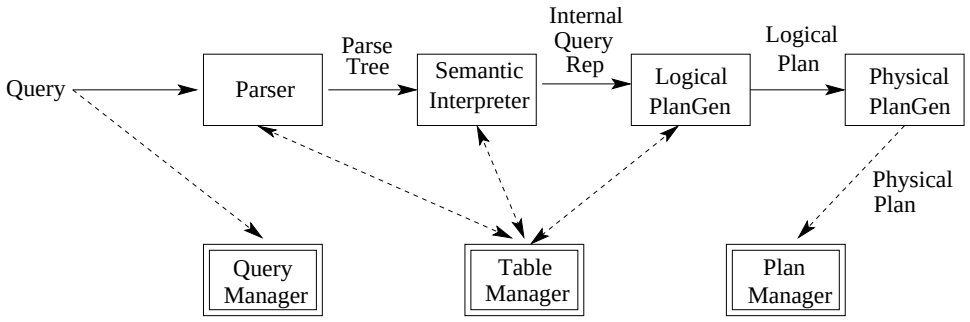

Figure 1: The planning component

6.1.1 Parser

- *Functionality:* Transform the query string to a parse tree representation of the query. (The parser is also used to parse the schema of a registered stream or relation.)
- *Relevant Code:*
  - `dsms/include/parser/*`
  - `dsms/src/parser/*`

6.1.2 Semantic Interpreter

- *Functionality:* Transform the parse tree to an internal representation of the query. The representation is still block-based (declarative?) and not an operator-tree. As part of this transformation, the semantic interpreter:
  - Resolves attribute references

  - Implements CQL defaults (e.g., adding an `Unbounded` window; see [ABW03].)
  - Other misc. syntactic transformations like expanding the "`*`" in `Select *`
  - Converts external string-based identifiers for relations, streams, and attributes to internal integer-based ones. The mapping from string identifiers to integers identifiers is maintained by `TableManager`.

- *Relevant Code:*
  - `dsms/include/querygen/query.h`
  - `dsms/include/querygen/sem_interp.h`
  - `dsms/src/querygen/sem_interp.cc`

6.1.3 Logical Plan Generator

- *Functionality:* Transform the internal representation of a query to a logical plan for the query. The logical plan is constructed from *logical operators*. The logical operators closely resemble the relational algebra operators (e.g., select, project, join), but some are CQL-specific (e.g., window operators and relation-to-stream operators). The logical operators are not necessarily related to the actual operators present in the execution subsystem. The logical plan generator also applies various transformations that (usually) improve the performance:
  - Push selections below cross-products (joins).
  - Eliminate redundant `Istream` operators (an `Istream` over a stream is redundant).
  - Eliminate redundant project operators (e.g., a project operator in a `Select *` query is usually redundant).
  - Apply `Rstream-Now` window based transformations.

- *Relevant Code:*
  - `dsms/include/querygen/log*.h`
  - `dsms/src/querygen/log*.cc`

6.1.4 Physical Plan Generator

- *Funcionality:* Transform a logical plan for a query to a physical plan. The operators in a physical plan are exactly those that are available in the execution subsystem (unlike those in the logical plan). (We have a separate logical plan stage in query generation because it is easier to apply transformations to logical plans than to physical plans. Logical operators are more abstract and easier to deal with than physical operators which have all sorts of associated low level details.)

  The physical plan generator is actually part of the plan manager (although this is not suggested by Figure 6.1), and the generated physical plan for a query is linked to the physical plans for previously registered queries. In particular, the physical plans for views that are referenced by the query now directly feed into the physical plan for the query.

- *Relevant Code:*
  - `dsms/include/metadata/phy_op.h`
  - `dsms/src/metadata/gen_phy_plan.cc`

6.1.5 Plan Manager

- *Functionality:* The plan Manager stores the combined "mega" physical plan corresponding to all the registered queries. The plan manager also contains the routines that:
  - Flesh out a basic physical plan containing operators with all the subsidiary execution structures like synopses, stores, storage allocators, indexes, and queues.
  - Instantiate the physical plan before starting execution.
- *Relevant Code:*
  - `dsms/include/metadata/plan_mgr.h`
  - `dsms/include/metadata/plan_mgr_impl.h`
  - `dsms/src/metadata/plan_mgr.cc`
  - `dsms/src/metadata/plan_mgr_impl.cc`
  - `dsms/src/metadata/inst_*.cc`
  - `dsms/src/plan_inst.cc`
  - `dsms/src/plan_store.cc`
  - `dsms/src/plan_syn.cc`
  - `dsms/src/plan_trans.cc`
  - `dsms/src/static_tuple_alloc.cc`
  - `dsms/src/tuple_layout.cc`

6.1.6 Table Manager

- *Functionality:* The table Manager stores the names and schema of all the registered streams and relation. The streams and relations could be either input (base) stream and relations or intermediate streams and relations produced by named queries. The table manager also assigns integer identifiers for streams and relations which are used in the rest of the planning subsystem.
- *Relevant Code:*
  - `dsms/include/metadata/table_mgr.h`
  - `dsms/src/metadata/table_mgr.cc`

6.1.7 Query Manager

- *Functionality:* The query manager stores the text of all the registered queries. Currently does not serve a very important role, but might be used in later versions for better error reporting.
- *Relevant Code:*
  - `dsms/include/metadata/query_mgr.h`
  - `dsms/src/metadata/query_mgr.cc`

6.2 Execution Subsystem

Figure 6.2 shows the main components of the STREAM execution engine and their interactions. Table 6.2 lists the different types of entities that exist in the execution engine, roughly classified based on their role and at the granularity that they function.

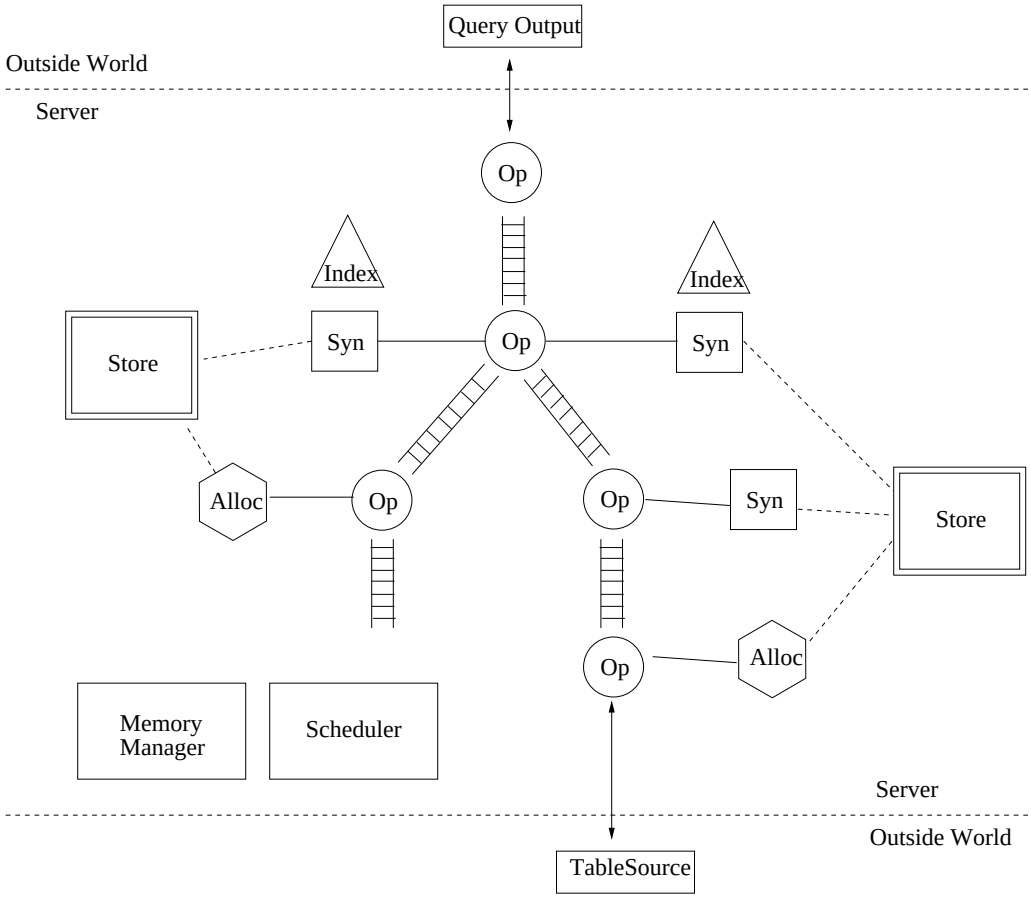

Figure 2: Architecture of STREAM Execution Subsystem. *Op* denotes operators, *Syn* denotes synopses, *Alloc* denotes storage allocators.

6.2.1 Data

- **Tuple:** A tuple is the basic unit of data. It is logically a collection of attribute values (as expected). In the implementation, a tuple is simply a pointer to a memory location (`char*`) where the attribute values are encoded. All the information required to consistently enter and extract attribute values from this memory location are present in the units that operate over the tuple (e.g., operators).

  Relevant Code: `dsms/include/execution/internals/tuple.h`

Data	Operational Units
• Low-level – Tuple – Element – Heartbeat • High-level – Stream – Relation	• Low-level – Arithmetic Evaluators – Boolean Evaluators – Hash Evaluators • High-level – Operators – Queues – Synopses – Indexes – Stores – Storaga Allocators • Global – Memory Manager – Scheduler

Table 1: Components of STREAM execution subsystem

- **Element:** An element is a tuple with a timestamp and a *sign*. Sign is either a + or a −. The interpretation of a sign should become clear from the description of a relation below.

  Relevant Code: `dsms/include/execution/queues/element.h`

- **Heartbeat:** A heartbeat is a special kind of element with just a timestamp and no tuple or sign associated. Heartbeats are used to communicate progress of time among operators; see [SW04] for technical details.

  Relevant Code: `dsms/include/execution/queues/element.h`

- **Relation:** A relation is a sequence of elements ordered by timestamps. Logically, such a sequence represents a time-varying bag of tuples, which is consistent with Definition 2.2. The bag of tuples in the relation at timestamp τ is obtained by inserting into the bag the tuples of all the elements with timestamp $\leq \tau$ having a + sign and deleting from the bag the tuples of all the elements with timestamp $\leq \tau$ having a − sign. Note that the representation of a relation is not unique.

  Relation produced by the STREAM operators (discussed in Section 6.2.3) satisfy the following property: for every − element in a relation sequence there exists a + element that occurs earlier in the sequence such that the − element and + element have identical tuples. By identical tuples, we mean tuples that point to the same memory location, and not tuples with identical attribute values. But two − elements cannot be mapped to the same + element in the above sense.

- **Stream:** A stream is a sequence of elements with a + sign, ordered by timestamps. Under this representation, a stream is a special kind of a relation with no − elements in its sequence.

6.2.2 Low-Level Operational Units

All direct operations over tuples are performed by objects known as *evaluators*. Each evaluator conceptually evaluates a fixed function or procedure over an input set of tuples. Different sets of input tuples can be "bound" to the evaluator before different evaluations. The evaluation can have "side-effects": the contents of one of the input tuples can be updated as a result of the evaluation. There are three types of evaluators:

- **Arithmetic evaluator:** Arithmetic evaluators evaluate simple arithmetic functions defined over the attributes of the input tuples. The result of the arithmetic functions is used to update the contents of one or more of the input tuples. For example, an arithmetic evaluator can take two input tuples, and set the attribute #1 of the second tuple to be the sum of attribute #1 and attribute #2 of the first tuple. Arithmetic evaluators can be used directly by operators (e.g., project operator) or within boolean evaluators to evaluate predicates involving arithmetic.

  Relevant Code:

  - `dsms/include/internals/aeval.h`
  - `dsms/include/internals/eval_context.h`
  - `dsms/src/internals/aeval.cc`

- **Boolean evaluator:** Boolean evaluators evaluate simple boolean predicates over the attributes of the input tuples. The boolean predicates can be conjunctions of comparisons, and the comparisons can involve arithmetic. The result of the evaluation (true/false) is returned as the output. For example, a boolean evaluator can take two tuples and return true if the attribute #1 of the first tuple is equal to attribute #2 of the second tuple.

  Relevant Code:

  - `dsms/include/internals/beval.h`
  - `dsms/include/internals/eval_context.h`
  - `dsms/src/internals/beval.cc`

- **Hash Evaluator:** Hash evaluators compute a hash function over a subset of the attributes of the attributes of the input tuples, and return a (currently 32 bit) hash value as output. Hash evaluators are currently used only within hash-based indexes.

  Relevant Code:

  - `dsms/include/internals/heval.h`
  - `dsms/include/internals/eval_context.h`
  - `dsms/src/internals/heval.cc`

6.2.3 High-Level Operational Units

- **Operators:** Operators are the basic processing units that operate over streams and relations: an operator takes one or more streams and produces a stream or a relation as output. Table 2 list the set of operators currently available in STREAM. Semantics of these operators are described in [ABW03].

  Each operator operates in a continuous fashion. Once an operator sees all its input elements with timestamp upto τ, it produces all its output elements with timestamp upto τ. Each operator produces its

Operator	Signature	Code
Binary Join	*Relation* × *Relation* → *Relation*	`bin_join.cc`
Binary Stream Join	*Stream* × *Relation* → *Stream*	`bin_str_join.cc`
Distinct (duplicate elimination)	*Relation* → *Relation*	`distinct.cc`
Dstream	*Relation* → *Stream*	`dstream.cc`
Except	*Relation* × *Relation* → *Relation*	`except.cc`
Group By Aggregation	*Relation* → *Relation*	`group_aggr.cc`
Istream	*Relation* → *Stream*	`istream.cc`
Partition Window	*Stream* → *Relation*	`partn_win.cc`
Projection	*Relation* → *Relation*	`project.cc`
Range Window	*Stream* → *Relation*	`range_win.cc`
Row Window	*Stream* → *Relation*	`row_win.cc`
Rstream	*Relation* → *Stream*	`rstream.cc`
Selection	*Relation* → *Relation*	`select.cc`
Union	*Relation* × *Relation* → *Relation*	`union.cc`

Table 2: List of data operators in STREAM. This list does not include the input and output operators which talk to the outside world. All the files in the Code column are in the directory `dsms/src/execution/operators/`

output elements in the order of increasing timestamps. Once an operator has produced all its output upto $\tau - 1$, it has the option of asserting that it will not generate any output element with timestamp $< \tau$ by generating a heartbeat with timestamp τ. This is useful especially if the output of an operator is sparse—e.g., a selection operator with a highly selective filter—to convey time-related information to the upstream operators.

Many operators need to maintain state. For example, the binary join operator needs to maintain the "current" bag of tuples in both its inner and outer relation. In STREAM, all the operator state that is not statically bounded in size is maintained in objects known as *synopses*. Synopses are discussed in more detail below.

Operators are connected to each other using intermediate element buffers called *queues*. Queues serve to (at least partially) decouple the running of one operator from another. An operator reads each of its inputs from one queue and writes its output to another queue.

A global scheduler schedules the operators of the system using some scheduling strategy. When scheduled, an operator runs for an amount of time specified by the scheduler, and then returns control back to the scheduler, which picks a different operator to run, and so on. An operator could *stall* while it is running if its output queue gets filled up. (Stalling occurs because all the queues currently have a fixed amount of memory.) A stalled operator maintains its current state to enable it to resume processing at a later time, and returns control back to the scheduler.

Relevant Code:

- `dsms/include/execution/operators/*.h`
- `dsms/src/execution/operators/*.cc`

- **Queues:** Conceptually, queues are FIFO buffers for elements. Elements are always inserted and read off from the queue in timestamp order. (This property is guaranteed by the operators using the queues.) The simplest queue has one operator that writes to it and one operator that dequeues from it. Sometimes

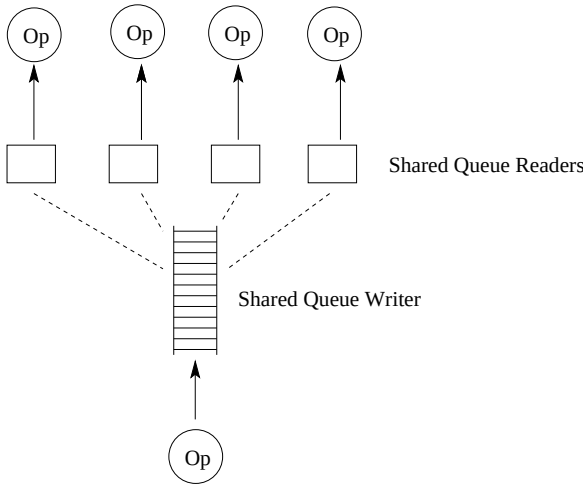

Figure 3: Configuration of a shared queue

the output of one operator is consumed by several (say $n > 1$) operators. In such cases the operators are interconnected by a *shared queue*. Such a shared queue is realized by using a collection of objects: 1 shared queue writer and n shared queue readers as shown in Figure 3. The shared queue readers contain operator specific state, while the shared queue writer serves a common store for all the currently active elements (all the elements which have been written by the source operator but not yet read by at least one sink operator). The operators themselves are oblivious to sharing.

Relevant Code:

- `dsms/include/execution/queues/*.h`
- `dsms/src/execution/queues/*.cc`

- **Synopses:** Synopses are objects that contain operator state. Each synopsis is owned by exactly one operator. Currently all the synopses contain bags of tuples. Also, all the tuples in a synopsis have the same schema, and as we will see, are allocated by the same storage allocator.

  There are four types of synopses, listed in Table 3, and they differ primarily in the interfaces that they export. The relation synopsis interface allows the owning operator to insert a tuple, delete an existing tuple, and scan the current bag of tuples based on some predicate. The window synopsis interface allows the owning operator to insert a tuple and delete/read the oldest tuple. The partition window synopsis is identical to the window synopsis, except that it allows the deletion/reading of the oldest tuple within a partition defined using certain tuple attributes. Finally, lineage synopsis is similar to relation synopsis, except that it allows the operator to specify a "lineage" along with each inserted tuple. The operator can later use the lineage to access the tuple.

  Relevant Code:

  - `dsms/include/execution/synopses/*.h`

Synopsis Type	Interface Methods
Relation Synopsis	`insertTuple(Tuple)`
	`deleteTuple(Tuple)`
	`scan()`
Window Synopsis	`insertTuple(Tuple)`
	`deleteOldestTuple()`
	`getOldestTuple(Tuple&)`
Partition Window Synopsis	`insertTuple(Tuple)`
	`deleteOldestTuple(Tuple t, Tuple partition)`
	`getOldestTuple(Tuple &t, Tuple partition)`
Lineage Synopsis	`insertTuple(Tuple, Tuple* lineage)`
	`deleteTuple(Tuple)`
	`scan()`
	`getTuple(Tuple&, Tuple *lineage)`

Table 3: Different Types of Synopses in STREAM

  - `dsms/src/execution/synopses/*.cc`

- **Indexes:** Synopses can internally contain an index to speed up scans. The existence of the index is oblivious to the operator that owns the synopsis. Currently the system only supports hash-based equality indexes.

  Relevant Code:

  - `dsms/include/execution/indexes/*.h`
  - `dsms/src/execution/indexes/*.cc`

- **Storage Allocators:** All tuples in the system are allocated by objects called storage allocators. Each storage allocator is owned by one operator, and is used to allocate tuples of the output elements of the operator. Not all operators own a storage allocator; for example, the select operator simply forwards its input tuples to its output and does not allocate new tuples. Storage allocators also keep track of tuple usage and reclaim the space of unused tuples.

  Relevant Code:

  - `dsms/include/execution/stores/store_alloc.h`

- **Stores:** The description of storage allocators and synopses above was focused mainly on the interface that they presented to the operators. Most of the actual logic of storage allocators and synopses are implemented within *stores*. (In fact, a storage allocator is a pure interface which is implemented by the store objects.) This level of indirection in the implementation was introduced to enable sharing of space and computation across synopses. More details about sharing can be found in [ABW03].

  Each store supports one storage allocator and a set of synopses. Each synopsis is associated with one store, and all the tuples of the synopsis are allocated by the store. Table 4 lists the types of stores and the synopses that they support. The set of synopses associated with a store are required to satisfy certain properties as mentioned in Table 4.

Store	Supported Synopses	Requirement
Simple Store	-	-
Relation Store	> 1 Relation Synopses	All synopses should have the same insert-delete sequence.
Window Store	> 1 Window Synopses > 1 Relation Synopses	All synopses should have the same insert sequence. Sequence of deletes should be identical to the sequence of inserts.
Partition Window Store	1 Partition Window Synopsis > 1 Relation Synopses	All synopses should have the same insert-delete sequence.
Lineage Store	1 Lineage Synopsis > 1 Relation Synopses	All synopses should have the same insert-delete sequence.

Table 4: Stores and the synopses that they support

Relevant Code:

- `dsms/include/execution/stores/*.h`
- `dsms/src/execution/stores/*.cc`

6.2.4 Global Operational Units

- **Memory Manager:** The memory manager manages a common pool of memory and allocates memory at a page granularity to stores, indexes, and queues on demand.

  Relevant Code:

  - `dsms/include/execution/memory/memory_mgr.h`
  - `dsms/include/execution/memory/memory_mgr.cc`

- **Scheduler:** The scheduler schedules the operators in the system as described earlier. Currently, the scheduler uses a simple, round-robin scheduling strategy.

  Relevant Code:

  - `dsms/include/execution/scheduler/*.h`
  - `dsms/include/execution/scheduler/*.cc`

References

[ABW03] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. Technical report, Stanford University Database Group, Oct. 2003. Available at: `http://dbpubs.stanford.edu/pub/2003-67`.

[MW+03] R. Motwani, J. Widom, et al. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the 1st Conf. on Innovative Data Systems Research*, pages 245–256, Jan. 2003.

[SW04] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proc. of the 23rd ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 263–274, June 2004.

[TMSF03] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. on Knowledge and Data Engg.*, 15(3):555–568, 2003.