Introducing Stream Mill:
User-Guide to the Data Stream Management System,
its Expressive Stream Language ESL, and
the Data Stream Mining Workbench SMM

Hetal Thakkar and Carlo Zaniolo
Computer Science Department
University of California, Los Angeles
hthakkar,zaniolo@cs.ucla.edu

October 8, 2010

1The contributions to Stream Mill by Haixun Wang, Richard Luo, and Yijian Bai are gratefully acknowledged.
Contents

1 Foreword 4

2 Data Streams and DB Tables 5
  2.1 Data Streams .................................................. 5
  2.2 Continuous Queries ........................................... 7
  2.3 Tables ............................................................ 9
  2.4 Snapshot Queries .............................................. 10

3 Client Tutorial 11
  3.1 Using the Client ................................................ 12

4 User-Defined Aggregates 19
  4.1 Base UDAs ..................................................... 19
  4.2 Blocking UDAs ............................................... 24
  4.3 Windowed User-Defined Aggregates ........................... 25
  4.4 Slides ......................................................... 26
  4.5 UDAs Defined Using C/C++ .................................... 27

5 Scalar and Table Functions 30
  5.1 External Functions ............................................. 30
  5.2 External Table Functions ..................................... 30

6 Extending sql for Online Mining 33
  6.0.1 Naive Bayesian Classification (NBC) ....................... 33

7 Mining Models 41
  7.0.1 Concept Drifts and Shifts ................................ 42
  7.0.2 Mining Flow ............................................... 43
  7.0.3 Formal Definition of MMDL ................................ 45
  7.1 Other online mining algorithms ............................... 46
    7.1.1 Clustering ................................................ 46
    7.1.2 Association Rule Mining ................................ 47
  7.2 Experiments .................................................... 51
Chapter 1

Foreword

Stream Mill is a Data Stream Management System (DSMS) designed to support complex applications, along with the simpler applications serviced by other DSMS. In particular, Stream Mill is the first DSMS to support stream mining applications; this is achieved by the Stream Mill Miner (SMM) workbench that is discussed later in this document. For now, we will focus on providing a user-friendly introduction to the system and its powerful query language Expressive Stream Language (ESL), which extends SQL into a Turing-complete language by means of User Defined Aggregates (UDAs).

Most of the ESL constructs are applicable to both database tables and data streams. This simplifies the task of ESL programmers who are likely to develop and test their ESL queries on tables before applying them to streams. It is recommended that new users learning the system follow this procedure; in fact, Stream Mill provides tools that facilitate the process of migrating queries on tables to continuous queries on data streams.

Stream Mill is based on a client/server architecture. Thus, the server is always running, and managing the incoming data streams. Users can login through a client and register continuous queries on those streams, in addition to snapshot (a.k.a. ad hoc) queries on the database. While snapshot queries complete and disappear, continuous queries keep sending additional results to the client until the queries are explicitly turned off by the user. The client is discussed in detail in Chapter 3. In writing his/her queries the ESL programs can use the following declared objects:

1. Imported data streams,
2. Data streams derived from other streams,
3. Database tables, residing on disk or main memory,
4. Tables derived from data streams,
5. User-Defined Aggregates, with, and without windows,
6. Table Functions.

In the next chapter we discuss items 1-4. In Chapter 3 we discuss the Stream Mill client. Items 5-6 are discussed in Chapter 4, and SMM is presented in Chapters 5 & 6.
Chapter 2

Data Streams and DB Tables

The basic data types supported by ESL on both data streams and tables are as follows:

- **INT**
- **REAL**
- **CHAR(n)**
- **TIMESTAMP** (newly implemented—MORE)

In addition to these, ESL supports a *reference* data type and an *array* data type; these are respectively used in the special context of memory tables and external UDAs, and will be discussed later in such contexts.

2.1 Data Streams

External data streams are imported into Stream Mill using the keyword `STREAM`. Here are three simple examples:

```
STREAM OpenAuction (itemID int, 
                  sellerID char(10), price real, start_time timestamp) 
ORDER BY start_time /* external timestamps */ 
SOURCE 'port4445';
```

```
STREAM ClosedAuction (itemID int, 
                      buyerID char(10), price real, current_time timestamp) 
ORDER BY current_time /* internal timestamp */ 
SOURCE 'port4446';
```

```
STREAM Bids (itemID int, 
             price real, bidderID char(10), bid_time timestamp) 
SOURCE 'port4447';
```
Thus, the schema declarations for data streams are similar to those for tables—modulo the three clauses order by, source, and target discussed next.

**Order by and Timestamps** ESL supports the following three kinds of timestamps: (i) *external timestamps*, (ii) *internal timestamps*, and (iii) *latent timestamps*.

*External timestamps* are values that are contained in the arriving tuples (e.g., placed there by application producing the data); therefore all is needed in the data stream declaration is to identify the column containing such timestamps using the order-by clause. For instance the data stream *OpenAuction*, defined above, is declared as having *start_time* as its external timestamp.

The *ClosedAuction* declaration, above, is instead assigned an *internal timestamp*, generated by recording the current time in a new column added to the incoming tuple. This new column is always called *current_time*—a reserved name used only to denote internal timestamps. Internal timestamps and external timestamps can also be viewed as *explicit* timestamps: ESL operators treat all explicit timestamps in the same way, no matter how they were originally generated.

When the order by clause is omitted in the CREATE STREAM there is no explicit timestamp. Such stream have *latent timestamps*. While the values of internal timestamps are made explicit eagerly as tuple enter the system, the values of latent timestamps are instantiated lazily and only when they are needed in operations whose semantics depends on explicit timestamps (e.g., union or window aggregates). For most operations, only the order of the tuples is of importance and the order can be supported without explicit timestamp columns in the tuples. Therefore, latent timestamps can reduce the size of the tuples and deliver the convenience of operational semantics when needed.

**Source and Target** The *source* clause, connects the stream source to the Stream Mill system. Stream Mill provides automatic wrappers that can listen on a specified port. These automatic wrappers, however expect the input to be in CSV (comma separated, line delimited) format. As shown above, all three streams use these automatic wrappers, denoted by *SOURCE 'portYYYY'*, where YYYY indicates the port on which the

---

1 As shown above, the data stream *Bid* has latent timestamps—but this is not the only data stream with latent timestamps created by the declarations above. In fact, a stream with latent timestamps is implicitly created for each externally timestamped stream. For externally timestamped streams, there is no assurance that their tuples arrive sorted according to their timestamps. ESL solves this problem by reassigning late tuples to a separate stream with latent timestamps, which can then be handled directly by the user. The name of these streams is that of the externally timestamped streams with the postfix *outOfOrder*. Therefore, the first declaration, above, also produces the data stream *OpenAuction.outOfOrder*, which has the same attributes as *OpenAuction* but latent timestamps.
data will arrive in CSV format. The use of the word *port* before a number, instructs the system to use the default wrapper at the indicated port. Then, data can be sent to this port via a simple TCP client; Appendix E shows a sample sender written in perl.

Stream Mill also supports sending the input data directly from the client which is also described in Appendix E. Alternatively, the users can write their own wrappers, which can accept input in any format and restructure it as required.

Similarly, results from ESL queries can be returned to a port, for consumption by external applications. The target clause specifies the machine and the port where the results should be pushed (e.g. TARGET ‘myserver.domain.com:YYYY’). In general, a stream defined with a target clause is a hook to connect the output to external applications. Thus, any tuples inserted into this stream can be seen at the external application in CSV format. Finally, for testing purposes, Stream Mill allows the user to send the data in a table to a stream. This will enable the user to test his program before deploying it over live data streams. This also allows the user to build his application over database tables and slowly migrate to data streams. This is illustrated by the following example.

```sql
TABLE myTab(a int, b real, c timestamp) SOURCE 'myTab';
STREAM myStr(a int, b real, c timestamp) SOURCE TABLE myTab;
```

Thus, every time the stream ‘myStr’ is activated, the system will dump the data in ‘myTab’ into ‘myStr’. Currently this feature is only available with the following restrictions.

- The source table must be disk resident and it should not have any keys/index over it.
- The schema of the table and the stream must match exactly, i.e. without any casting. Furthermore, the length of char fields must also be the same.

### 2.2 Continuous Queries

Blocking queries are not allowed on data streams. Thus the only queries allowed are those expressing combinations of the following operations: selection, projection, union and join with a table (or a window on a stream). Queries involving monotonic aggregates and window aggregates are also allowed and will be discussed in later sections.

**Derived Streams.** ESL allows deriving a stream from an existing stream using the create stream construct as shown in the example below.

```sql
CREATE STREAM expensiveItems AS (  
    SELECT itemID, sellerID, price, start_time  
    FROM OpenAuction WHERE price > 1000);
```
This expression defines a new stream that can be used in the `from` clause of other queries. Observe that this is similar to the ‘create view’ construct of SQL, and thus can be cascaded any number of times.

**Unions** Union of two union-compatible streams is allowed (see example below), here union-compatible means matching schema and matching timestamp type.

```sql
/* Union */
SELECT itemID, price, start_Time, 'bid' FROM Bids
UNION
SELECT itemID, price, current_Time, 'Close' FROM CloseAuction;
```

The direct join of two streams is not allowed; however the join of a stream with a database table or the join of a stream with a window on another stream is supported. Here we have a table named `StandardPrice` that memorize the typical price for a particular `itemID`. Thus the following query finds items that are closed above their typical price.

**Join with Tables**

```sql
/* Windowed Join */
SELECT b.itemID, b.price AS p1, c.price AS p2
FROM ClosedAuction AS c,
    TABLE(Bids OVER(RANGE 10 MINUTE PRECEDING)) AS b
WHERE c.itemID = b.itemID AND c.price > 1.2 * c.price;
```

Clearly `TABLE(Bids OVER(RANGE 10 MINUTE PRECEDING)) AS b` generates an object that has the properties of a database table. In fact this object can be replaced with an actual table, whereby we get the join of a stream with a database table. Since the result of joining a stream with a table is another stream, we are here dealing with a continuous query, whose results are sent to the client. Also, the results of the previous union query are output to the client.

**Tables Derived From Data Streams.** Additionally, ESL allows creating a snapshot of a stream by defining a table over the stream. Such snapshot tables are defined using a range predicate or a count predicate, i.e. these tables can contain the tuples that have arrived within last N minutes/hours/days (logical) or the last K tuples (physical) in the stream. We use the `over` clause with the `rows` predicate below, which can also be replaced with a range predicate to create a logical table. Note that this table is kept up-to-date by the system and can be used like any other database table.

```sql
CREATE TABLE HotItems AS (
    SELECT itemID, bidderID, price, start_time
    FROM Bids OVER (ROWS 100 PRECEDING CURRENT)
    WHERE price > 10000) REFRESH IMMEDIATE;
```
As described next, a query on this derived table returns a table and it is thus a snapshot query (a.k.a. ad hoc).

### 2.3 Tables

Three types of tables are currently supported in Stream Mill, as follows:

1. Secondary storage tables with a B+ tree index on one or more attributes, \(^2\):

   \[
   \text{TABLE employees(Eno int, Name char(18), Sal real, Dept char(6))}
   \]
   \[
   \text{BTree(Eno) source 'mydb/employees';}
   \]

   The SOURCE declaration associates this table with the file `/mydb/employees` and make it persistent. Data in persistent tables remains even after the Stream Mill system restarts. Tables declared without a source are transitory and the data in these tables is lost if the system restarts.

2. Secondary storage tables with indexed by R+ trees on a pair or a quadruplet of real attributes: the first can be used to index points, and the second for rectangles:

   \[
   \text{TABLE mypoints(x real, y real, object char(10)) RTE(x,y);}
   \]
   \[
   \text{TABLE myrectangles(tx real, ty real, bx real, by real, object char(10)) RTE(tx,ty,bx,by);}
   \]

3. Main memory tables, with a hash-based index on one or more attributes:

   \[
   \text{TABLE memo(j Int, Region Char(20))}
   \]
   \[
   \text{MEMORY AS VALUES(0, 'root-of-tree');}
   \]

   This example also illustrates that a transitory table can be initialized to the results of the query defined using the AS query option. In this example, we use constant values to initialize the table. In general, the initialization query can use the content of previously declared tables. Note that the AS query clause is optional.

   A source declaration can only be given for B+tree and R+tree tables, but not for MEMORY tables since these are never persistent. MEMORY tables also support a special reference data type that is quite convenient in implementing complex data structures. This is discussed in Appendix A.

   The LOAD construct of ATLaS/Stream Mill can be used to load into a table data from an external file in CSV format. Newly loaded tuples are appended to the existing tuples. Since the user does not have access to the server, this statements are more suitable for ATLaS.

---

\(^2\)These are managed by the embedded database system Berkeley DB [6]
## 2.4 Snapshot Queries

Tables can be consulted in any continuous query on a data stream; however queries involving only tables are always snapshot queries, i.e., they are transient queries, that complete their execution after accessing the current tables. Also, continuous queries cannot update tables—whereas snapshot queries can. For instance, say that an employee table, with key Eno, is stored in such a format in `/mydb/employees`. Then the following program first gives a 5% raise to the employees in the ‘QA’ department, and then prints all the employees who now make more than 60K:

```sql
/* Begin of ATLaS program— this is a comment*/
table employees(Eno int, Name char(18), Sal real, Dept char(6))
    Btree(Eno)  source '/mydb/employees';
update employees set Sal = Sal * 1.05
    where Dept='QA';
insert into stdout select Eno, Name
    from employees where Sal > 60000;
/* End of ATLaS program */
```

The `insert into stdout` clause before `select` in the last statement can be omitted without changing the meaning of this program. Thus Stream Mill supports the standard `select, insert, delete, update` statements of SQL-2 over database tables. Observe that keywords can be written in upper case or lower case—however, attribute names and other user-defined identifiers are case-sensitive.

When writing complex queries on data streams, the user should first write and test the analogous queries on tables. Stream Mill provides a number of tools to facilitate a graceful transition from queries on tables to queries on data streams.
Chapter 3
Client Tutorial

Stream Mill is based on a client server architecture, where multiple clients can connect to a single server. The server is always running, whereas users can start a new client and then exit it any time to view/monitor their streams/queries. The Stream Mill Client is shown in Figure 3.1. We will describe each of its components in detail here.

The client allows 3 main types of functions, as follows: (A) File editing, (B) Object viewing/monitoring, and (C) Object definition. Corresponding to these 3 functions
there are 3 modes, namely, Not Logged-in, View Library, and Logged-in. In Not Logged-in mode, only functions of type A are available, while in the View-Library mode both functions of type A and B are available. Functions of all types are available in the Logged-in mode. We discuss a typical usage scenario in Section 3.1 and provide a detailed description of each command in Table 3.1.

3.1 Using the Client

Working on Local Files. When the user starts the Stream Mill client, the mode is Not-Logged-in. In this mode, the user can edit the local (client-side) files through File and Edit menus.

Login into the server. Returning users can now move to ‘Logged-in’ state by ‘User→Login’ menu item. However, new users must first create an account using the ‘User→Create New User’ menu item. A new user must give a user name, a password, and email.

Defining the Streams. The user can now type (or better cut and paste) the declarations of streams in the Editor pane. For instance, user can load the following declaration, or any number of STREAM and CREATE STREAM statements into the Editor pane:

STREAM traffic(station_id INT, speed INT, time TIMESTAMP) 
    SOURCE 'port4439';
CREATE STREAM fasttraffic from traffic where speed >10000;

Then, by issuing the ‘Create→Stream(s)’ menu item these streams are uploaded into the server and (assuming no syntax errors) the streams are recorded in the system catalog under this user name.

Complex Aggregates and Functions. User-defined aggregates and table functions, described in the next chapters play an important role in advanced applications. By clicking on ‘Create→Aggregate’ menu item, the aggregate in the Editor pane is uploaded and stored in the Stream Mill catalog. This can be verified via the ‘View/Modify→Aggregates’ menu item. Note, aggregates must be created one by one.

---

1Currently the system does not allow password retrieval, thus the users are urged to remember their user name and password (the email is not utilized, but it is required and should have correct syntax).

2For simplicity, we assumed that the data arrives on port 4439 and it will be in CSV format. In cases where these assumptions do not hold, the user can define a custom data source as shown in Appendix D. For this simple example, we will have a perl script that will send data at this particular port in the required format, as given in Appendix E.
**Defining the Queries.** The next step is to define a continuous query that will view the data arriving in this stream, such as follows:

```
SELECT station_id, speed, time FROM traffic;
```

Once this query is in the Editor pane, the user will click on ‘Create→Quer(ies)’ menu item and upload the query. The query is then cataloged, and ready for activation.

**Activating Continuous Queries and Streams.** The queries (streams) that, as result of the previous steps, have now been entered in the server catalog should now be activated using the ‘View/Modify→queries (streams)’ menu item. This pops up a dialog box, which shows the defined queries (streams). Once the complete path from the source stream to the query has been activated results are returned to the client.

**Monitoring Continuous Queries.** The user can monitor the results of the activated queries via the Monitoring Engine, which is started using the ‘View/Modify→Monitoring Engine’. The Monitoring Engine shows a list of queries, which may return results to the user. Currently, in Stream Mill queries are assigned a unique id, that consists of the user’s ip address and a unique number. If the query returns results to the user, the results are fed to a standard output buffer that is named as ‘stdout,’ concatenated by the query id. Thus, the user should look for the buffer that corresponds to the query. The user should right click on the buffer and click the ‘Activate→Text Viewer’ menu item. This will show the results of the query in a separate text box. Results will appear in this text box as soon as data is fed to the source stream.

**Sharing Libraries.** If a user decides to make his/her definitions available to others, he/she can invoke the ‘User→Make Public’ menu item in Logged-in mode to create a library ³. This allows other people to use his definitions by prepending the object names with ‘UserName$’. Others can also view all his definitions by using the ‘Library→View Libraries’ menu item. This pops up a combo box that lists all the libraries on the Stream Mill system. User can select the library that he is interested in and click ‘Ok’. This brings the client in the View Library mode. In this mode only File editing and Object viewing/monitoring functionalities are available, i.e. in this mode the user cannot delete the definitions of the library or create new ones.

Thus, the three modes of the client facilitate the user to interact with the server to manage his/her objects and view results. Note, for user convenience the client also has a ‘Tool bar’ with buttons for different create/view/modify tasks. While the server retains all user definitions, we encourage the user to save his/her definitions locally, to avoid inconvenience in case of server failure. Next, we describe couple of unique features of the Stream Mill client.

³Of course, user can later use the ‘User→Make Private’ menu item, in case he/she changes his mind.
Using Script Files. Instead of defining different items, such as streams, tables, queries, and aggregates, separately, the client allows uploading them at the same time as a script. This still allows the user to group declarations in modules. Furthermore, these scripts can also activate the created modules. For instance, let’s consider the script below.

```plaintext
1: # at symbol used to parse, so no at symbols in comments either
2: # mystreams is the module name, multiple modules w/ repeating streams tag
3: @streams mystreams
4: stream abc(a int, b int) source 'port5445';
5: stream abcd(a int, b int) source 'port5446';

6: # gets default module name here, again can have multiple
7: @tables
8: table a(x int) memory;

9: # each aggregate must be preceded by the 'at' aggregate tag
10: @aggregate
11: aggregate tmp(a int):int {
12: initialize: iterate: {
13: insert into return values(a);
14: }
15: }

16: # myqueries is the module name, again multiple modules with queries tag
17: @queries myqueries
18: select a, b from abc;
19: select tmp(a) from abc
20: # to activate modules by name
21: @activate queries myqueries
22: @activate streams mystreams
```

We note that the order of definitions in the script is important, i.e. if a query depends on a declaration, then the declaration must be specified first. In the example script above, line 3 specifies a streams tag (@streams), thus the following lines define some streams. Furthermore, ‘mystreams’ in line 3 specifies the module name for this set of streams. While assigning a module name is voluntary, if one is assigned, it can be used to activate the module. Lines 4 and 5 define couple of streams. The user can create multiple such stream modules by repeating the streams tag (@stream). Line 7 specifies a tables tag (@tables), i.e. it will be followed by table declarations. Note in this case we have not assigned a module name. Line 10 defines an aggregate tag, i.e. it is followed by an aggregate definition. We note that each aggregate must be preceded by the aggregate tag (@aggregate). Line 17 defines a queries tag with module name ‘myqueries’, which is followed by a series of continuous queries. Again, users can define multiple query modules by repeating the queries tag. Finally, lines 21
and 22 activate query and stream modules defined previously using module names for reference. Special tags discussed here may be repeated as many times as needed in any order. Furthermore, if a script crashes in the middle of execution, it does not ‘undo’ its actions that completed successfully.

**Launching Ad-Hoc Queries and Updates on the Database.** These are also referred to as snapshot queries. Users can simply load these queries/statements to the editor pane and click on ‘View/Modify→Execute Snapshot Query’ menu item. The results of these queries are presented in the Results pane. Since the server must translate and compile these queries before running them, this usually takes 3-4 seconds.

<table>
<thead>
<tr>
<th>Menu Item</th>
<th>Description</th>
<th>Available in Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Used for file management and printing.</td>
<td>All Modes</td>
</tr>
<tr>
<td>Edit</td>
<td>Used for file editing.</td>
<td>All Modes</td>
</tr>
</tbody>
</table>

**User Menu**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User→Create New User</td>
<td>First time users of stream mill must create a user for themselves. This will ask the user simple questions, such as user name, password (2 times for verification), and email. Currently the system does not allow password retrieval, thus the users are urged to remember their user name and password (The email is not utilized, but it is required and should have correct syntax). On successful creation, this command brings client into Logged-in mode for the newly created user, else the mode does not change.</td>
</tr>
<tr>
<td>User→Login</td>
<td>Once the user has created an account, he/she can simply log in using this menu item. Upon logging in the client enters the ‘Logged-in’ mode, which allows the user all 3 functionalities.</td>
</tr>
<tr>
<td>User→Make Public</td>
<td>This command creates a library for this user, thus other users can view and use his/her definitions.</td>
</tr>
<tr>
<td>User→Make Private</td>
<td>This command makes the user private, thus other users can no longer view or use his/her definitions.</td>
</tr>
<tr>
<td>User→Logout</td>
<td>Logs out the current user and brings client back in Not Logged-in mode.</td>
</tr>
</tbody>
</table>

Create Menu
<table>
<thead>
<tr>
<th><strong>Create→Data Source</strong></th>
<th>If the user wants to create a custom data source, they can create it using this menu item. The user should load the C/C++ code for the data source in the editor pane and invoke this menu item. Example of a custom data source is given in Appendix D. Note, results of invoking this command, including errors, are shown in the Status pane.</th>
<th>Logged-in mode only.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Create→Table(s)</strong></td>
<td>The user loads the table definition(s) in the editor pane and invokes this menu item. Note, in this case the user can define multiple tables at the same time, which allows the user to group table definitions. The user is also asked a name that he would like to assign to this group of declarations, so that they can be deleted/activated together, such groups are also referred to as modules in Stream Mill. Again the results of this command are displayed in the Status pane.</td>
<td>Logged-in mode only.</td>
</tr>
<tr>
<td><strong>Create→Stream(s)</strong></td>
<td>Similar to ‘Create→Table(s)’ menu item, but for defining data streams. User can create multiple streams at the same time.</td>
<td>Logged-in mode only.</td>
</tr>
<tr>
<td><strong>Create→Script</strong></td>
<td>This menu item is used to create tables, streams, aggregates, and continuous queries all at same time. This allows the user to manage different declarations as a single program. User can also activate queriesstreams he/she created, by module name.</td>
<td>Logged-in mode only.</td>
</tr>
<tr>
<td><strong>Create→Aggregate</strong></td>
<td>The user can define aggregates using this menu item. The user should load definition of an aggregate in the Editor pane and invoke this command. Windowed aggregates are also defined similarly. Note, the user should define one aggregate at a time, unlike table and stream declarations. Also note, that in Stream Mill if an aggregate invokes another aggregate, then the invoked aggregate must be defined inside the main aggregate. CAGGREGATES are also defined similarly.</td>
<td>Logged-in mode only.</td>
</tr>
<tr>
<td><strong>Create→External Aggregate</strong></td>
<td>Similar to ‘Create→Aggregate’.</td>
<td>Logged-in mode only.</td>
</tr>
<tr>
<td><strong>Create→External Function</strong></td>
<td>In this case, the user must provide the definition of the function in ESL and also provide its implementation in C/C++. Thus, the user should load the ESL definition of the function in the Editor pane and invoke this menu item. This results in a file chooser menu, which prompts the user to select the file in which the C/C++ code for the external function resides, thus the user should pick the C/C++ file that contains this code. Note, the user should omit the source clause in the external function definition in ESL.</td>
<td>Logged-in mode only.</td>
</tr>
<tr>
<td>Create—Quer(ies)</td>
<td>Similar to table and stream declarations the user can also create multiple continuous queries at the same time using this menu item.</td>
<td>Logged-in mode only.</td>
</tr>
<tr>
<td>Create—Time Series Query</td>
<td>Use this menu item to create an SQL-TS query. Note, this command only takes one query at a time.</td>
<td>Logged-in mode only.</td>
</tr>
<tr>
<td>Create—Model Type</td>
<td>This menu item is used to create new model, which is discussed with SMM extension.</td>
<td>Logged-in mode only.</td>
</tr>
<tr>
<td>View/Modify Menu</td>
<td></td>
<td></td>
</tr>
<tr>
<td>View/Modify—Data Sources</td>
<td>This menu item pops up a new window that lists the data sources defined by this user. This new window also allows deleting the data sources. Deletion is not allowed in View Library mode.</td>
<td>Logged-in and View Library mode.</td>
</tr>
<tr>
<td>View/Modify—Tables</td>
<td>This menu item pops up a new window that lists the tables defined by this user and let’s the user delete them.</td>
<td>Logged-in and View Library mode.</td>
</tr>
<tr>
<td>View/Modify—Streams</td>
<td>This menu item pops up a new window that lists the streams defined by this user and let’s the user activate, deactivate, and delete them.</td>
<td>Logged-in and View Library mode.</td>
</tr>
<tr>
<td>View/Modify—Aggregates</td>
<td>This menu item pops up a new window that lists the Aggregates defined by this user and let’s the user delete them.</td>
<td>Logged-in and View Library mode.</td>
</tr>
<tr>
<td>View/Modify—External Functions</td>
<td>This menu item pops up a new window that lists the external functions defined by this user and let’s the user delete them.</td>
<td>Logged-in and View Library mode.</td>
</tr>
<tr>
<td>View/Modify—Queries</td>
<td>This menu item pops up a new window that lists the queries defined by this user and let’s the user activate, deactivate, and delete them.</td>
<td>Logged-in and View Library mode.</td>
</tr>
<tr>
<td>View/Modify—Time Series Queries</td>
<td>This menu item pops up a new window that lists the SQL-TS queries defined by this user and let’s the user activate, deactivate, and delete them.</td>
<td>Logged-in and View Library mode.</td>
</tr>
<tr>
<td>View/Modify—Model Types</td>
<td>This menu item pops up a new window that lists the Model Types defined by this user and let’s the user delete them.</td>
<td>Logged-in and View Library mode.</td>
</tr>
<tr>
<td>View/Modify—View All Definitions</td>
<td>This menu item pops up a new window that shows all defined objects in a text box.</td>
<td>Logged-in and View Library mode.</td>
</tr>
<tr>
<td>Menu Item</td>
<td>Description</td>
<td>Mode</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>View/Modify→Monitoring Engine</td>
<td>This menu item starts the Monitoring Engine, which shows the results of continuous queries. The monitoring engine lists all the queries of this user, which can potentially return results (whether they are active or not). Currently, in Stream Mill client queries are assigned a unique id, that contains the ip address and a unique number. If the query returns results to the user, the results are fed to a standard out buffer that is named as ‘stdout’ concatenated by the query id. Thus, the user should look for the buffer that corresponds to the query he/she is interested in. The user should right click on the buffer and click the ‘Activate→Text Viewer’ menu item. This will show the results of the query in a separate text box. Other views are also available for advanced applications. Note, user can view results of multiple queries at the same time. Upon activation, text boxes for multiple buffers may overlap, thus the user is encouraged to move them around as convenient.</td>
<td>Logged-in and View Library mode.</td>
</tr>
<tr>
<td>View/Modify→Execute Snapshot Queries</td>
<td>This menu item allows the user to query database tables and tables derived from streams. User can execute multiple snapshot queries at the same time. Their results are displayed in the Status pane (without much formatting, for now).</td>
<td>Logged-in and View Library mode.</td>
</tr>
</tbody>
</table>

**Library Menu**

<table>
<thead>
<tr>
<th>Menu Item</th>
<th>Description</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library→View Libraries</td>
<td>This pops up a combo box with a list of available libraries. User can select one of these and click ‘Ok’ to view definitions in this library. Upon clicking ‘Ok’ the client enters the View Library mode for the selected library/user.</td>
<td>Logged-in and View Library mode.</td>
</tr>
</tbody>
</table>
Chapter 4

User-Defined Aggregates

Stream Mill supports the five standard aggregates `count`, `sum`, `avg`, `min`, and `max` without the `DISTINCT` option. Stream Mill requires that the user uses `minr`, `maxr`, and `sumr` aggregates for real values. These built-in aggregates can be invoked in three different modes:

1. **Basic Invocation**: this is the traditional SQL-2 fashion, with optional group-by and having clauses,

2. **Window Invocation**: this follows the syntax and semantics of SQL:2003 OLAP functions with logical or physical windows.

3. **Window+Slide Invocation**: This is an extension of the last semantics introduced by DSMS, as described later in this chapter.

User-Defined Aggregates (UDAs) represent a main source of power that turns ESL into a Turing Complete language (also Non-blocking Complete, since it can express all monotonic functions [20]). A major strength of Stream Mill with respect to other DSMSs, is that it supports slides and windows on arbitrary aggregates, as opposed to only built-in ones. ESL allows a user to optimize the performance of a UDA on the various kinds of invocation by declaring (i) a base version and (ii) a window version of such UDA. The user can do that natively, in ESL, or using C/C++. The details of this process are described in this chapter.

### 4.1 Base UDAs

As a first example, we define an aggregate equivalent to the standard `avg` aggregate in SQL.
**Standard Average** The first line of this aggregate function declares a local table, `state`, to keep the sum and count of values processed so far. While, for this particular example, `state` contains only one tuple, it is in fact a table that can be queried and updated using SQL statements and can contain any number of tuples (see later examples). These SQL statements are grouped into three blocks labelled respectively `INITIALIZE`, `ITERATE`, and `TERMINATE`. To compute the average, the SQL statement in `INITIALIZE` inserts the value taken from the input stream and sets the count to 1. The `ITERATE` statement updates the table by adding the new input value to the sum and 1 to the count. The `TERMINATE` statement returns the final result(s) of computation by `INSERT INTO RETURN` (to conform to SQL syntax, `RETURN` is treated as a virtual table; however, it is not a stored table and cannot be used in any other role):

```sql
AGGREGATE myavg(Next Int) : Real
{
    TABLE state(sum Int, cnt Int);
    INITIALIZE : {
        INSERT INTO state VALUES (Next, 1);
    }
    ITERATE : {
        UPDATE state SET sum=sum+Next, cnt=cnt+1;
    }
    TERMINATE : {
        INSERT INTO RETURN SELECT sum/cnt FROM state;
    }
}
```

The basic initialize-iterate-terminate template used to define the average aggregate of SQL-2, can now be used to define powerful new aggregates required by new database applications.

**OnLine Average** For instance, there is much current interest in online aggregates [19]. Since averages converge toward the correct value well before all the tuples in the set have been visited, we can have an online aggregate that returns the average-so-far every, say, 200 input tuples. (In this way, the user or the calling application can stop the computation as soon as convergence is detected.) Online averages can be expressed in ESL as follows:

```sql
AGGREGATE online_avg(Next Int) : Real
{
    TABLE state(sum Int, cnt Int);
    INITIALIZE : {
        INSERT INTO state VALUES (Next, 1);
    }
    ITERATE : {
        UPDATE state SET sum=sum+Next, cnt=cnt+1;
    }
}
```
UPDATE state SET sum=sum+Next, cnt=cnt+1;
INSERT INTO RETURN
SELECT sum/cnt FROM state WHERE cnt % 200 = 0;
}
TERMINATE : {  }
}

Therefore, the online average program has been obtained from the traditional average program by removing the statements from TERMINATE and adding a RETURN statement to ITERATE. Our UDA online_avg takes a stream of values as input and returns a stream of values as output (one every 200 tuples). In this example only one tuple is added to output by the the INSERT INTO RETURN statement; in general, however, such statement can produce several tuples. Thus ESL UDAs operate as general stream transformers.

ESL uses the same basic framework to define both traditional aggregates and non-blocking aggregates. ESL UDAs are non-blocking when their TERMINATE clause is either empty or absent.

The default semantics for SQL aggregates is that the data is first sorted according to the GROUP-BY attributes; this is a blocking operation. However, ESL’s default semantics for UDAs is that the data is pipelined through the INITIALIZE and ITERATE clauses where the input stream is transformed into the output stream: the only blocking operations (if any) are those specified in TERMINATE, and only take place at the end of the computation.

**SQLCODE.** This a convenient labor-saving device found in most SQL systems, that comes very handy for the ESL programmer who wants to correlate a statement with the next. SQLCODE is set to a positive value when the last statement had a null effect, and to zero otherwise. Thus to tell the user that no employee was found in department 1024, we can modify the previous program as follows:

```sql
SELECT Sex, online_avg(Sal)
FROM employee WHERE Dept=1024 GROUP BY Sex;
SELECT 'Nobody found in that department'
WHERE SQLCODE >0;
```

In the last statement, the predicates in the WHERE clause controls its conditional execution, in a fashion similar to that of the IF clauses in a procedural programming language. In fact, the ESL compiler recognizes, and optimizes execution of, such conditional predicates.

**Minima: Points and Values.** In the next Example, we have a sequence of point-value pairs, and we define a minpair aggregate that returns the point where a minimum occurs along with its value at the minimum.
AGGREGATE minpair(iPoint Int, iValue Int): (mPoint Int, mValue Int)
{
    TABLE mvalue(value Int) MEMORY; TABLE mpoints(point Int) MEMORY;
    INITIALIZE: {
        INSERT INTO mvalue VALUES (iValue);
        INSERT INTO mpoints VALUES(iPoint);
    }
    ITERATE: {
        UPDATE mvalue SET value = iValue WHERE iValue < value;
        DELETE FROM mpoints WHERE SQLCODE = 0;
        INSERT INTO mpoints SELECT iPoint FROM mvalue
            WHERE iValue = mvalue.value;
    }
    TERMINATE: {
        INSERT INTO RETURN SELECT point, value FROM mpoints, mvalue;
    }
}

Here we used two internal tables: the mvalue table holds, as its only entry, the current min value, while mpoints holds all the points where this value occurs. In the ITERATE statement we used SQLCODE to ‘remember’ if the previous statement updated mvalue; this is the situation in which the old value was larger than the new one and the old points must be discarded.

The last statement in ITERATE adds the new iPoint to mpoints if the input value is equal to the current min value. Note, the formal parameters of the UDA function are treated as constants in the SQL statements. Thus, this third INSERT statement adds the constant value of iPoint to the mpoints relation, provided that iValue is the same as the value in mvalue—thus the FROM and WHERE clauses operate here as conditionals. The RETURN statement returns the final list of min-pairs.

For instance, say that we have a time series containing the daily closing prices of certain stocks arranged in temporal sequence (i.e. the table stock_prices, below). Then the following program computes the local minima for each stock:

```sql
/* The declaration of AGGREGATE minpair should go here*/
TABLE stock_prices(Day int, Stock char(4), Cprice Real)
    source '/mydatabase/stock_prices'
    select Stock, minpair(Day, Cprice) -> iPoint, minpair(Day, Cprice) -> iValue
    from stock_prices
    group by Stock
```

Observe the use of “→” to identify the different components in the two-column tuples returned by the aggregate minpair. Instead of using the int type for Day column, we can also use the newly implemented timestamp data type. We do not do that here since the minpair UDA works over integers.

The next table summarizes the syntax for declaring new aggregates.
### Initializing Tables and Combining Blocks

Let us now introduce the following two syntactic variations of convenience supported in Stream Mill:

- Tables declared in UDAs can be initialized as part of their declaration, via an SQL statement. This executes at the time when the first tuple is processed, thus the result is the same as if the initialization had been executed in the INITIALIZE block.

- Different blocks can be merged together when they perform the same function.

In the next example the INITIALIZE and ITERATE blocks are merged together.

Our Online Averages UDA could also have been written as follows:

```sql
AGGREGATE online_avg(Next Int) : Real
{
    TABLE state(sum Int, cnt Int) AS VALUES(0, 0);
    INITIALIZE:ITERATE:
    {
        UPDATE state SET sum=sum+Next, cnt=cnt+1;
        INSERT INTO RETURN
        SELECT sum/cnt FROM state WHERE cnt % 200 = 0;
    }
}
```

In the previous example, the statement has been omitted: this is equivalent to writing `TERMINATE: { }`. An empty INITIALIZE statement can also be omitted in a similar fashion.

The results produced by online average depends on the order in which the data is streamed through the UDA. This illustrates a common situation in stream processing: the abstract semantics of the aggregate used is order-independent, but approximations must be used because of efficiency and real-time requirements (e.g., nonblocking computations); often, the approximate UDA is order-dependent.

In other situations, no approximation is involved, and the dependence on order follows from the very semantics of the UDA. For instance, this is the case of the rising aggregate described below.

### Rising

In addition to temporal extensions of standard aggregates (suggested homework: write them in ESL), TSQL2 [17] proposes this new aggregate to return the...
maximal time periods during which a certain attribute value has been increasing mono-
tonically. We can apply this aggregate to our stock\_prices(Day Int, Stock char(4),
Cprice Real) table to find the periods during which different stocks have been rising,
as follows:

```sql
select Stock, rising(Day, Cprice) → Start, rising(Day, Cprice)→ End
from stock\_prices
  group by Stock
```

where `rising` is defined as follows:

```sql
AGGREGATE rising(iPoint Int, iValue Real) : (Start Int, End Int)
{ TABLE rperiod(First Int, Last Int, Value Real) MEMORY;
  INITIALIZE:
  { INSERT INTO rperiod VALUES (iPoint, iPoint, iValue);
  }
  ITERATE:
  { INSERT INTO return SELECT First, Last
    FROM rperiod
    WHERE iValue <= Value AND First < Last;
    UPDATE rperiod SET Last=First, Value=iValue
    WHERE iValue > Value;
    UPDATE rperiod SET First=Last, Last=iPoint, Value=iValue
    WHERE SQLCODE > 0;
  }
  TERMINATE:
  { INSERT INTO return SELECT First, Last
    FROM rperiod
    WHERE First < Last;
  }
}
```

Therefore we have a sequence of time-value ordered by increasing time. We store
a zero length period `iPoint, iPoint` whenever the new `iValue` is not increas-
ing (also at INITIALIZE). Also if a non-zero length period is currently held in
rperiod, we return it. When the new `iValue` is larger than the previous stored
value, we advance the End of the current period to the current point.

## 4.2 Blocking UDAs

Base UDAs that return values in TERMINATE are blocking. Blocking query operators
return results only after they have seen their whole input: thus they cannot be applied
on a data stream using the basic invocation. Base aggregates that do not return values
in their TERMINATE can instead be applied to a data stream, or a table, using the basic
invocation. This is for instance the case of online\_avg where the following statement is correct independent on whether employee(Eno, Name, Sex, Dept, Sal), is a stream or a table:

```
SELECT Sex, online\_avg(Sal)
FROM employee WHERE Dept=1024 GROUP BY Sex;
```

Blocking aggregates can only be applied over data streams using a window (or window+slide). Stream Mill allows the user to provide a special version of aggregates that is optimized for windowed invocation as we discuss next.

### 4.3 Windowed User-Defined Aggregates

Therefore, ESL allows invocation of arbitrary UDAs with an OVER clause as proposed in the SQL standards and as shown in the example below.

```
SELECT itemID, avg(price)
OVER(PARTITION BY itemID RANGE 10 MINUTES PRECEDING)
FROM Bids
```

Note such windowed UDAs are currently only supported in Stream Mill. We are currently working on supporting them in ATLaS.

A naive solution to execute this query would be to store all the tuples in the defined window in a temporary table and invoke the aggregate over this temporary table. Upon arrival of another tuple, first update the temporary table and execute the aggregate again over the whole table. However, this naive solution is very inefficient, since it does a lot of redundant computations, therefore ESL allows definition of windowed aggregates as we will see next. In situations where the user has not specified such windowed implementation, ESL falls back to this naive solution. A windowed version of the standard average aggregate is presented below.

```
WINDOW AGGREGATE avg(Next Int) : Real {
    TABLE inwindow(Next Int);
    TABLE state(sum Int, cnt Int) AS VALUES(0, 0);
    INITIALIZE:ITERATE: {
        UPDATE state SET sum=sum+Next, cnt=cnt+1;
        INSERT INTO RETURN
        SELECT sum/cnt FROM state;
    }
    EXPIRE: {
        UPDATE state SET sum=sum-oldest().Next, cnt=cnt-1;
    }
}
```
For all windowed aggregates Stream Mill system maintains an inwindow table that contains the tuples in the window. The declaration of this table is optional (it will be created and maintained regardless). The schema of this inwindow table exactly matches the input of the aggregate. Note the user can call this aggregate using the OVER clause with RANGE or ROWS predicate, the system takes care of maintaining the window, thus the user only writes one aggregate, which is generic for logical or physical windows.

Given this, the execution of the above aggregate works as follows. Up on arrival of a tuple, first check if there are any expiring tuples. If there are expiring tuples, for each such tuple first execute the expire routine then pop the tuple out of the inwindow table. Then, insert the new tuple in the inwindow table and execute the initialize/iterate routine. Note the oldest() function above gives the value for the specified column of the oldest tuple of the inwindow table, i.e. of the tuple that is going to expire.

Such windowed implementations are highly optimized, efficient, and consume much less memory compared to the naive solution discussed above. For example consider the windowed version of the standard max aggregate below.

```sql
WINDOW AGGREGATE max(Next Int) : Int {
  INITIALIZE: ITERATE:
  {
    DELETE FROM inwindow WHERE inwindow.Next < Next;
    INSERT INTO RETURN VALUES oldest(inwindow.Next);
  }
  EXPIRE: {
  }
}
```

Note for the max aggregate we do not store all the tuples in the inwindow table. In fact, we can delete some of the tuples that are superseded by more recent tuples. Therefore the above aggregate only keeps tuples that can potentially become max and deletes the rest of the tuples. In general, SQL statements are allowed to delete records from the inwindow table, but not insert/update records.

An unlimited preceding clause is also allowed over these aggregates as follows, to specify an unlimited window.

```sql
SELECT itemID, max(price)
  OVER(PARTITION BY itemID RANGE UNLIMITD PRECEDING)
FROM Bids
```

### 4.4 Slides

Another important construct that comes with windowed aggregates is the slide construct. The slide construct allows the user to specify how often the results of the
aggregate should be reported. The example below shows how this slide construct can be used. In this case the results are returned every 5th tuple.

```sql
SELECT itemID, max(price)
    OVER(PARTITION BY itemID ROWS 19 PRECEDING SLIDE 5)
FROM Bids
```

Thus, the original window of 19+1 tuples is subdivided into 4 panes each containing 5 tuples. The optimal way to implement this is to use the blocking version of max to derive the max for the last incoming group of 5 tuples, and then pass this result to the window version of max that now operates on a window of four tuples. This summarizes the optimized implementation of builtin aggregates in the system. Moreover the same approach based on the pipelining of the two versions of the aggregate can also be used for arbitrary UDAs as described in [2].

In the previous example, the window is divided into smaller panes. However, the pane can also be larger than the window. For instance in the following example, average of the last 10 minutes are returned every hour:

```sql
SELECT avg(price), time
    OVER(PARTITION 10 MINUTES PRECEDING SLIDE 60)
FROM Bids
```

The situation where the window and the slide have the same size is called a tumble. Clearly, when the size of the slide is equal to or greater than that of the window, every new window is recomputed anew using the base version of our blocking aggregate.

Such optimizations are generically employed in Stream Mill to optimize any arbitrary UDAs. We do not discuss this feature much here, since it does not affect the language or its expressive power.

### 4.5 UDAs Defined Using C/C++

Stream Mill supports the definition of UDAs in C/C++. We explain this feature through an example below.

```c
CAGGREGATE avg(next REAL) : REAL {
    @/* global c stuff here, includes, function definitions, static vars, etc */
    @/* other ESL defs, tables etc */
    @/ status structure */
    double tsum;
    int tcnt;
    @
```
A few intricacies should be noted for these C/C++ aggregates. C/C++ code is put between `@` symbols. Furthermore, `@` symbols cannot be randomly, in fact only limited number of C/C++ code blocks (surrounded by `@` symbols) at designated locations are allowed. First we note that the signature of the aggregate is almost the same as for ESL aggregates, apart from changing the keyword AGGREGATE with CAGRRE-
GA TE. Next the first C/C++ code block, where the user should put any includes, func-
tions, structure/class definitions, and static variables. This is followed by an optional
block of ESL definitions, in case the user wants to define any ESL tables, which can be
accessed through C/C++ code. Next is another C/C++ code block, the user should de-
fine aggregate state variables here. In ESL state of an aggregate is maintained in a spe-
cial structure called ‘status’. For ESL aggregates, this structure consists of ESL tables,
however for C/C++ aggregates user can put any valid C/C++ variables. Note that to
access these variables in C/C++ code inside initialize/iterate/expire/terminate clauses,
the user must use ‘status->varName’. Finally, the initialize/iterate/expire/terminate
clauses, which consist of one and only one C/C++ code block. Note that since these
aggregates may return multiple values, they are returned as a table, called ‘status-
>ret’. Furthermore, the inwindow table for windowed aggregates is also stored in the
status structure as ‘status->win’. Also note that ESL creates a separate status struc-
ture for each group in a specified query. Finally, if the user defined any ESL tables in
these aggregates, they can also be accessed as ’status->tableName’. Furthermore, all
tables discussed here, i.e. the ‘status->ret’ table, ‘status->win’ table, and any ’status-
>tableName’ are accessed/modified using BerkeleyDB API, as shown in the example
above for the ’status->ret’ table. These C/C++ aggregates are used just like UDAs
written natively in ESL.

CAGGREGATE provides an important tool to realize an extensible system, since it
allows us to incorporate functions from foreign libraries in Stream Mill. For instance,
in Appendix F, we discuss the integration of C4.5 code downloaded from xxx.
Chapter 5

Scalar and Table Functions

Besides aggregate functions, ESL supports scalar functions and table functions.

5.1 External Functions

In addition to the set of built-in scalar functions described in the appendix, ESL allows users to import external functions, defined in C/C++ or other languages. For instance, the user can provide the following statement in the editor pane to import the function `ginif` which takes one integer-type parameter and returns an integer result.

```
external int ginif(a int);
```

Such external functions are supported with the help of shared libraries. As described in the Client chapter, the user is asked for a C/C++ code corresponding to this function, which will be compiled in a shared library. At this point, the function is available to the user, who can test it with a statement such as:

```
select gini(a) from test;
```

The function can also be exported and made available to other users.

5.2 External Table Functions

In much the same way, we can use external UDF as table functions.

For instance, we want to stream through the first K Fibonacci numbers. It is not difficult to write a C function to generate the Fibonacci numbers. The following ATLaaS program demonstrates how to use such an external table function.

```
external table (i int, f int) fib(k int);

select t.i, t.f from table (fib(10)) as t;
```
In order to declare an external table function, we must use TABLE as the return type. The above declaration indicates ‘fib’ is an external function and ‘fib’ returns a stream of tuples (i,f), where f is the i-th Fibonacci number. Then, in the following query, we stream through the first 10 Fibonacci numbers by calling ‘table (fib(10))’.

How do we implement a table function in C? Unlike stateless scalar functions, table functions must keep their internal state between calls. More specifically, the function must be able to: i) determine the first call from subsequent calls; ii) tell the caller whether a tuple is successfully returned; iii) use a mechanism to return tuples to the caller. As an example, the following code implements function 'fib':

```c
#include <db.h>

struct result {
    int a;
    int b;
};

int fib(int first_entry, struct result *tuple, int k) {
    static int count;
    static int last;
    static int next;

    if (first_entry == 1) {
        count = 0;
        next=1;
        last=0;
    }
    if (count++ <k) {
        tuple->a = count;
        tuple->b = last;

        last = next;
        next = next+tuple->b;
        return 0;
    } else {
        return DB_NOTFOUND;
    }
}
```

In addition to the arguments (here is ’k’) passed to the table function, we have 2 extra arguments: i) first_entry, if first_entry=1 then it is the first call; ii) tuple, which is a pointer to a structure where results are to be stored. External table functions always return an integer value, 0 if successful, DB_NOTFOUND otherwise.
A possible use of table functions is to scan file system data, and return results to the database system after filtering. Our test indicates that on a Linux system, external table functions accessing file system data is almost 100 times faster than accessing the same data in the Berkeley DB format.
Chapter 6

Extending sql for Online Mining

In this section we introduce (i) the extensions to SQL That are required to support data mining tasks and (ii) the additional constructs needed to support them in a data stream environment. Toward this goal we use Naive Bayesian Classifier (NBC) as an example classifier. NBC is a simple, but effective classifier, which is useful in many real world applications.

6.0.1 Naive Bayesian Classification (NBC)

Let us consider the situation where we want to train a classifier on a static table, such as the PlayTennis example of Table 6.1.

<table>
<thead>
<tr>
<th>Outlook</th>
<th>Temp</th>
<th>Humidity</th>
<th>Wind</th>
<th>Play</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>Weak</td>
<td>No</td>
</tr>
<tr>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>Strong</td>
<td>Yes</td>
</tr>
<tr>
<td>Overcast</td>
<td>Hot</td>
<td>High</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>Rain</td>
<td>Cold</td>
<td>Medium</td>
<td>Weak</td>
<td>No</td>
</tr>
<tr>
<td>Rain</td>
<td>Cold</td>
<td>High</td>
<td>Strong</td>
<td>No</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 6.1: The **PlayTennis** relation

In the training phase, NBC only requires derivation of the number of ‘Yes’/‘No’ decisions for each categorical value appearing in the training table. As a running example, the counts, after the first three tuples of the PlayTennis table have been processed, are as in Table 6.2.

Let’s name this table **DescriptorTbl**. The first three entries in **DescriptorTbl** have been generated by using a standard SQL statement such as the following.

```sql
SELECT 1, Outlook, Play, count(*)
FROM PlayTennis
```
<table>
<thead>
<tr>
<th>Col</th>
<th>Val</th>
<th>Dec</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sunny</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Sunny</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Overcast</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Hot</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Hot</td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>High</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>High</td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Weak</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Weak</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Strong</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>All</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>All</td>
<td>Yes</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 6.2: The **DescriptorTbl** relation after processing first 3 tuples

GROUP BY Outlook, Play

Similar SQL statements for Temp, Humidity, and Wind attributes will generate the rest of the entries. At the end we also add up the counts of ‘Yes’ and ‘No’ and store them under the label ‘All’.

A limitation of this approach is that it requires a varying number of SQL statements for tables having a different number of columns. Therefore, we cannot write a **generic** query to create the **DescriptorTbl** directly from the original table. However, systems such as Weka, assure **genericity**, whereby a mining algorithm can be applied over arbitrary tables, regardless of their schema. Therefore, to enable generality SMM provides a generic table function called **verticalize**, whose merits become even more obvious once we look at the prediction task. Say that we want to classify a tuple \( t = (\text{Sunny}, \text{Hot}, \text{High}, \text{Weak}) \). Then an invocation of **verticalize** would return the following tuples, \((1, \text{Sunny}), (2, \text{Hot}), (3, \text{High}), \text{and} (4, \text{Weak})\).

Now by taking the natural join of the vertical tuples with the **DescriptorTbl** above, we get the entries that we have to multiply, as the first step to predict the classification of this tuple. Obviously multiplications can be performed efficiently by simply adding the logs of these numbers. Note, we use \( \text{Count} + 1 \), to implement the Laplace estimator [4], which avoids the problem of non-existent value-class combinations. This means that we should have 1 for the col-value combinations that do not occur, i.e. count 0, but \( \log(1) = 0 \), thus this will not affect the sum aggregate. Thus, the following SQL query defines a view, namely **Alternatives**, over the **TestTuples** and **DescriptorTbl** tables.

CREATE VIEW Alternatives AS 
SELECT d.Dec, sum(abs(log(d.Count + 1)))
FROM TestTuples AS t, DescriptorTbl AS d,
    TABLE(verticalize(t.Outlook, t.Temp, t.Humidity, 
        t.Wind, 'All')) AS ts(ColNo, Val)
GROUP BY d.Dec

The Alternatives view has K tuples, where K denotes the number of possible classification decisions, and for each such decision, we have the product of probabilities represented in log form. Thus for the simple case of ‘Yes’ and ‘No’ the following query can be used to return the more probable classification (with ties arbitrarily broken for ‘Yes’).

CREATE VIEW Decision AS
SELECT t1.Dec
FROM Alternatives AS t1, t2
WHERE t1.Sumlogs > t2.Sumlogs OR
(t1.Sumlogs = t2.Sumlogs AND T1.Dec='Yes')

Let’s now get back to the training problem, where originally the queries to generate the DescriptorTbl were not generic, i.e. they were specific to a table schema. However, given a vertical view over the table, training can also be performed with a simple generic query. For instance, given an SQL query of Example 1, SMM generates a vertical view from the PlayTennis table, as shown in Table 6.3. Thus, counting is then performed with a simple generic query as in Example 2.

Example 1 Vertical view over PlayTennis table

CREATE VIEW Vertical
SELECT ts.Col, ts.Val, t.Dec FROM PlayTennis as t,
    TABLE(verticalize(Outlook, Temp, Humidity, Wind,
    'All')) AS ts (Col, Val)

<table>
<thead>
<tr>
<th>Col</th>
<th>Val</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sunny</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>Hot</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>High</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>Weak</td>
<td>No</td>
</tr>
<tr>
<td>5</td>
<td>All</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>Sunny</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Hot</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>High</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>Strong</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>All</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 6.3: Verticalized view over PlayTennis table

Example 2 Counting over vertical view

CREATE VIEW DescriptorTbl AS
SELECT Col, Val, Dec, count(*)
FROM Vertical
GROUP BY Col, Val, Dec

Thus, with small extensions SQL can support NBC generically over databases. However, this approach cannot be directly applied over data streams, due to many challenges. First is the application of standard SQL aggregates, such as sum, count, etc., which are blocking and thus cannot be applied over data streams. Secondly, the self-join query above is also blocking and can take unbounded memory. These problems have been previously recognized by researchers in extending SQL for data streams [12, 3, 5], who have proposed the use of window constructs to alleviate the blocking behavior of these operators. However, we note that window constructs alone are not enough to express many complicated mining algorithms. Therefore, SMM supports an extension of SQL, namely Expressive Stream Language (ESL) [5], which minimally extends SQL to express complex queries over data streams, as we discuss next.

In the streaming scenario, the TestTuples table now becomes an input stream defined by the following statement, where the source clause at the end of the statement denotes the port on which the data arrives. SMM creates a CSV wrapper to read data arriving at this port. Alternatively, the user may also provide his/her own wrapper to read the data. Finally, this statement may also contain an order by clause, which denotes the order of the tuples (typically a timestamp value).

CREATE STREAM TestTuples (Outlook CHAR(10),
Temp CHAR(10), Humidity CHAR(10),
Wind CHAR(10)) SOURCE 'port5444';

Then the previous create view definitions must be changed by replacing the CREATE VIEW clause with the CREATE STREAM clause. We also create a vertical stream, namely VerticalStream, from the TestTuples stream, similarly to Example 1. Furthermore, ESL extends SQL with User Defined Aggregates (UDAs) and windows over UDAs to provide tremendous expressive power, both theoretically [20] and practically [5].

Therefore, the ClassifyNaiveBayesian UDA, given in Example 3, classifies test tuples using the NBC generated by the SQL query of Example 2.

Example 3 NB Classification Aggregate

AGGREGATE ClassifyNaiveBayesian(col INT,
val CHAR(10)): CHAR(3) {
TABLE pred(dec INT, tot REAL);
INITIALIZE: {
INSERT INTO pred
SELECT Dec, abs(log(Count+1))
FROM DescriptorTbl;
}
ITERATE: {
UPDATE pred p SET tot = tot +
(SELECT abs(log(Count+1))
FROM DescriptorTbl WHERE Val = val
AND Col = col AND Dec = p.dec);

}  
TERMINATE: {
    INSERT INTO RETURN
    SELECT p.dec FROM pred p
    WHERE NOT EXIST (
        SELECT * FROM pred p1
        WHERE p1.tot > p.tot
        OR (p1.tot = p.tot AND p1.dec < p.dec));
    }
}

The *ClassifyNaiveBayesian* UDA (Example 3) sums up the probabilities of each outcome, up on arrival of each vertical tuple in the **initialize** and **iterate** states. These states essentially maintain a sum for each value-dec combination. The **terminate** state determines the most likely outcome based on the maintained sums. However, we note that the computation presented in *ClassifyNaiveBayesian* UDA is blocking, since it only returns the outcome up on seeing the end of the stream in the **terminate** state, thus it cannot be applied over data streams. Therefore, SMM extends standard SQL:2003 windows over UDAs, as opposed to just built-in aggregates, to convert the blocking UDAs to non-blocking. Thus, the *ClassifyNaiveBayesian* UDA is invoked with an ESL query as follows.

```sql
CREATE STREAM Predictions AS
    SELECT ClassifyNaiveBayesian(Col, Val)
    OVER (ROWS 4 PRECEDING SLIDE 5)
FROM VerticalStream
```

The **rows 4 preceding** clause represents a window of 5 tuples (including the current tuple). Furthermore, the **slide 5** clause instructs SMM to return results every 5 tuples. In general, such windows allow invocation of blocking UDAs over streams, since the execution is constrained over the specified window. The queries like the one above are called tumbling window queries in SMM, since the slide size is greater than or equal to the window size [5]. In such cases, SMM repeats the following computation over each tumbling window, uniformly across different UDAs. SMM executes the **initialize** state for the first tuple, the **iterate** state for the next (window size - 1) tuples and then it executes the **terminate** state, since we reached the end of the window. This processing is repeated from the next (slide size - window size) tuples.

Thus, in this case **iterate** state is executed 4 times after the **initialize** state followed by the **terminate** state. The processing is then repeated from the next tuple since the slide size equals the window size. Further optimizations for different window and slide sizes are discussed in [5]. Therefore, SMM enables efficient classification of streams using a simple continuous query, above, with the help UDAs and windows over UDAs.
Furthermore, note that these computations over streams can be viewed as a definition of a workflow where tuples flow from one operator to the next (as shown in Figure 6.1—a computation model that naturally fits the streaming environment, as illustrated by other DSMSs [12]. For instance, Figure 6.1, shows verticalization of the test stream followed by Classification. Many mining algorithms in fact require execution of multiple such steps, including cleaning, post-mining, etc. Therefore, SMM supports the complete mining process as a workflow, as we discuss later in the paper.

![Figure 6.1: Data Stream Workflow](image)

Similar to the classification UDA, we can define a generic UDA to train an NBC. If we assume that there is no concept drift or shift, then we can train the NBC on an unlimited preceding window, an SQL:2003 construct fully supported in ESL, as shown in the ESL statement below (the UDA in the query is defined a bit later).

```
CREATE VIEW NBClearn AS
SELECT LearnNaiveBayesian(Col, Val, 4, t.Dec)
OVER (RANGE UNLIMITED PRECEDING)
FROM VerticalStream
```

However, in the presence of concept shifts and drifts, instead of an unlimited preceding window, we would want to train over a window of N recent tuples—where N could be say 50000 tuples.

**Example 4 Invoking Learn Aggregate**

```
CREATE VIEW NBClearn AS
SELECT LearnNaiveBayesian(Col, Val, 4, t.Dec)
OVER (RANGE 50000 ROWS PRECEDING)
FROM vertstream
```

The NBC learner is defined as a UDA, as in Example 5 (this UDA also updates the external table DescriptorTbl).

**Example 5 Windowed Aggregate (Learning NBC)**

```
WINDOW AGGREGATE LearnNaiveBayesian(col INT,
    val CHAR(10), totCols INT, classVal INT) : INT {
    TABLE tupleSummary (Col INT, Val CHAR(10)
        Dec INT, Count INT);
    INITIALIZE: ITERATE: {
        UPDATE tupleSummary
            SET Count = Count + 1
```
WHERE Col = col AND Val = val
    AND Dec = classVal;
INSERT INTO tupleSummary VALUES (col, val,
    classVal, 1) WHERE SQLCODE <> 0;
/* we omit some additional details here, for clarity
updateDescriptorTbl is another UDA, that simply,
updates the value in the DescriptorTbl. */
SELECT updateDescriptorTbl(Col, Val, Dec, Count)
    FROM tupleSummary WHERE col = totCol;
DELETE FROM tupleSummary
    WHERE col = totCols;
}
EXPIRE: {
    UPDATE tupleSummary
    SET Count = Count - 1
    WHERE Col = oldest().col AND
        Val = oldest().val AND Dec = classVal;
}
}

The LearnNaiveBayesian UDA (Example 5) is a windowed aggregate, which also
consists of an additional state, namely EXPIRE. The exp state is executed once for
each expiring tuple. The system automatically determines the tuples that have ex-
pired, based on the window size specified in the query. Thus, the aggregate above
keeps the statistics for each original tuple (INITIALIZE and ITERATE state), till it sees
the last vertical tuple of the original tuple, i.e. col = totCols, in the tupleSummary
table. At the last vertical tuple, it calls the updateDescriptorTbl UDA that updates
the DescriptorTbl. Note, exp state performs similar computation for tuples expir-
ing out of the window. Thus, windowed UDAs allow the users to specify incremental
computation of mining aggregates.

However, updating the DescriptorTbl for each training tuple is not acceptable
in many cases, since both the classifier stability and efficiency may suffer. We should
instead update it every, say 1000, tuples. This problem is solved with slides, as sup-
ported by many DSMS projects [12, 3, 5]. Therefore, the following ESL query updates
the DescriptorTbl every 1000 tuples.

SELECT LearnNaiveBayesian(Col, Val, 4, t.Dec)
    OVER (ROWS 50000 PRECEDING SLIDE 1000)
FROM vertstream

Then the updateDescriptorTbl UDA, invoked in Example 5, should only be in-
voked at the end of each slide, as denoted by the isEndOfSlide flag, which is set to
1 when the processing reaches the end of the slide, 0 otherwise. Thus, the user can
specify the actions to take when a tuple arrives/expires and also when a slide of tu-
uples arrive/expire. This delta computation method based on the windows and slides
provides tremendous flexibility and efficiency. Therefore, UDAs with windows allow
SQL to overcome its expressive power limitations and thus solve the first challenge, as discussed in the introduction. Furthermore, they can be defined natively, i.e. in SQL itself, or in a procedural language. We show UDAs written in SQL in the examples, since they tend to be high level and concise. For best performance, one will eventually want to redefine mining UDAs in a procedural language—with excellent performance as we show in Section 7.2. It is important to note that when UDAs are defined in a procedural language they will be stored in a library by SMM for future reference. Furthermore, SMM allows the user to define new mining models using the Mining Model Definition Language (MMDL), as we discuss next.
Chapter 7

Mining Models

The formal syntax for MMDL is given in Section 7.0.3. Here, we continue our running example to define an NBC model in MMDL, as shown in Example 6.

Example 6 Model Type for Naïve Bayesian Classifier

```plaintext
CREATE MODEL TYPE NaiveBayesianClassifier {
    SHARED TABLES (DescriptorTbl),
    Learn (UDA LearnNaiveBayesian,
        WINDOW TRUE,
        PARTABLES(),
        PARAMETERS()),
    Classify (UDA ClassifyNaiveBayesian,
        WINDOW TRUE,
        PARTABLES(),
        PARAMETERS())
};
```

The NBC mining model has two tasks associated with it, namely Learn and Classify. The definition also specifies the UDAs associated with them. In this case UDAs of Examples 5 and 3, for Learn and Classify tasks, respectively. The model definition also describes the allowed parameters (PARAMETERS clause) and where to store them (PARTABLES clause) for each task. Storing the parameters in tables, allows the user to change them as the algorithm progresses; a feature that is very useful in advance streaming applications, as seen in Section 7.1.2. The window and slide related parameters for data streams are also specified similarly. Finally, we note that both UDAs share the DescriptorTbl, which is defined in Example 2, where the Learn task stores the statistics, which will be used by the Classify task. Thus, a mining model defined in SMM can have multiple tasks that are associated with it. These tasks may share one or more tables. Once the user has defined such a mining model, he/she can begin using the model by first creating an instance of the model as follows,
CREATE MODEL INSTANCE NaiveBayesianInstance
AS NaiveBayesianClassifier;

Once an instance is created as above, the user can invoke the tasks of the mining model. SMM also provides a unified syntax to invoke these tasks. The formal syntax to invoke these tasks is discussed in Section 7.0.3.

7.0.1 Concept Drifts and Shifts

For now, we return to our discussion of continuous maintenance of NBC over a data stream to illustrate the complete integration of advance mining methods in SMM and the extensions that may be required to support them. The UDA of Example 5 maintains an NBC over a window of tuples from the training stream. The size of the window, physical or logical, is defined by the user at query time. However, in presence of concept-shifts and drifts, it is very difficult to determine the size of the window, such that the accuracy does not suffer [24]. Thus, ensemble based methods have been proposed to increase the accuracy of classifiers in presence of concept-drifts and shifts [24, 9, 14]. Without loss of generality, we discuss the generic integration of ensemble based bagging in SMM. We note that other methods can be similarly integrated in SMM through it’s extensible framework.

**Ensemble Based Weighted Bagging:** Ensemble based weighted bagging was proposed in [24] to improve the accuracy of classifiers in presence of concept drifts and shifts. The approach is applicable when there are two parallel streams, a training and a testing stream, and both are generated by the same underlying concepts. The approach divides incoming training stream into disjoint blocks of data (called tumbling windows) and learns a new classifier for each window. The learning can be performed using any arbitrary classifier (called the base classifier). Thus, we have an ensemble of learned classifiers, one for each recent training window. The approach uses the latest window of training tuples to approximate the accuracy of the existing classifiers over the testing stream. Thus, each pre-existing classifier is assigned a weight proportional to its accuracy on the most recent training window. The newly arriving testing tuples are first classified using each of the classifier from the ensemble. Then, a weighted voting scheme is employed to determine the final classification of the test tuples. The flow diagram in Figure 7.1 shows this process pictorially.

In Figure 7.1, hexagon boxes denote UDAs and the labels between these boxes represent data streams. We note that the UDA named ‘Classifier Building’ learns the next classifier to be stored with the ensemble of classifiers. For instance, Example 5, shows ‘Classifier Building’ UDA for NBC. Similarly, the ‘Classification Task’ UDA predicts the classification of each tuple based on each classifier in the ensemble. For instance, Example 3 shows ‘Classification Task’ UDA for NBC. Therefore, in Figure 7.1 the general flow and processing of data tuples, both training and testing, does not depend on the particular classification algorithm used. In fact, only the UDAs labeled ‘Classifier Building’ and ‘Classification Task’ are specific to the particular classification algorithm used. Thus, any classification algorithm that provides implementation for
these two tasks, as UDAs, can be used as a base classifier for the weighted bagging. Thus, built-in and arbitrary user defined classification algorithms can take advantage of weighted bagging without having to reimplement it. Furthermore, Example 7 shows the MMDL for ensemble based bagging with NBC, which has 6 mining tasks, namely BuildEns, Train, UpdateEns, Classify, ManageWeights, and Voting. UDAs for these tasks can be easily implemented in our framework, but we skip their discussion in interest of space. Thus, any user can instantiate this mining model and invoke different mining tasks to achieve ensemble based bagging, without writing complex SQL code.

Example 7 Ensemble Based Bagging over NBC

```
CREATE MODEL TYPE BaggingEnsemble {
    SHAREDTABLES (activeEnsembles, ensClassTbl,
                    ensembleWeights),
    BuildEns (UDA BuildEns,
               WINDOW FALSE, ... ),
    Train (UDA LearnNaiveBayesian,
           WINDOW TRUE, ... ),
    UpdateEns (UDA UpdateEnsembles,
               WINDOW FALSE, ... ),
    Classify (UDA ClassifyNaiveBayesian, ... ),
    ManageWeights (UDA UpdateWeights, ... ),
    Voting (UDA WeightedVoting, ... ),
};
```

7.0.2 Mining Flow

While models, as discussed above, define a set of tasks that may share tables among them, the user must know the order in which these tasks must be executed. For instance, ensemble based bagging requires that the training stream goes through series of mining tasks as shown in Figure 7.1. However, assuming that users who want to take advantage of such advance methods, know the details of these mining tasks, is not realistic. Therefore, we observe that the execution of different mining tasks represents a flow of tuples, as shown in Figures 7.1 and 6.1. Therefore, SMM allows
the mining experts to define data flows on top of mining tasks in MMDL. For our initial implementation, these flows are specified in ESL, where the expert defines a step by step execution to connect different mining tasks. For instance, Example 8 defines training and testing flows for ensemble based bagging. Note, each flow has an input stream and an output stream, called instream and outstream, respectively. The flow creates intermediate streams to connect these mining tasks. The detailed syntax of the run statements is discussed in 7.0.3. Note, the flows defined in Example 8 essentially model the graphical flows shown in Figure 7.1 for ensemble based bagging. Furthermore, in future, the SMM system will provide a GUI where such flows can be defined, with model tasks and streams as drag and drop objects. These flows are appended at the end of model type definition, after the definition of mining tasks in MMDL, e.g. in Example 7. Note, the expert may define many such flows with different combination of mining tasks, to provide rich functionality to the user. Then, the user can instantiate the mining models and simply invoke one of the flows, as shown at the end of Example 8, as opposed to having to determine the details of low level mining tasks. Therefore, the complex mining flows are now only specified once, by the data mining experts, and reused by all users.

Example 8  Ensemble Based Bagging Flows

Flow Training (  
CREATE STREAM buildEnsTrain AS  
  RUN BuildEns ON INSTREAM;  
CREATE STREAM NBCTrain AS  
  RUN Train ON buildEnsTrain;  
RUN UpdateEns ON NBCTrain;  
CREATE STREAM ensClassiTrainPairs AS  
  SELECT a.ensId trainEns, b.ensId testEns,  
    b.id, b.col, b.val, b.lbl, b.numCols  
  FROM buildEnsTrain b, activeEnsembles a;  
CREATE STREAM ensClassiTrain AS  
  RUN Classify ON ensClassiTrainPairs;  
RUN ManageWeights ON evalClassiTrain;  
),

Flow Testing (  
CREATE STREAM buildEnsTest AS  
  RUN BuildEns ON INSTREAM;  
CREATE STREAM ensClassiTestPairs AS  
  SELECT a.ensId trainEns, b.ensId testEns,  
    b.id, b.col, b.val, b.lbl, b.numCols  
  FROM buildEnsTest b, activeEnsembles a;  
CREATE STREAM evalClassiTest AS  
  RUN Classify ON ensClassiTestPairs;  
INSERT INTO OUTSTREAM AS  
  RUN Voting ON evalClassiTest;  
),

44
Thus, SMM solves the third challenge of providing a language for mining workflow, via ESL statements that allow the user to define end-to-end mining tasks in MMDL (more Examples in Section 7.1).

### 7.0.3 Formal Definition of MMDL

In this section we present the formal syntax of MMDL as supported in SMM. The language allows definition of new mining models, which can be instantiated by arbitrary users. Mining models consist of different mining tasks that combine to provide a mining algorithm. For instance, for classification algorithms we may have one or more data cleaning tasks followed by a learning task and a classification task. Furthermore, mining models consist of flows that connect its mining tasks, e.g. ensemble based methods, as discussed previously, to define end-to-end mining process. Thus, MMDL allows the definition of mining flows as part of the mining models.

| ModelDef → | CREATE MODEL TYPE ModelNm '{
| SHAREDTABLES '{ [TableNm]* '}',
| [TaskDef]+[
| [FlowDef]+[
| '}'
| TaskDef → | TaskNm '{
| UDA UdaNm,
| WINDOW 'TRUE'|'FALSE',
| PARTABLES '{ [TableNm]* '}',
| PARAMETERS '{ [ParamNm]* '}'
| '}'
| FlowDef → | FlowNm '{
| SQLStatements
| '}'

**Table 7.1: MMDL Syntax**

Thus, in SMM, a data mining model is composed of one or more UDAs, which can share one or more tables that contain required state information. For instance, in the case of NBC, there are two UDAs, one for learning (Example 5) and another for predicting (Example 3), and they share the DescriptorTbl, as shown in Example 6. Additionally, they also consist of mining flows that specify end-to-end mining process. Many classification algorithms, ensemble based methods, association rule mining algorithms, etc., fall under this general framework, as we discuss in later Sections.

The formal syntax of MMDL is given in Table 7.1. MMDL allows the user to define tasks that belong to the mining model; each model must have at least one task.
associated with it. Furthermore, it allows the user to specify the UDA associated with each task and the parameters that it accepts. It also allows the user to specify zero or more tables (created outside or by the UDA) that will be shared by the tasks. Note that each instance of the mining model will use a different instance of these shared tables, thus different invocations of the mining algorithm do not conflict with each other. Furthermore, MMDL allows the data mining experts to define zero or more mining flows for complex mining processes, as seen in Example 8. Then, the users can create an instance of the mining model and invoke the tasks and flows of the mining model. The unified syntax used to invoke these tasks is given in Table 7.2.

\[
\text{TaskInv} \rightarrow \text{RUN ModelInstNm} \cdot \cdot \cdot \text{TaskNm} \\
\hspace{1em} \text{WITH TableStreamNm} \\
\hspace{2em} \text{[USING (' [ParamAssign]* ')'?]} \\
\text{ParamAssign} \rightarrow \text{ParamNm} '=' \text{Value}
\]

Table 7.2: Task Invocation Syntax

Therefore, the \textit{Learn} task of the NBC, defined in Example 6, is invoked as follows.

\texttt{RUN NaiveBayesianInstance.Learn WITH TrainingSet;}

Note, the \textit{TrainingSet} is assumed to have the same schema as expected by the UDA associated with the \textit{Learn} task—the system performs this validation automatically. Also, note that the USING clause is omitted in the above statement, since there are no additional parameters to be passed, but more advance algorithms may require them.

Thus, SMM effectively solves the first and third challenges, as discussed in the introduction, with UDAs, windows over UDAs, and MMDL, which allows definition of mining models with workflows. Next, we discuss integration of a few advance mining methods in this framework, to address the second challenge discussed in the introduction.

### 7.1 Other online mining algorithms

The proposed framework is useful in expressing many other analytic and mining tasks, such as sequence pattern matching, clustering, association rule mining, etc. For instance, SMM incorporates the SQL-TS language, based on Kleene Closure, for time series queries, which is being considered for inclusion in SQL standards. Here we focus on clustering and association rule mining, since these represent the core data mining methods, other than classification.

#### 7.1.1 Clustering

Many algorithms have been proposed for online data stream clustering [1, 10, 17, 7]. These algorithms incrementally maintain a clustering over a continuous window as the
stream arrives. Furthermore, many of these algorithms further divide the window in disjoint blocks, called tumbles, and combine the execution for tuples in each tumble. This is easily captured via the windows/slides framework supported in SMM. Here we discuss the integration of D-stream [7] in the proposed framework, since it does not require the user to specify the number of clusters and it finds clusters of arbitrary size and shape. Furthermore, it is shown to be faster than other state-of-the-art online clustering algorithms [7], such as CluStream.

D-Stream is a windowed grid-based clustering algorithm, which creates clusters based on grid-cell density. As introduced in [7], D-stream has two phases, an online phase and an off-line phase. The online phase determines and stores the grid-cell for each incoming tuple, whereas the off-line phase is only executed every $S$ tuples, where $S$ is the slide size, and updates the maintained clustering. SMM’s window/slide framework naturally accommodates such incremental computation. Thus, in SMM, D-Stream maintains a summary of the grid cells over the whole window, called $W_{sum}$, and the clustering over these cells. Furthermore, it maintains another summary structure for the current slide, called $S_{sum}$. Thus, for each incoming tuple, D-stream simply determines the grid cell that the tuple belongs to and updates the statistics for the cell in $S_{sum}$ (online phase). Every $S$ tuples, D-stream merges $S_{sum}$ with $W_{sum}$ and updates the clustering (off-line phase). Furthermore, SMM performs similar computation over the tuples and slides expiring out of the window. The algorithm described above is easily implemented with a windowed UDA that works for generic data streams. Due to space constraints, we omit the UDA for this algorithm and refer to [13] for more details.

### 7.1.2 Association Rule Mining

Association rule mining is a popular mining task, which also has many online applications. Finding frequent patterns is usually the first step in association rule mining and has been thoroughly tackled by the previous work [11, 18, 22]. In particular, the SWIM algorithm [22] has been recently proposed to address this problem in an online setting. This algorithm, first divides up the stream into smaller slides (a.k.a. panes), and then uses other off-the-shelf mining methods (e.g., fp-growth [18]) to mine each slide individually. SWIM constructs a superset of all globally frequent patterns by taking the union of all locally frequent ones. This superset of candidates will then be verified to detect actual global frequent patterns. Verification constitutes the core computation and bottleneck in the SWIM algorithm, which is solved via conditional counting, described next. Given a set of patterns, the goal of conditional counting is to find their exact frequencies only if they are above a certain threshold. By skipping the full processing of those patterns that are less frequent, a verifier (i.e., a program that performs the verification) can significantly outperform a naive counting in which all patterns are fully processed regardless of their frequency. Thus, SWIM efficiently verifies the aforementioned candidate patterns. Every time that the window slides, both the new and expired slides are processed to accordingly maintain the union of all
candidate patterns, in an incremental fashion. In general, other existing algorithms can
also be easily integrated in the SMM framework. Furthermore, association rule mining
often tends to produce a large number of rules. Thus, many post-processing techniques
have also been proposed to summarize, analyze, mine and rank the rules found from
the mining process [21]. Thus, an online mining workbench must support the complete
mining process, as opposed to only supporting the core problem for finding frequent
patterns.

Previous research projects have focused on integrating association rule mining in
relational DBMSs. For instance, OLE DB for DM supports association rule mining
much in the same way as classification. However, the queries to invoke these algo-
rithms get increasingly complex, due to the required structure of the data [23], which in
turn complicates the integration of other post-processing techniques. Instead, Calders
et al. [6], propose an approach to incorporate association rule mining in relational
databases through virtual mining views. Furthermore, this approach achieves much
closer integration and allows the mining system to push down the constraints related
to frequent patterns mining for optimized execution. In addition to support and confi-
dence thresholds, these also include specification of high (low) importance items that
should always (never) be reported. Therefore, Calders et al. [6] propose a 3-table view
of the discovered association rules as shown below.

\[
\begin{align*}
\text{Sets} & : (sid \text{ int, item int}); \\
\text{Supports} & : (sid \text{ int, supp real}); \\
\text{Rules} & : (rid \text{ int, sida int, sidc int, sid int, conf int});
\end{align*}
\]

The \text{Sets} table stores the frequent patterns by their id and items. The \text{cw Supports}
table stores the support of the frequent patterns by their id. Finally, the \text{Rules} table
stores the confidence of each rule by rule id with antecedent id, consequent id, and par-
rent set id (a set containing the union of antecedent and consequent). This framework
is easily extended to work with data streams, by converting the \text{Supports} and \text{Rules}
tables into streams. Furthermore, it can be extended to integrate different post-mining
techniques as we discuss next.

First, we define a mining model in MMDL that provides similar view over discov-
ered patterns and rules. A sample definition of an association rule mining model is
given in Example 9. Of course, the users can modify and/or extend this mining model
to derive new mining models.

\textbf{Example 9} Association Rule Mining Model

\begin{verbatim}
CREATE MODELTYPE AssociationRuleMiner {
  SHARDED TABLES (Sets, RulesHistory),
  FrequentItemsets (UDA FindFrequentItemsets,
                   WINDOW TRUE,
                   PARTABLES(FreqParams),
                   PARAMETERS(sup Int, uninterestingItems List,
                               rejectedPats List, acceptedPats List, ...),
  );
}.
\end{verbatim}
AssociationRules (UDA FindAssociationRules,
    WINDOW TRUE,
    PARTABLES(AssocParams),
    PARAMETERS(conf Real, corr Real)
),
PruneSummarizeRules (UDA PruneSummarizeRules,
    WINDOW TRUE,
    PARTABLES(PruneParams),
    PARAMETERS(chiSigniThresh Real)
),
MatchWithPastRules (UDA MatchPastRules,
    WINDOW TRUE,
    PARTABLES(AssocParams),
    PARAMETERS()
),
Flow ARMFlow (
    CREATE STREAM FrequentPatterns AS
        RUN FrequentItemsets
        ON INSTREAM;
    CREATE STREAM AssocRules AS
        RUN AssociationRules
        ON FrequentPatterns USING confidence > 0.60
            AND correlation > 1;
    CREATE STREAM PrunedRules AS
        RUN PruneSummarizeRules
        ON AssocRules USING chiSigniThresh > 0.50;
    INSERT INTO OUTSTREAM AS
        RUN MatchPastRules
        ON PrunedRules;
),
);
CREATE MODEL INSTANCE AssocRuleMinerInstance
    OF AssociationRuleMiner;
RUN AssocRuleMinerInstance.ARMFlow ON Transactions
    USING sup > 10, window = 1M, slide = 100K;

In Example 9, association rule mining is decomposed into four sub-tasks, namely FrequentItemsets, AssociationRule, PruneSummarizeRules, and MatchWithPastRules. These are implemented as UDAs, as discussed previously (UDAs can been written declaratively in ESL or in a procedural programing language, such as C++). For instance, we use the SWIM algorithm to continually find the frequent patterns, i.e. FrequentItemsets task, from a set of transactions. Thus, the analyst can invoke the tasks of the mining model, one after the other, in a step-by-step procedure, or simply invoke the ARMine flow, defined at the end of the MMDL (Example 9). This flow is also shown in detail in Figure 7.2.

In step 1, the flow invokes the frequent patterns mining algorithm, e.g. SWIM,
over *instream* (the transactions stream). We specify the size of the window and slide to instruct the algorithm to report frequent patterns every 100K (slide) transactions for the last 1 million (window) tuples along with the support threshold. Also note, the user may specify a list of patterns that should be always rejected (or accepted), regardless of their frequency. A continuous algorithm such as SWIM can utilize these constraints to efficiently prune (or keep) nodes (or include nodes) that may not be of interest (or are of interest regardless of frequency). Indeed, SMM pushes down the parameters specified in the USING clause, to the underlying algorithm for optimized execution. The frequent patterns algorithm must also update the *Sets* table. The results of this frequent patterns algorithm are inserted into the *FrequentPatterns* stream, denoted by the CREATE STREAM construct. The *FrequentPatterns* stream is in fact the same as the *Supports* stream as proposed in Calders et al. [6].

Step 2 finds association rules based on the results of step 1. Any algorithm that generates rules based on frequent patterns can be used for step 2. The algorithm also takes confidence and correlation thresholds to prune the resulting association rules. This task and other tasks downstream may utilize the *Sets* table, thus it is denoted as a SHAREDTABLE in model type definition. Finally, the results of this mining task are inserted into the *AssocRules* stream.

![Diagram](image)

**Figure 7.2: Association Rule Mining: End to end flow**

---

50
of associations, which makes it difficult, if not impossible, for the analyst to manage. Therefore many research efforts have focused on pruning and summarizing the results of association rule mining. For instance, Liu et al. [21] attempts to find rules that are insignificant or that over-fit the data. Therefore, while pruning and summarizing represents an optional step, it has significant practical value for the analyst. Finally, in Step 4, the discovered rules are matched against previous rules, to determine new rules. Since, many rules may already be frequent, the analyst should only be notified about new rules, as opposed to overwhelming him/her with a large set of unchanged rules.

Thus, the complete association rule mining process is supported in this framework, where UDAs and windows allow definition of complex mining and MMDL allows specification of mining tasks and flows among the mining tasks. Furthermore, the framework is extensible in that it allows easy integration of other mining algorithms. Next, we compare the performance of SMM with existing static mining systems.

7.2 Experiments

The two main goals of the experiments are (i) to compare SMM’s performance to existing mining packages, such as Weka, and (ii) to evaluate the overhead of integrating different algorithms in SMM. All experiments were run on a Pentium4, 2.4GHz machine running Linux with 1GB RAM. We used Weka version 3.4.12 and JRE version 1.6.0. Note, all algorithms first load the data into main memory and thus we ignore this time.

7.2.1 SMM vs. Weka

In this experiment we compare two classifiers widely used in real-world KDD applications, namely NBC and C4.5 decision tree classifier over two datasets (one synthetic and one real-world). The first dataset is called Iris (synthetic), which contains 4 real-
valued attributes and a prediction attribute, which can take 3 different values [16]. The data represents lengths and widths of flower petals and sepals and the prediction determines the type of the flower. The second dataset contains heart disease data (real-world) with 13 attributes and a prediction attribute that takes 5 distinct values signifying risk of heart disease [16]. The elapsed time for training and testing 1 million tuples with NBC (i.e. 1 million tuples for training and another 1 million for testing) is presented in Figures 7.3(a) and (b), where HD denotes the Heart Disease dataset. The results show that NBC in SMM always outperforms the one in Weka (more than twice as fast in many cases).

The second set of experiments were over the open source implementation of C4.5 decision tree classifier. SMM allows easy integration of algorithms, whereas Weka requires that the algorithm is written in Java. Thus, Weka implements an equivalent classifier to C4.5, namely J48. Therefore, we compare the standalone version of C4.5 with the one integrated in SMM and the Weka version (i.e. J48). Again, the results for training and testing 1 million tuples are presented in Figures 7.4(a) and (b). It is easy to see that the overhead of integrating C4.5 in SMM is nominal (around 10-15%), whereas J48 is significantly slower (more that twice). Thus, SMM provides an efficient framework that allows easy integration of new and existing mining algorithms. We further study the overhead of SMM in the next set of experiments.

7.2.2 Cost of Integration

In this experiment we compare the performance of SWIM algorithm, implemented as standalone and integrated in SMM as an external aggregate. We use the IBM QUEST data generator [2] Figure 7.5 shows the results for minimum support 0.15%, window size 20K and varying slide size. From Figure 7.5, we see that SMM only has about 15-20% overhead over standalone SWIM. Users are likely to be willing to pay such an overhead, since SMM will save the effort of managing massive and bursty data streams, via QoS guarantees and load shedding techniques supported in SMM, and allow easy integration of such algorithms.

Therefore, the experiments show that along with offering an extensible framework that uniformly and generically supports advance mining algorithms, SMM also enables efficient mining of data streams.

7.3 Related Work

On-line data stream mining has been the focus of many research efforts. For instance, Ester et al. [10] proposed extending a static clustering algorithm, namely DBScan, for continuous clustering of data streams. Similarly, there have been efforts to build online classification algorithms, such as decision tree classifiers [15]. Additionally, researchers have focused on improving the accuracy of on-line classifiers using ensemble based methods [24, 9, 14]. Such approaches are generically supported, i.e. over any
Furthermore, there has also been significant research in finding frequent patterns and association rules over a stream of transactions. For instance, Chi et al. [8] propose the Moment algorithm, which is a differential algorithm to maintain closed frequent patterns over continuous windows. On the other hand, Mozafari et al. [22] proposed the SWIM algorithm to maintain frequent patterns over large sliding windows. The SWIM algorithm is much more scalable in terms of window size and slide size [22], thus we discuss its integration SMM in Section 7.1.2. However, we note that the open and extensible architecture of SMM allows the user to easily integrate new mining algorithms.

7.4 Conclusion

For all the research interest in (i) DSMSs and in (ii) data stream mining algorithms, very little progress had made in the past, toward combining the two—although it is clear that (ii) cannot be successfully deployed without the QoS services provided by (i). Foremost among the technical challenges that prevented this integration, are the SQL-based continuous query languages used by most DSMSs and the lackluster experiences by major vendors in adding KDD extensions to their DBMSs. Thus, the first contribution of SMM is to show that limitations of SQL are overcome by minimal extensions that combine UDaS with window/slide constructs. Thus, the proposed system, SMM, compares with the Weka paragon, in terms of open architecture and extensibility, while outperforming it in terms of performance. Thus, SMM enables integration of new mining algorithms with the help of UDAs, windows/slides over UDAs, and MMDL, which also allows definition of complex mining flows.
Bibliography


[11] Haixun Wang and Wei Fan and Philip S. Yu and Jiawei Han: Mining Concept-Drifting Data Streams using Ensemble Classifiers. KDD 2003.


Bibliography


Appendix A

References and Arrays

The reference type, denoted by \texttt{REF}, is currently supported only for \textit{in-memory} tables, which are declared with the \texttt{MEMORY} option, as in following example:

\begin{verbatim}
TABLE faculty(name CHAR(20), dept CHAR(20)) MEMORY;
\end{verbatim}

In-memory tables can have columns of Reference types, which are essentially points to tuples in some other tables (or refer to themselves). For example, we can define the following tables:

\begin{verbatim}
TABLE faculty(name CHAR(20), dept REF(department)) MEMORY;
TABLE department(name CHAR(20), chair REF(faculty)) MEMORY;
\end{verbatim}

We can find out the name of the chair of the CS department by using the following query:

\begin{verbatim}
SELECT chair->name
FROM department
WHERE name = 'CS';
\end{verbatim}

Object ID and Path Expression In ESL, each tuple in an in-memory table has its unique OID (object id). In SQL statements, we treat OID of a tuple as its pseudo column. The type of the OID column is \texttt{REF(table)}, where table is the table the tuple is in.

The following example demonstrates the use of OID and reference types.

\begin{verbatim}
TABLE tree(name char(10), father REF(tree)) MEMORY;

INSERT INTO tree
SELECT 'mary', t.OID
FROM tree AS t
WHERE t.name = 'tom';
\end{verbatim}
In the above example, we define a table with a column that refers to a tuple in the same table. The subsequent INSERT statement creates a new tuple whose father is another tuple in the table with name ‘tom’. The expression \( t.OID \) retrieves the OID of the current tuple.

With reference types and OID and we can use path expressions to navigate through the tables. The following query finds the name of Jane’s grandfather. Note that if \( t \) is of reference type, then \( t \) and \( t->OID \) are the same thing, which means \( father->name \) is the same as \( father->OID->name \).

```sql
SELECT father->father->name
FROM tree AS t
WHERE t.name = 'jane';
```

A critical application of reference types and path expression is the implementation of in-memory data structures that are critical for the performance of many algorithms, including the Apriori algorithm discussed next.

**Array Types** A new addition to ATLaS/ESL is passing of arrays to an aggregate. ATLaS/ESL now has built-in functions for creating arrays out of list of values. These arrays however can only be used to pass to an aggregate, i.e. they cannot be stored in a database table. Also, ATLaS/ESL provides built-in table functions to convert these arrays of schema (col, val) to tuples, the type of the val column is dependent on the type of the array. Below is a list of array type supported by ATLaS/ESL along with the function to create such an array and table function to flatten such an array. These four different types of arrays store set of int, real, char(), and timestamp values, respectively.

- **IExt** - `newIExt`, `IExtVert`
- **RExt** - `newRExt`, `RExtVert`
- **CExt** - `newCExt`, `CExtVert`
- **TExt** - `newTExt`, `TExtVert`

Note that the `newXExt` functions take arbitrary number of arguments. Also, note that ATLaS/ESL is strict with the types of arguments being passed to these functions. For instance `newIExt` will not accept real values as input. The user can explicitly cast incompatible type by using built-in functions, such as `realtoInt`. Please see built-in functions section for more details. Example usage of these functions is shown below.

```sql
/* In this example, we create a simple int array with three values a, a*a, and a*a*a, using the newiext built-in function. We pass this array to an aggregate show show how this array can be flattened using the iextvert built-in function. */
```
TABLE A(a Int, b Char(10));

AGGREGATE test(a Int, b Char(10), c IExt):Int {
   INITIALIZE:
   ITERATE: {
      SELECT a, b, X.col, X.val
      FROM TABLE(iextvert(c)) AS X;
   }
}

INSERT INTO A VALUES(1, 'abc');
INSERT INTO A VALUES(2, 'abc');

SELECT test(a, b, newiext(a, a*a, a*a*a))
FROM A;
Appendix B

Built-in Functions

ESL supports the following built-in scalar functions. New functions are being added constantly.

- **srand(INT) : INT**
  The srand() function sets its argument as the seed for a new sequence of pseudo-random integers to be returned by rand(). These sequences are repeatable by calling srand() with the same seed value. srand() always returns 0.

- **rand() : REAL**
  The rand() function returns a pseudo-random real between 0 and 1. The following code set 10 as a random seed, and displays two random values.

```sql
VALUES(srand(10));
VALUES(rand(), rand());
```

- **sqrt(REAL) : REAL**
  The sqrt(x) function returns the non-negative square root of x.

- **timeofday() : CHAR(20)**
  The gettimeofday function gets the system’s notion of the current time. The current time is expressed in elapsed seconds and microseconds since 00:00 Universal Coordinated Time, January 1, 1970. It returns a string in the form of x’y”, where x is the seconds and y is the microseconds. This function is mainly used to measure the performance of ATLAS queries, as in the following example:

```sql
INSERT INTO stdout VALUES(timeofday());

... some ATLAS queries ...
```
INSERT INTO stdout VALUES(timeofday());

Many new built-in functions have been implemented recently and are discussed below. Built-in functions to manipulate arrays were discussed in the data structures section. Here we simply list other built-in functions with a brief description.

- **pow** - scalar function, just like c/C++ pow func
- **ceil** - scalar function, just like c/C++ ceil func
- **log** - scalar function, just like c/C++ log func
- **inttostring, stringtoint, stringtoreal, realtostring, inttoreal, and realtime** - set of scalar functions to convert between types
Appendix C

BNF

Syntax for SQL queries supported over database tables is presented in Table C.1. Streams are viewed as append only tables in Stream Mill. Thus, only the select and the insert statements are allowed over streams. Furthermore, blocking operators are not allowed over data streams, thus EXISTS, INTERSECT, and EXCEPT may not be used with data streams. Additionally, ORDER BY does not make sense for streams, since they are always ordered and the order may not be changed. Stream union is permitted along with table joins. Streams may not be used in sub-query. Apart from these differences, there are new types of queries that are available on streams. Therefore, Table C.2 redefines the variables in Table C.1 that require change to accommodate data streams.
### SQL Statements

<table>
<thead>
<tr>
<th>SQL-statement</th>
<th>→</th>
<th>select-st</th>
<th>delete-st</th>
<th>insert-st</th>
<th>update-st</th>
</tr>
</thead>
<tbody>
<tr>
<td>select-st</td>
<td>→</td>
<td>query [order-clause] ';'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>order-clause</td>
<td>→</td>
<td>‘ORDER’ ‘BY’ exp [‘ASC’</td>
<td>‘DSC’] [‘,’ exp [‘ASC’</td>
<td>‘DSC’]]*</td>
<td></td>
</tr>
<tr>
<td>query</td>
<td>→</td>
<td>query-block [set-op query-block]*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set-op</td>
<td>→</td>
<td>‘UNION’</td>
<td>‘INTERSECT’</td>
<td>‘EXCEPT’</td>
<td></td>
</tr>
<tr>
<td>query-block</td>
<td>→</td>
<td>‘SELECT’ hxp [, hxp]*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘FROM’ squn [‘,’ squn]*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[‘WHERE’ exp]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[‘GROUP’ ‘BY’ exp [‘,’ exp]*]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[‘HAVING’ exp]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete-st</td>
<td>→</td>
<td>‘DELETE’ ‘FROM’ id [‘WHERE’ exp] ‘;’</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert-st</td>
<td>→</td>
<td>‘INSERT’ ‘INTO’ id select-st ‘;’ VALUES (‘ exp ’)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>update-st</td>
<td>→</td>
<td>‘UPDATE’ id ‘SET’ update-list [‘WHERE’ exp] ‘;’</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>update-list</td>
<td>→</td>
<td>id ‘=’ exp [‘,’ id ‘=’ exp]*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>exp</td>
<td>→</td>
<td>‘NIL’</td>
<td>num</td>
<td>float</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>exp (‘=’</td>
<td>‘&lt;’</td>
<td>‘&lt;=’</td>
<td>‘&gt;’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>exp (‘AND’</td>
<td>‘OR’</td>
<td>‘IN’</td>
<td>‘NOT’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘max’</td>
<td>‘min’</td>
<td>‘count’</td>
<td>‘sum’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>id ‘(‘ exp [‘,’ exp]* ‘)’</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ref</td>
<td>→</td>
<td>ref ‘→’ id</td>
<td>id [‘,’ id] ‘→’ id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>→</td>
<td>letter [letter</td>
<td>digit]*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>letter</td>
<td>→</td>
<td>(‘a’−‘z’</td>
<td>’A’−‘Z’)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>digit</td>
<td>→</td>
<td>(‘0’−‘9’)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hxp</td>
<td>→</td>
<td>exp [‘AS’] hxp-alias</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hxp-alias</td>
<td>→</td>
<td>id [‘,’ id]* ‘)’</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>squn</td>
<td>→</td>
<td>id qun-alias</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| qun-alias      | → | [‘AS’] id [‘,’ id @ ‘,’ id ‘)’]

Table C.1: SQL Statements supported in Stream Mill over DB tables
### SQL Statements on Streams

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>squn</td>
<td>→</td>
<td>id qun-alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hxp</td>
<td>→</td>
<td>exp ['AS'] hxp-alias</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pby</td>
<td>→</td>
<td>‘PARTITION‘ ‘BY‘ exp [',' exp]*</td>
</tr>
<tr>
<td>range</td>
<td>→</td>
<td>‘RANGE‘ num unit ‘PRECEDING‘</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>slide</td>
<td>→</td>
<td>‘SLIDE‘ num unit</td>
</tr>
</tbody>
</table>

Table C.2: SQL Statements supported in Stream Mill over DB tables
Appendix D

Example Custom Data Source

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/param.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include <dbt.h>
#include <buffer.h>

using namespace ESL;

#define MAX_BUFFER 65536
extern "C" int getTuple(buffer* dest);
extern "C" int closeConnection();

int fdesc = -1;
int listensockfd = -1;
struct sockaddr_in listensock;

int init()
{
    int p;
    /*general purpose */
    int op;

    listensock.sin_family=AF_INET;
```
listensock.sin_port=htons((unsigned short)5533);
listensock.sin_addr.s_addr=INADDR_ANY;

/* create socket, terminate in case of failure */
if((listensockfd=socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == -1) {
    perror("Error calling socket()");
    return -1;
}

if(bind(listensockfd, (struct sockaddr *)&listensock,
        sizeof(listensock))) {
    perror("Error calling bind()");
    return -1;
}

/* make socket listening for connections, terminate in case of failure */
if(listen(listensockfd, 1)) {
    perror("Error calling listen()");
    return -1;
}

op = fcntl(listensockfd, F_GETFL, 0);
if (op != -1) {
    op |= O_NONBLOCK;
    fcntl(listensockfd, F_SETFL, op);
}

p=sizeof(listensock);
if((fdesc=accept(listensockfd, (struct sockaddr *)&listensock,
                 (socklen_t *)&p)) == -1) {
    return 1;
}

op = fcntl(fdesc, F_GETFL, 0);
if (op != -1) {
    op |= O_NONBLOCK;
    fcntl(fdesc, F_SETFL, op);
}

return 0;
int tryAccept()
{
    int p, op;

    p = sizeof(listensock);
    if((fdesc=accept(listensockfd, (struct sockaddr *)&listensock,
                     (socklen_t *)&p)) == -1)
    {
        return 1;
    }

    op = fcntl(fdesc, F_GETFL, 0);
    if (op != -1) {
        op |= O_NONBLOCK;
        fcntl(fdesc, F_SETFL, op);
    }

    return 0;
}

void stringPad(char * src, char * dest)
{
    int strlength = strlen(src);
    int i = 0;
    for(i; i < strlength; i++)
    {
        dest[i] = src[i];
    }
    for(i; i < 9; i++)
    {
        dest[i] = ' ';
    }
    dest[9] = '\0';
}

void processMessage(dbt * data, char * buf)
{
    int numberOfColumns = 3;
    char * val;
    char val1[50];
    int curColumn = 0;
    int offset = 0;
    int pad = 0;
    int temp;

    val = (char*)strtok(buf, "\,");  
    temp = atoi(val);
memcpy(data->data, (char*) &temp, sizeof(int));
val = strtok(NULL, ",");
stringPad(val, vall);
strcpy(data->data + sizeof(int), vall);
val = strtok(NULL, ",");
temp = atoi(val);
memcpy(data->data + sizeof(int) + 10, (char*) &temp, sizeof(int));

struct timeval tv;
struct timezone tz;
gettimeofday(&tv, &tz);
memcpy(data->data + sizeof(int) + 10 + sizeof(int), (char*) &tv,
       sizeof(struct timeval));
data->setTime(&tv);
}

void putDataInBuffer(char* dataBuf, buffer* dest)
{
    char *tupleStr = NULL;
    struct timeval tv;

    char* dataCopy = strdup(dataBuf);

    tupleStr = strsep(&dataCopy, 
                      
    while(tupleStr != NULL && strlen(tupleStr) > 0)
    {
        cDBT tuple(500, &tv);
        int tupleStrSize = strlen(tupleStr);
        processMessage(&tuple, tupleStr);
        dest->put(&tuple);
        tupleStr = strsep(&dataCopy, 
                          
}
}

int getTuple(buffer* dest)
{
    char buf[MAX_BUFFER];
    int rc;

    if(fdesc < 0 && listensockfd < 0)
        rc = init();

    if(rc == -1)
        return -1;  // Error establishing connection

    if(fdesc < 0)
tryAccept();

if (fdesc < 0)
{
    return 2;  // No data
}

int olen = read(fdesc, buf, sizeof(buf));

if (olen == -1 && errno == EAGAIN)
    return 2;  // No data

if(olen == 0)
{
    return 1;  // Connection Closed
}

buf[olen] = '\0';
putDataInBuffer(buf, dest);

return 0;  // Got data
}

int closeConnection()
{
    // printf("Closing connection %d, %d\n", fdesc, listensockfd);
    if(fdesc >0)
        close(fdesc);
    if(listensockfd >0)
        close(listensockfd);
    fdesc = -1;
    listensockfd = -1;
    return 0;
}
Appendix E

TCP CSV Sender in Perl

#!/usr/bin/perl
use FileHandle;
use Socket;
use Time::HiRes qw( usleep);

$usage = "ss.pl host port times delay
where
    host = The name or IP address of the host to connect to
    port = The port number on which the host is listening
    times = Number of times the data should be read
    delay = Delay between the iterations
";

if ($#ARGV<3) {
    print "Missing one or more parameters.\nUsage: $usage"
    exit;
}

$host = $ARGV[0];
$port = $ARGV[1];
$times = $ARGV[2];
$sens = $ARGV[3];

if ($times <= 0 || $sens < 0) {
    print "Missing one or more parameters.\nUsage: $usage"
    exit;
}

#open the socket
$iaddr = inet_aton($host);
$paddr = sockaddr_in($port,$iaddr);
$proto = getprotobynum(’tcp’);

socket(SOCK, PF_INET, SOCK_STREAM, $proto)
```perl
|| die "Couldn’t open socket: $!";
connect(SOCK, $paddr) || die "connect: $!";
autoflush SOCK, 1;

$iter = 0;
while ($iter < $times) {
    $iter = $iter + 1;
    print("Fetching sensor-counts iteration $iter\n");
    system("wget -q http://www.dot.ca.gov/traffic/d7/update.txt -O u.txt");
    system("gawk -f add_time.awk u.txt > sensor-counts");

    print("Sending data to Server\n");
    open(SENSOR, "sensor-counts") || die "Can’t open sensor data file: $!\n";
    $count = 0;
    while (<SENSOR>) {
        # parse the line
        ($stationid, $speed, $time) = split(/,/);

        $str = " $stationid, $speed, $time";
        if ($count < 10) {
            print("sending:$str");
            print(SOCK "\$str");
            $count++;
            usleep(100000);
        }
    }
    close(SENSOR);
    print("Done sending to server. $count tuples sent\n");
    if ($sens > 0) {
        sleep $sens;
    }
}

print("Closing connection to server.\n");
close(SOCK);
exit;
```

### E.1 The Alternative

Alternatively the user can send the data via the client by choosing View/Modify → Send Data. A window such as that in Figure E.1 will be presented where the user can specify the input data file, domain and port to where the input data will be sent. Note: in order to see the output, the monitoring window and the appropriate query must be opened and enabled respectively. Furthermore the port to which the output stream will be sent must be enabled on the client.
Figure E.1: Data Input Window
Appendix F

C4.5 Decision Tree Classifier

table iris(SL int, SW int, PL int, PW int, isSetosa int) memory;

external int createClassifier (SL int, SW int, PL int, PW int,
                                 isSetosa int, ensembleCnt int)
in '/home/hthakkar/c4.5/C4.5/Src/createClassifier.so';

external int evalClassifier (SL int, SW int, PL int, PW int)
in '/home/hthakkar/c4.5/C4.5/Src/evalClassifier.so';

load from 'iris1.data' into iris;

select createClassifier(s.SL, s.SW, s.PL, s.PW, s.isSetosa, 200000)
    from iris as s;

select evalClassifier(SL, SW, PL, PW) from iris;
Appendix G

FAQs and Common Issues

G.1 FAQs

**Question 1** What does “fail to connect to server” mean?

**Answer 1** You will see this message in the ‘Status pane’ if the Stream Mill server is not running. If you have access to the server, you should restart it, else contact the administrator via email given below.

**Question 2** When I click on “View Library” menu, The combo box only has one library named “socket error”, what is wrong?

**Answer 2** This is because the Stream Mill server is not running. If you have access to the server, you should restart it, else contact the administrator via email given below.

**Question 3** I have defined a data stream and a query on it and activated them both. I have also sent input to the port defined in stream definition. How do I see the results?

**Answer 3** You can see the results through the monitoring engine. Open the monitoring engine and right click on the buffer with the same id as the id of the query whose result you want to monitor. You may also want to monitor the error stream, just in case there are any errors.

**Question 4** I have defined a data stream and a query on it and activated them both. I have also sent input to the port defined in stream definition. But I don’t see the results in the monitor engine. What is going wrong?

**Answer 4** There are couple of things that can go wrong.

- The Stream Mill system is behind a firewall, so currently it can only accept input stream on a few ports that are open. Please contact the administrator at the email provided below.
The system you are running the client from may be behind a firewall or it is behind a router. Stream Mill client currently can not send data to host behind firewall or router, since the client requires opening a TCP port as a server.

Question 5  I try defining/viewing objects, but always get an error in the 'Status pane'. What can be wrong?

Answer 5  This is because the client is not able to talk to the server, i.e. either the server is down or the client's Internet connection has problems. You should look at the console massages, if there are any errors there. Usually, you will see a bunch of stack traces there. You should make sure your connection is working properly. If so, then it is most likely an issue with the server. Please contact at the email address given below.

Question 6  What are latent timestamps? Are they same as no timestamps?

Answer 6  Almost. For operators whose semantics depend on time, timestamps are automatically generated for tuples with latent timestamps.

Question 7  What is the syntax for creating derived stream from an existing stream?

Answer 7  See Section 2.2.

Question 8  What is the syntax for creating a windowed table over a stream?

Answer 8  See Section 2.2.

Error Reporting: - If you run across errors when using StreamMill, you can report them to hthakkar@cs.ucla.edu. - We appreciate your cooperation.

G.2 Common Issues

Stream Mill has certain peculiarities that cause undue stress thus we request that you try not to fall in any of the following traps:

- StreamMill is in the testing phase right now, thus it is not error-proof. Thus, we request that you keep your stream/query/aggregate definitions stored in your local hard drives, in case you have to recreate them. Stream Mill will recover them most of the times.

- Please only use alphabetic characters in your user name, not doing this will cause strange errors.

- Remember to end all your definitions in the 'Editor pane' with a semi colon ';'.

76
• Please do not use C/C++ keywords in Stream Mill, again this will cause strange errors.

• Union statements cannot be directly used as output in Stream Mill, thus if you use a union statement, insert its results into a temporary stream and then query the temporary stream to view the results.

• Stream Mill now supports both memory and disk tables. If contents of your table changes frequently then you should use memory tables. However, the contents of memory tables are lost when Stream Mill server restarts. Memory tables are created by using the memory keyword, e.g. table a(..) memory; Disk tables on the other hand are created using 'source' clause, e.g. table a(..) source 'a';.

• In Stream Mill, the source clause should not be used when defining external functions, since the source is automatically defined based on the name of the function.

• To define stream/tables you should use 'stream a(..) source 'portYYYY'" and similarly for tables, i.e. you should omit the 'create' keyword. However, to create derived streams/tables you should include the create keyword, e.g. 'create stream a as (select )'.