SMM: a Data Stream Management System for Knowledge Discovery

Hetal Thakkar #, Nikolay Laptev $, Hamid Mousavi $, Barzan Mozafari $, Vincenzo Russo &, Carlo Zaniolo $

# *Google Inc.*
$ *Computer Science Department, UCLA*
& *ICAR/CNR*
{hthakkar,nlaptev,hmousavi,barzan,russo,zaniolo}@cs.ucla.edu

***Abstract*—The problem of supporting data mining applications proved to be difficult for database management systems and it is now proving to be very challenging for data stream management systems (DSMSs), where the limitations of SQL are made even more severe by the requirements of continuous queries. The major technical advances that achieved separately on DSMSs and on data stream mining algorithms have failed to converge and produce powerful data stream mining systems. Such systems, however, are essential since the traditional pull-based approach of cache mining is no longer applicable, and the push-based computing mode of data streams and their bursty traffic complicate application development. For instance, to write mining applications with quality of service (QoS) levels approaching those of DSMSs, a mining analyst would have to contend with many arduous tasks, such as support for data buffering, complex storage and retrieval methods, scheduling, fault-tolerance, synopsis-management, load shedding, and query optimization. Our Stream Mill Miner (SMM) system solves these problems by providing a data stream mining workbench that combines the ease of specifying high-level mining tasks, as in Weka, with the performance and QoS guarantees of a DSMS. This is accomplished in three main steps. The first is an open and extensible DSMS architecture where KDD queries can be easily expressed as user-defined aggregates (UDAs)—our system combines that with the efficiency of synoptic data structures and mining-aware load shedding and optimizations. The second key component of SMM is its integrated library of fast mining algorithms that are light enough to be effective on data streams. The third advanced feature of SMM is a Mining Model Definition Language (MMDL) that allows users to define the flow of mining tasks, integrated with a simple box&arrow GUI, to shield the mining analyst from the complexities of lower-level queries. SMM is the first DSMS capable of online mining and this paper describes its architecture, design, and performance on mining queries.**

I. Introduction

Problem statement. Data Stream Management Systems (DSMSs) represent a vibrant area of current research, focusing on supporting continuous queries over massive and bursty data streams, while providing quality of service (QoS) guarantees. The development of major research prototypes [17], [20], [6] and commercial systems by high-tech startups [39], [12], underscores the progress achieved by DSMSs and their enabling technology. However, DSMSs remain very limited in their ability to handle complex applications, such as data stream mining. To the best of our knowledge, no commercial DSMS or research prototype supports on-line data stream mining, in spite of its growing importance in an assortment of applications that include: network traffic monitoring, intrusion detection, web click-stream analysis, highway traffic congestion management, credit card fraud detection, and many others [42]. In fact the importance of on-line knowledge discovery has created much interest among data mining researchers, who have proposed fast and light mining algorithms, whereby massive data streams can be mined with real-time response [10], [16], [11], [43], [15], [21]. However, the progress on algorithms has not provided robust and efficient computing environmenst on which to deploy such algorithms with QoS guarantees. However, for a successful deployment on massive data streams, on-line mining queries require the robustness, efficiency, and reliability provided by DSMSs, via technical advances that include the ability to manage queues, synoptic structures (such as windows), scheduling strategies, and load shedding protocols. Therefore, in this paper we address this critical problem, for which, neither DSMS researchers nor data stream mining researchers have provided any solution so far.

Problem significance. The success of KDD systems, such as Weka [13], over stored data illustrates that users much prefer to work with a KDD workbench that allows them to focus on high-level mining tasks rather than having to deal with the complexities involved in implementing these tasks or even customizing existing ones. The need for a workbench such as Weka becomes even more critical in the context of mining data streams, since they require a considerably more complex computational environment than stored data. In fact, even a simple mining task, such as on-line classification, normally requires the concurrent execution of several tasks, including (i) data cleaning, (ii) training, and (iii) prediction, whereas for stored data, the users can perform these tasks leisurely in successive steps. Thus, KDD researchers and practitioners tackling a data stream mining problem would much prefer to concentrate on the data mining task at hand and rely on the DSMS for handling the underlying complexities and delivering the required QoS.

Historical challenges. For all the benefits that can be achieved by extending a DSMS to support a data stream mining environment, this objective poses research challenges so significant that the system proposed in this paper, namely the Stream Mill Miner (SMM) is the first system that claims to have achieved it. The crux of the problem is that most DSMSs use SQL-based continuous query languages and thus face similar issues to those faced by database vendors, who in mid-90s tried to introduce static data mining algorithms in DBMSs. While the objective of extending SQL and DBMSs with OLAP primitives was achieved with relative ease (and great financial returns), supporting data mining tasks using DBMS-provided constructs and functions, proved to be exceedingly difficult for SQL-based relational DBMSs, due to the expressive power limitations of sql [30]; moreover, the well-known study presented in [38] proved that OR-DBMSs do not fare much better either[1].

Therefore, in their visionary paper [29], Imielinski and Mannila called for a quantum leap in the functionality and usability of DBMSs, whereby (i) mining queries can be formulated with the same ease of use as other queries in relational DBMSs, and (ii) query optimizers generate efficient execution plans for these declarative mining queries. The notion of Inductive DBMS (IDBMS) [29] was thus born, which inspired approaches such as MSQL [28], DMQL [26], and Mine Rule [33]. However, these proposals suffer from limited generality and performance. For instance, MSQL and Mine Rule only support association rule mining. Therefore, DBMS vendors have responded to users' demands by less ambitious approaches that are largely based on addition of mining libraries to their DBMSs; examples include the Oracle Data Miner [2], Microsoft's OLE DB for DM [40], and the now discontinued IBM DB2 Intelligent Miner [1].

Question of extensibility. While some of these systems provide unique features, including user friendly graphical interfaces, and support for mining models [40], they also suffer from many limitations. In particular they are closed systems that provide little in terms of user extensibility, and their vendor-specific features are not part of SQL standards. Thus, these proprietary systems are at a disadvantage with respect to Weka [13], an open system, that in addition to a comprehensive set of machine learning methods, supports extensibility through a standard Java API. Thus, while Weka is designed for mining stored data, it outlines a level of functionality, extensibility, and usability that we would like to provide in our workbench for data stream mining.

Contributions. Several ideas leading to SMM were first outlined at a data stream workshop [42], where we investigated the idea of supporting on-line classification in a DSMS. In this paper, we discuss the integration of the complete mining process, using a novel mining language, called MMDL, which allows the users to define new mining methods and work-flows. SMM also supports the rendering and usage of these methods and work-flows by an attractive box&arrow GUI, which greatly enhances the usability of the system. Thus, starting with an SQL-based continuous query language, SMM implements a full-fledged on-line mining system that is extensible, and supports (a) a rich library of mining methods, (b) definition of new mining models and work-flows, (c) a mining-aware optimization and load shedding API to enhance the efficiency and deliver the guaranteed QoS and, (d) a user-friendly GUI. The main research contributions of SMM are as follows:

1) SMM overcomes the expressive power limitations of SQL via powerful UDAs, and extends its DSMS kernel into an on-line mining workbench with full-functionality. The resulting language is called Continuous Stream Language (CSL).
2) SMM integrates a rich library of data stream mining algorithms that (a) are fast and light as required for online mining, (b) dovetail with the constructs and mechanisms (windows, slides, etc.) efficiently supported by SMM, and (c) are generic, i.e., they apply to data streams with different number of columns and attribute types.
3) SMM is the first DSMS that provides load shedding even for User-Defined Aggregates. In fact, using our declarative syntax and our simple load shedding API, all mining tasks can benefit from a mining-aware load shedding engine that further improves the quality of mining results
4) Finally, SMM supports complete, end-to-end mining process specified as a continuous mining work-flow. The analysts can freely define them using the Mining Model Definition Language (MMDL) proposed in this paper or a GUI. Users can invoke these mining algorithms via a simple and uniform syntax.

Paper organization. In this paper, we illustrate the techniques and architecture of SMM via a succession of case studies focusing on the integration of key mining methods for classification, association, and ensemble-based learners. These examples illustrate: (i) how to use the basic SQL-based constructs of SMM to support mining tasks, (ii) how to define new mining tasks and models to enrich the mining library of the system, and (iii) how existing, off-the-shelf mining algorithms, such as C4.5 decision trees, SWIM[35], etc., can be imported into SMM .

The rest of this paper is organized as follows. Section II describes the SQL extensions required for expressing complex mining algorithms. In Section III, we discuss ensemble based methods, MMDL, and the integration of mining algorithms as work-flows. In Section V, we use MMDL constructs to model the complete mining process for association rule mining. Section VI studies the performance of the proposed framework. Section VII covers further extensions that we have successfully developed on top of SMM , followed by the related work and conclusion.

II. Continuous Mining in SQL

The difficulty that SQL-based systems encounter in expressing and efficiently supporting complex data mining algorithms, such as Apriori, is well known and documented [38]. But intertwined with this *expressive power* problem, we find 2 other issues when mining data streams, (i) *genericity* and (ii) support for *non-blocking* queries. To illustrate these additional

[1] Sarawagi et al. [38], attempted to implement frequent itemset mining in DB2 without extending DB2. As discussed in the paper, this proved very difficult to implement and the performance was worse than that obtained with the cache-mining approach.

issues we use the example of Naive Bayesian Classification (NBC), which is among the very few data mining tasks that can be expressed in SQL.

Let us consider the situation where we want to train a classifier on the minexample of Table I.

TABLE I
A MINI TRAINING SET FROM THE PLAYTENNIS EXAMPLE

TrainTbl	**Outlook**	**Temp**	**Humidity**	**Wind**	**Dec**
	Sunny	Hot	High	Weak	No
	Sunny	Hot	High	Strong	Yes
	Overcast	Hot	High	Weak	Yes

In order to predict the Dec value (decision to play or not) of a new tuple $X = \langle x_1, \ldots, x_n \rangle$, the NBC classifier compares the probabilities of the possible values C for column Dec using Equation 1 (obtained assuming independent probabilities).

$$p(C|X) = \frac{p(x_1|C) \times \ldots \times p(x_n|C)}{p(X)} \times p(C) \qquad (1)$$

To derive the prediction C with max probability, we only need to count the number of training tuples for (i) each C value, and (ii) each (x_i, C) value pair. However, since training tables can have hundreds of columns, applying the SQL count aggregate on each column is cumbersome, and inefficient since it requires many passes through the data. Moreover, we want to write generic NBC queries, i.e., queries that work on tables with arbitrary number of columns, as needed for inclusion in a mining library. A simple answer to these requirements is illustrated by Table II and Table III. The first table, namely DescrptrTbl, summarizes the counts of the various occurrences that will be used in the prediction. Conceptually, we can think of the DescrptrTbl as the classifier that we have learnt.

Table III instead outlines a simple intermediate representation that can be used to compute Table II from Table I in a generic fashion. Thus, we use verticalization to write generic algorithms that can be applied to streams independent of their schema. Genericity is required in any data mining workbench: for instance Weka represents data as an array of doubles (a kind of verticalization) to enable generic algorithms. SMM provides a built-in table function called verticalize to produce a view representing Table II as shown in Example 1. For instance, if presented with the tuple $\langle \texttt{Sunny}, \texttt{Hot}, \texttt{High}, \texttt{Weak}, \texttt{All} \rangle$, verticalize would return the tuples: $\langle 1, \texttt{Sunny} \rangle$, $\langle 2, \texttt{Hot} \rangle$, $\langle 3, \texttt{High} \rangle$, $\langle 4$, $\texttt{Weak} \rangle$, and $\langle 5$, $\texttt{All} \rangle$. Thus once the input table is conceptualized in this column-oriented representation, we can proceed with the count of the 'Yes' and 'No,' grouped by the **(Col,Val)** pair, as shown in Example 2. Also observe that the total counts of 'Yes' and 'No,' is realized by simply using an additional 'All' column in **verticalize**. We note that columns that are not required for the mining process can be projected out first, before verticalization, to avoid any overhead associated with it.

TABLE II
THE **DescrptrTbl** FOR **TrainTbl**

DescrptrTbl	**Col**	**Val**	**Dec**	**Total**
	1	Sunny	No	1
	1	Sunny	Yes	1
	1	Overcast	Yes	1
	2	Hot	No	1
	2	Hot	Yes	2
	3	High	No	1
	3	High	Yes	2
	4	Weak	No	1
	4	Weak	Yes	1
	4	Strong	Yes	1
	5	All	No	1
	5	All	Yes	2

TABLE III
VERTICALIZED VIEW OF **TrainTbl**

TrainTbl	**Col**	**Val**	**Dec**
	1	Sunny	No
	2	Hot	No
	3	High	No
	4	Weak	No
	5	All	No
	1	Sunny	Yes
	2	Hot	Yes
	3	High	Yes
	4	Strong	Yes
	5	All	Yes
	1	Overcast	yes
	2	Hot	yes
	3	High	yes
	4	Weak	yes
	5	All	Yes

Example 1: Column Oriented view over TrainTbl

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

Example 2: Counting over ColumView

```
CREATE VIEW DescrptrTbl AS
SELECT Col, Val, Dec, count(*)
FROM ColumView
GROUP BY Col, Val, Dec
```

The benefits of this column-oriented representation become even more obvious when we turn to the prediction problem. Let us assume that the tuples to be classified are contained in a table TestTuples having the same columns as TrainTbl, except that the Dec column is missing and we instead have an ID column which uniquely identifies the tuples in the table (e.g., by a sequence number). Then, the extraction of the correct statistics for each tuple can be implemented by a natural join of the table TrainTbl with the DescrptrTbl. Thus, for each tuple ID, we multiply the corresponding 'Yes' and 'No' entries for each of its categorical attributes (including for 'All'). Multiplications can be performed efficiently by simply using the SQL sum aggregate on the *logs* of these positive numbers. Thus, the following SQL query defines a view, namely TestStats, obtained by joining the verticalized tuples of TestTuples with the DescrptrTbl.

Example 3: Deriving Statistics for Each Tuple

```
CREATE VIEW TestStats(ID, Val, Dec, LgT) AS
SELECT d.ID, d.Dec, log(d.Total)
FROM TestTuples AS t , DescrptrTbl AS d,
     TABLE(verticalize(t.Outlook, t.Temp, t.Humidity,
                    t.Wind, 'All')) AS ts(Col,Val)
WHERE d.Val=ts.Val AND d.Col=ts.Col
```

On this view, we implement Equation 1 by simply summing the logs for the totals of (x_i, Dec) minus the log for the $(\text{`}All\text{`}, Dec)$ counts[2] and then taking the max among the resulting values. Due to space limitations, we omit the simple SQL code for this computation.

The NBC discussed above can be extended for data streams by using non-blocking aggregates. However, more complex mining tasks cannot be expressed with SQL easily and require user-defined functions. These complex mining tasks rely heavily on BLOBS, CLOBS and complicated aggregations/summarization, which make scalar functions much less suitable to express incremental continuous computations than user-defined aggregates. Indeed, DSMS research has demonstrated aggregates applied through constructs such as *windows*, *slides*, and *tumbles* are critical in the computation of non-blocking queries on data streams, particularly for analytics and data mining tasks [17], [6], [8]. While most DSMSs only support these window constructs on buil-in aggregates, SMM supports them on arbitrary UDAs, which can be defined using INITIALIZE, ITERATE, TERMINATE, and EXPIRE actions [8]. While blocking UDAs specify some TERMINATE actions, window aggregates are non-blocking and specified using EXPIRE actions. Window aggregates can be freely applied over data streams [8].

Before we discuss these properties with the help of examples, let us observe that UDAs might be the solution of choice even in situations where they are not strictly required for expressivity. For instance, a fully-functional NBC should also incorporate support for (i) continuous values (e.g., by assuming some Gaussian distribution), (ii) various treatments of null values in the training tables, and (iii) Laplace estimators to compensate for missing combinations in the training set. While each of these additions can be expressed in SQL, the whole resulting code would be unwieldy, and thus its encapsulation into a UDA is much preferred in practice.

A. Data Streams

Let us say that the stream of input tuples to be classified is TestStream declared using the CSL statement of Example 4 below. Since tuples are classified individually as they arrive, we no longer need the explicit ID used in TestTuples. The definition of a SOURCE clause at the end of the statement denotes the port on which the data arrives[3]. As a result of this, SMM creates a CSV wrapper to ingest data arriving at this port (A user can also provide his/her own wrapper to read data of a different format).

Example 4: The Input Stream

```
CREATE STREAM TestStream (Outlook CHAR(10),
    Temp CHAR(10), Humidity CHAR(10),
    Wind CHAR(10), Dec CHAR(10))
            SOURCE 'port5444';
```

The next CSL statement on this incoming stream is that of Example 5, that performs a verticalization operation, similar to that of Example 1. By comparing these two examples, we see that the constructs used in CSL queries are basically those of SQL, only the create-view construct is replaced with the create-stream construct.

Example 5: Vertical view of TestStream

```
CREATE STREAM VertTestStream
SELECT ts.Col, ts.Val, t.Dec FROM TestStream as t,
    TABLE(verticalize(Outlook, Temp, Humidity, Wind ))
                    AS ts (Col, Val)
```

Since each incoming tuple in TestStream generates a group of four successive tuples in VertTestStream, the prediction task will be computed on 'tumble' window of size 4. For the 'tumble' effect, we will specify a window of size 4 (ROWS 3 PRECEDING plus the current one), and a slide of the same size: SLIDE 4. Thus, the incoming data stream is broken into chuncks of 4 consecutive tuples, called 'tumbles'. On this, we call ClassifyNaiveBayesian as follows:

Example 6: NBC Predictor Calls on windows and slides

```
CREATE STREAM Predictions AS
SELECT ClassifyNaiveBayesian(Col, Val)
    OVER (ROWS 3 PRECEDING SLIDE 4)
FROM VerticalTestStream
```

SMM uses special execution strategy for tumbles. Since each tumble's (consecutive chunck of 4 tuples) is disjoint, SMM simply executes the base version of the UDA over each 'tumble', i.e., starting with the first tuple of the 'tumble' and returning values upon receiving the last tuple of the 'tumble' (rather than the last tuple in the input, which would cause a blocking behavior) and repeating this procedure at the next input tuple [8]. Therefore ClassifyNaiveBayesian is defined as a base aggregate through the three INITIALIZE, ITERATE, and TERMINATE states. The computation in these states can be specified either in C++ or Java, or natively in SQL itself. With natively defined UDAs, SQL becomes a Turing complete and NonBlocking-complete language [30], enabling compact and declarative definition of complex mining algorithms, and rapid prototyping of applications. SMM is (at the best of our knowledge) the only DSMS that supports this capability. An important advantage of this approach (over UDAs defined in an external language) is the ability of accessing not only tables that are internal to the UDA (such as pred in Example 7) but also external tables, such as DescrptrTbl. Thus, in Example 7 the pertinent values of probability are selected from DescrptrTbl in the INITIALIZE state, and then their logs are added in the ITERATE state (with a Laplace estimator applied), and finally the most probable decision is selected in TERMINATE.

Example 7: NBC Prediction Aggregate

```
AGGREGATE ClassifyNaiveBayesian
        (column INT, cvalue CHAR(10)): CHAR(10)
```

[2] this second log must be added back once at the end, as per $\times p(C)$ in Equation 1.

[3] In CSL , this stream declaration can also contain an ORDER BY clause, to specify a timestamp based order of the tuples.

```
{ TABLE pred(pdec INT, ptot REAL);
  INITIALIZE: {
     INSERT INTO pred
        (SELECT dt.dec, 0
        FROM DescrptrTbl AS dt
        WHERE dt.Col = column AND dt.Val = cvalue ;
       UPDATE pred SET pdec= pdec +
        (SELECT log(1+dt.Total)
        FROM DescrptrTbl AS dt
        WHERE dt.Val = cvalue
        AND dt.Col = column ANDdt.Dec = pdec); }
ITERATE: {
  UPDATE pred p SET ptot = ptot -
       (SELECT log(dt.Total+1))
       FROM DescrptrTbl AS dt
       WHERE dt.Val = ‘All‘
        AND dt.Col = column AND Dec = pdec); }
  UPDATE pred SET ptot = ptot +
       (SELECT log(dt.Total+1))
       FROM DescrptrTbl AS dt
       WHERE dt.Val = cval
        AND dt.Col = column AND dt.Dec = pdec);
TERMINATE: {
  INSERT INTO RETURN
     SELECT m.pdec FROM pred AS m
     WHERE NOT EXIST (
        SELECT * FROM pred AS s
        WHERE s.ptot > m.ptot
         OR (s.ptot = m.tot AND s.pdec < m.pdec));
} }
```

Note that in the above example we assumed that the DescrptrTbl was somehow already precomputed. Let us say rather than using a precomputed table, we want to regenerate the table DescrptrTbl continuously from new incoming tuples as to account for concept shift or drift. We then train our classifier on a stream of new training tuples that have been verticalized into a VertTrainStream using CSL statements that, modulo the addition of the column Dec, are basically identical to those of Examples 4 and 5. On this incoming VertTrainStream, we now apply the UDA LearnNaiveBayesian, which we define in Example 9, over N most recent tuples, where N can be defined in terms of count or time. In the example below we use $N = 50000$ tuples.

Example 8: Training on the last 50000 samples

```
CREATE VIEW NBClearn AS
SELECT LearnNaiveBayesian(Col, Val, 4, t.Dec)
     OVER (RANGE 50000 R0WS PRECEDING)
FROM VerticalTrainStream
```

The NBC learner is defined as a windowed UDA in Example 9, below; this UDA basically computes the count statistics for the incoming tuples and then writes them into the external table DescrptrTbl (which for performance reasons should be defined as an in-memory table, an option supported in SMM).

Example 9: 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 values 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;
     INSERT INTO tupleSummary VALUES (oldest().col,
        oldest().val, classVal, -1) WHERE SQLCODE <> 0;
   }
}
```

Being a windowed UDA LearnNaiveBayesian has to take into account the tuples leaving the window, besides the incoming tuples. Therefore, in its definition, the EXPIRE state replaces the TERMINATE state of base aggregates. The EXPIRE state is executed once for each expiring tuple. The system automatically determines the tuples that have expired, 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 updates the DescrptrTbl. Note, EXPIRE state performs delta computation of statistics for tuples expiring out of the window in similar manner. Thus, windowed UDAs allow the users to specify incremental computation over incoming data streams.

Updating the DescrptrTbl 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 the SLIDE construct, that can be used to divide-up the window into panes [17], [6], [8]. Therefore, the following CSL query updates the DescrptrTbl every 1000 tuples.

Example 10: Using slides to pace the training steps

```
SELECT LearnNaiveBayesian(Col, Val, Dec)
     OVER (R0WS 50000 PRECEDING SLIDE 1000)
FROM vertstream
```

The *updateDescrptrTbl* UDA, invoked in Example 9, should only be invoked at the end of each slide.

Thus, for windowed UDAs, the user can specify the actions to be taken when a tuple arrives/expires, and also when a *slide* of tuples expire. This supports delta computation on windows and slides, and delivers great flexibility and efficiency.

In summary, even for the few data mining tasks, such as NBC, where in theory SQL has sufficient expressive power, in practice UDAs are required for (i) the encapsulation and modularization needed to manage the code, (ii) the flexibility of using the various kinds of windows that all DSMSs have recognized as essential for data stream processing, and (iii) the many optimizations supported by CSL for these constructs. Furthermore, for more complex mining tasks UDAs become the sine-qua-non that provides the expressive power necessary to implement them.

III. Mining Models and Mining Flows

As shown in [42], expressive UDAs supported by SMM can be used to implement many ad hoc mining algorithms. However for user convenience, it is essential to provide a uniform interface for defining and invoking these user defined mining algorithms. Therefore, we propose the Mining Model Definition Language (MMDL). Syntax for MMDL is formally defined in Section IV-A. Here, we continue our running example to define a model for NBC in MMDL, as in Example 11.

Example 11: ModelType for Naïve Bayesian Classifier

```
CREATE MODEL TYPE NaiveBayesianClassifier {
    SHAREDTABLES (DescrptrTbl),
    Learn (UDA LearnNaiveBayesian,
        WINDOW TRUE,
        PARTABLES(),
        PARAMETERS()
      ),
    Classify (UDA ClassifyNaiveBayesian,
        WINDOW TRUE,
        PARTABLES(),
        PARAMETERS()
      )
};
```

The NBC mining model comprises the two tasks: *Learn* and *Classify*. Thus, the model definition specifies the UDAs associated with these tasks, e.g., UDAs of Examples 9 and 7, for *Learn* and *Classify* tasks, respectively. and the parameters associated with each task (the PARAMETERS clause), and where to store them (the PARTABLES clause). Storing the parameters in these 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 V. Finally, we note that both UDAs share the DescriptorTbl, where the *Learn* task stores the statistics, which will be used by the *Classify* task at the time of prediction. Thus, a mining model defined in SMM can have multiple tasks that are associated with it and these tasks may share one or more tables.

To complete our NBC example next we show how we invoke the mining model created above. First we create a mining model instance of NBC and then we invoke the Learn and the Classify tasks on the training and the testing stream, respectively.

Example 12: Bagging Ensemble Instance

```
CREATE MODEL NBCInstance
                    INSTANCE OF NaiveBayesianClassifier;
RUN NBCInstance.Learn ON myTrainStream
        USING window = ROWS 50000;
RUN NBCInstance.Classify ON myTestStream
        USING window = ROWS 3, slide = 4;
```

Note, that the MMDL statements of Example 12 are the same as the CSL queries of Examples 8 and 6, with the same window and slide specification.

Thus, all MMDL statements are converted to equivalent CSL queries and executed within SMM , as we discuss in Section IV-B in detail. Next, we discuss implementation of a more advanced mining algorithm within this framework.

A. Ensemble Methods

The UDA of Example 9 maintains an NBC over a window of tuples from the training stream. However, it is very difficult to determine the optimal size of the window (to get the best accuracy in classification), since the data characteristics may change with time. Therefore, instead of maintaining a single classifier, the user may want a more complex solution (e.g., [43]) and maintain several small classifiers. This approach, also assures a better adaptation in the presence of concept-shift and concept-drift, since new classifiers can be continuously trained based on the latest statistics, while older or inaccurate classifiers can be retired. Thus, ensemble based methods increase the accuracy of classifiers in presence of concept-drifts and shifts [43], [11], [21]. Therefore, a DSMS workbench, must support such advanced mining methods. Here, we discuss ensemble-based weighted bagging in detail below. However, other methods can be similarly integrated into SMM through its extensible framework.

Ensemble Based Weighted Bagging: Ensemble based weighted bagging was proposed in [43] 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 the incoming training stream into disjoint blocks of data and learns a new classifier for each block. Therefore, we can use a base UDA with tumbling windows to learn a new classifier for each block of data. Then we will have an ensemble of learned classifiers, one for each recent training 'tumble'. Besides training new classifiers, the recent training tuples are also used to estimate the accuracy of the current ensemble of classifiers—i.e., those that have been previously derived from the training 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 classifiers from the ensemble. Then, a weighted voting scheme is employed to determine the final classification of the test tuples. Figure 1 shows this process pictorially.

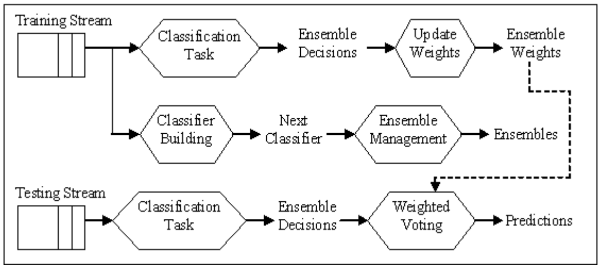

Fig. 1. Generalized Weighted Bagging

In Figure 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 9 is one implementation of 'Classifier Building' UDA for NBC. Similarly, the 'Classification Task' UDA predicts the class of each tuple based on each classifier in the ensemble. For instance, Example 7 is one implementation of 'Classification Task' UDA for NBC. Thus, the general flow and processing of data tuples, depicted in Figure 1, 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 being used. Thus, any classification algorithm (NBC, decision tree, nearest neighbor, etc.), can be introduced into SMM for ensemble based weighted bagging by implementing a 'Classifier Building' UDA and a 'Classification Task' UDA. Therefore, both built-in and arbitrary user-defined classification algorithms can take advantage of this advanced technique without having to re-implement it.

However, we note that specifying the various steps required for weighted bagging represents a daunting task for analysts and less experienced users. This is also true for many other advanced mining processes, that require a step by step invocation of several mining tasks [42]. Therefore, MMDL supports specification of one or more mining flows within the mining model definition. These complex mining processes only have to be specified once during model definition can be reused by all users. Consider for instance Example 13. This example shows the mining model definition 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 as discussed in [42]. Furthermore, Example 13 defines two complex mining flows at the end, one for training and another for testing, which invoke the other mining tasks defined in the model. Note, these flows essentially model the data flow as presented in Figure 1. While these flows are expressed as a series of CSL statements below, SMM also provides a GUI to define these work-flows pictorially as we discuss in the next section.

IV. Mining Model Flows via GUI

SMM supports a 'box&arrow' based GUI to enable definition of complex mining flows, such as that of Example 13. Due to space limitations, we only give a brief introduction to this GUI. Figure 2, shows the definition of two simple workflows for learning and classifying using NBC, where the training and testing streams are first fed to a UDA for verticalization and then fed to the appropriate UDA for learning/classification. The user can define arbitrary workflows by dropping different entities and connecting them with arrows. SMM then builds a mining model flow from this GUI.

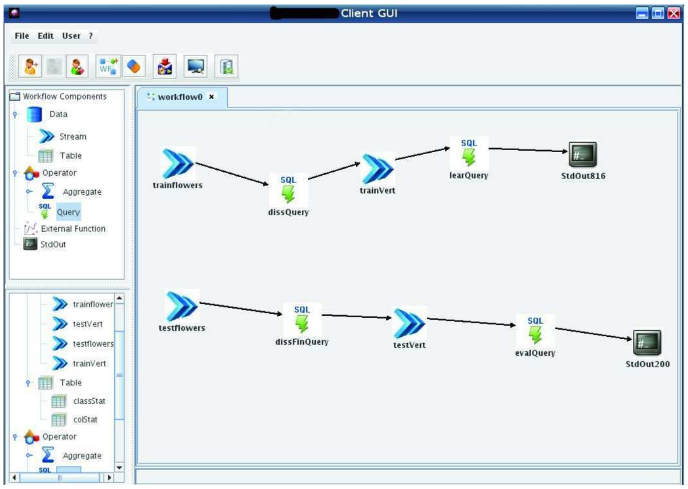

Fig. 2. SMM Client GUI

Example 13: Ensemble Based Bagging

```
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, ... ),
    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)
}
```

A. Definition of MMDL

In this section, we present the formal syntax of MMDL as supported in SMM [34]. The language allows definition of new mining models, which can be instantiated and invoked 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

TABLE IV
MMDL SYNTAX

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

its mining tasks, e.g. ensemble based methods, as discussed previously, to define end-to-end mining process. Thus users can integrate new mining algorithms in the system by defining a mining model and providing the implementation of the underlying methods as UDAs. Users can optionally define mining flows for better usability of their methods.

The formal syntax of MMDL is given in Table IV. Each model must have at least one task. Users must specify a UDA associated with each task and the parameters that it accepts. Users may also specify zero or more tables that can be shared by the tasks of the mining model.

Furthermore, MMDL allows the data mining experts to define zero or more mining flows for complex mining processes, as seen in Example 13. Once a mining model is defined, 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 built-in and user-defined tasks is given in Table V.

Example 12 showed how we use this syntax to instantiate an NBC mining model and execute the mining tasks. The with clause specifies the input stream and the using clause specifies the arguments that mining task takes.

B. Compilation of MMDL

When the user specifies a new mining model, SMM stores the mining model details in the data dictionary. Then, when the user creates an instance of the mining model, SMM will store an entry in the data dictionary for the instance and also create all the shared tables. These shared tables will have special names that tie them to the instance. Thus, two instances of the same mining model do not conflict with each other. After the user has created an instance he can decide to invoke one of the mining tasks or flows. If the user decides to invoke one of the tasks, then internally SMM invokes the corresponding aggregate over the specified stream with the given parameters (including window and slide). Note, that the SMM system is intelligent in that this invocation of the aggregate will only modify the tables that are corresponding to current mining model instance. The output of this invocation can be diverted to the user or into another stream. If the user decides to invoke one of the mining flows then SMM executes each statement in the flow definition. Note these statements will be concurrently running and they may in turn invoke other mining tasks of the model. Also, note that the user can then also invoke other flows of the mining task concurrently. For example, the user may first start executing the training flow for a classification task and then invoke the testing flow. Thus, MMDL is integrated into SMM by compiling the mining models, tasks, and flows into CSL streams, queries, and UDAs. Next, we show how MMDL is used to support complete end-to-end association rule mining in SMM .

TABLE V
TASK INVOCATION SYNTAX

TaskInv	→	RUN ModelInstNm ‘.‘ TaskNm WITH TableStreamNm [USING ‘(‘ [ParamAssign]* ‘)‘]?
ParamAssign	→	ParamNm ‘=‘ Value

V. ASSOCIATION RULE MINING

Association rule mining represents an important mining method, which also has many online applications. The association mining process consists of many steps, starting with the difficult task of finding frequent patterns. Efficient pattern mining from data streams represents a difficult problem that has been the focus of much research work [16], [27], [35]. In particular, recently proposed SWIM algorithm [35] optimizes the use of window/slide constructs to achieve efficiency in an online setting. Thus, the 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 [27]) to mine each slide individually. Here each window is composed of k slides, where k can be anywhere from 1K to 1M. 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 accordingly, to maintain the union of all candidate patterns, in an incremental fashion. In general, other existing algorithms for frequent pattern mining can also be easily integrated in the SMM framework.

Association Rule PostMining. The straightforward derivation of rules from frequent pattern tends to produce a large number of rules. Thus, many post-processing techniques have been proposed to summarize, analyze, mine and rank the rules found from the mining process [32]. 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 algorithms get increasingly complex, due to the required structure of the data [40], which in turn complicates the integration of other post-processing techniques. Instead, Calders et al. [9], propose an approach to incorporate association rule mining in relational databases through virtual mining views. 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 *confidence* thresholds, these also include specification of high (low) importance items that should always (never) be reported. Therefore, Calders et al. [9] propose a 3-table view of the discovered association rules as shown below.

```
Sets(sid int, item int);
Supports(sid int, supp real);
Rules(rid int, sida int, sidc int, sid int, conf int);
```

The Sets table stores the frequent patterns by their id and items. The Supports table stores the support of the frequent patterns by their id. Finally, the Rules table list the confidence of each rule by storing its rule id (**rid**), its antecedent itemset (**sida**), its consequent itemset (**cida**), and the union of the two (**sid**). This framework is easily extended to work with data streams, and to integrate different post-mining techniques as we discuss next.

First, we define a mining model in MMDL that provides similar view over discovered patterns and rules. The definition of an association-rule mining model, and a sample instance are given in Example 14. Of course, the users can modify and/or extend this mining model to derive new mining models.

Example 14: Association Rule Mining Model

```
CREATE MODELTYPE AssociationRuleMiner {
    SHAREDTABLES (Sets, RulesHistory),
    FrequentItemsets (UDA FindFrequentItemsets,
        WINDOW TRUE,
        PARTABLES(FreqParams),
        PARAMETERS(sup Int, uninterestingItems List,
          rejectedPats List, acceptedPats List, ...)
    ),
    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 AssocRuleMinerInstance
    NSTANCE_OF AssociationRuleMiner;
RUN AssocRuleMinerInstance.ARMFlow ON Transactions
      USING sup > 10, window = ROWS 1M, slide = 100K;
```

In Example 14, association rule mining is decomposed into four sub-tasks, namely FrequentItemsets, AssociationRule, PruneSummarizeRules, and *MatchWithPastRules*. As previously discussed, these are implemented as UDAs. 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 (Example 14). This flow is also shown in detail in Figure 3.

The first task invoked in our example, is 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 results of this frequent patterns algorithm are inserted into the **FrequentPatterns** stream, denoted by the CREATE STREAM construct. The **FrequenPatterns** stream is in fact the same as the Supports stream as proposed in Calders et al. [9]. The frequent patterns algorithm must also update the **Sets** table.

The second step in our flow, finds association rules based on the results of the previous task. Any algorithm that generates

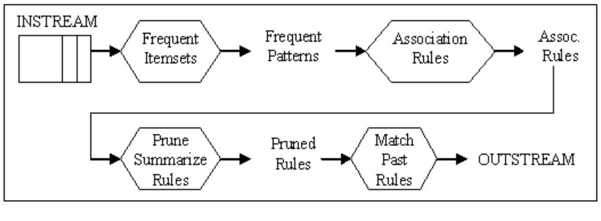

Fig. 3. Association Rule Mining: End-to-end flow

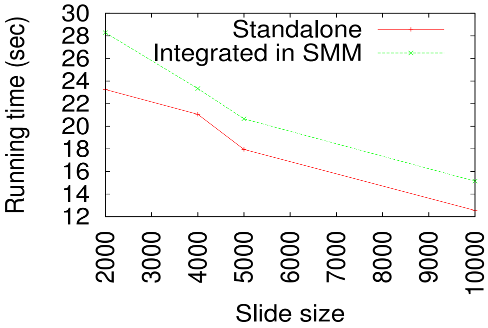

Fig. 4. Standalone SWIM vs. Integrated in SMM

rules based on frequent patterns can be used in this second step. The algorithm also takes confidence and correlation thresholds to prune the resulting association rules. This task and other tasks down stream 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.

The third step prunes/summarizes the association rules, which is an optional step. Association rule mining is an exhaustive method. Thus, it often produces a large number 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. [32] 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 the fourth and final, 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 tasks and MMDL allows declarative specification of mining tasks and flows connecting these tasks. Furthermore, the framework is extensible in that it allows easy integration of new mining algorithms.

VI. Performance

We will now discuss the results of four sets of experiments designed to evaluate the scalability, performance and robustness of SMM. We first evaluate the SMM's performance on a single mining query, using existing mining packages, such as C4.5 and Weka. Then we evaluate the ability of SMM to support multiple queries concurrently, when the system is not experiencing an overload situation. Finally, we discuss SMM ability to deal with overload situations through the advanced load shedding algorithms that have been integrated into the system. We decided to run our experiments on a single-processor machine, with a Pentium4, 2.4GHz processor, 1GB RAM, running under Linux. 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 load time.

A. *Running C4.5 and the Cost of Integration*

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 [3] to generate the test dataset. Figure 4 shows the results for minimum support 0.15%, window size 20K and varying slide size. From Figures 5–6, we see that SMM only has about 15-20% overhead over standalone SWIM. *C4.5 and Weka.* In this experiment, we compare SMM with C4.5 and Weka (J48) decision tree classifier on two real-world datasets. The first dataset is called Iris, which contains 4 real-valued attributes and a prediction attribute, which can take 3 different values [24]. The second dataset contains heart disease data with 13 attributes and a prediction attribute that takes 5 distinct values signifying risk of heart disease [24]. We have increased the dataset size by copying all the tuples multiple times. We compare the open source implementation of C4.5 decision tree classifier and the Weka decision tree classifier (J48) with the one integrated in SMM . SMM allows easy integration of such existing mining algorithms through UDAs defined in C/C++ or Java. The results for training and testing 1 million tuples are presented in Figures 5 and 6. Note to enable a direct comparison, for these experiments we do not apply any load shedding and the accuracy of the classifiers is the same. The better performance obtained by SMM and C4.5 over Weka and J.8 is not surprising considering the better performance of C++ over Java, and the modest run-time overhead incurred by running external algorithms in SMM . However, the fact that, we could take an off-the-shelf mining algorithm and include it in SMM (with minimal programming effort and performance overhead), is a clear demonstration of the extensibility of our system. In general, SMM provides an efficient framework that allows easy integration of new and existing mining algorithms,

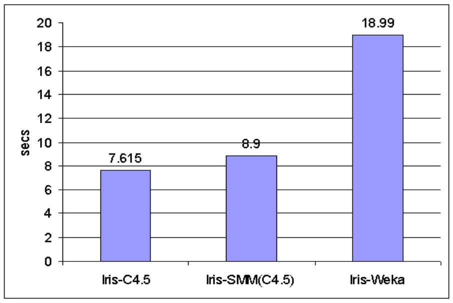

Fig. 5. C4.5 Decision Tree over Iris

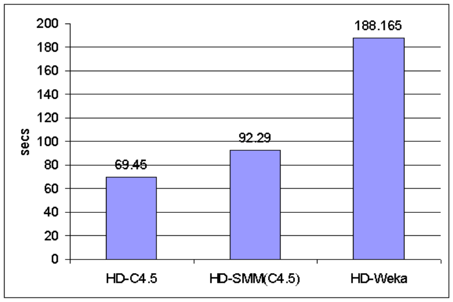

Fig. 6. C4.5 Decision Tree over Heart Disease

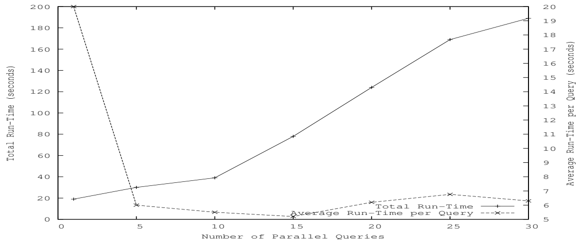

Fig. 7. SMM Scalability

and it is also conducive to the concurrent execution of multiple mining tasks, as we discuss next.

B. Concurrent Queries

In this experiment we evaluate the performance of SMM over multiple parallel queries. A total of thirty parallel queries were run. These queries are random queries that describe patterns over Nasdaq[1], RFID[2] and SIGMOD[3] datasets. We measured throughput (total output per unit time) as the number of queries changes from 1 to 30. As the number of queries changes from 1 to 15, we see the throughput increase from 14 to 20 *kbps*. Beyond 15 queries, the throughput becomes lower, and for 25 queries it return to be of 14 *kbps*. When multiple queries are running, resources are shared and thus a noticeable drop in average run time per query can be observed between a single and multiple query execution time.

Figure 7 depicts the total running time for all queries and the average response time for each query. Observe that the total running time grows slowly. Furthermore the average response time per query is almost constant (for number of queries > 1) and demonstrates the advantage of our parallel query architecture. In summary SMM is able to derive multiple queries concurrently with good performance and response time whenever the processor and memory load do not exceed the maximum capacity of the system. In overload situation SMM is able to maintain QoS by relying on its load shedding capabilities, which are discussed next.

C. Mining-Aware Load Shedding

Providing efficient support of load shedding for continuous queries has been a cornerstone for achieving QoS guarantees in SMM [41], [7], [36]. Furthermore, SMM is the first to extend this to data mining computations. The basic approach used by SMM is proposed in [7] where load-shedders are inserted into the query execution graph to sample the data streams as to ensure minimal drop in quality of various aggregates being computed. In [31] a generalization was proposed whereby complex mining aggregates besides simple aggregates could be supported. This result was further improved in [36] where an algorithm was proposed to achieve optimal load shedding, while accommodating different requirements for different users, query sensitivities, and penalty functions. This load shedding algorithm is fully integrated in SMM. Based on our experiments the overhead of the load shedding algorithm is around 2% even under the most extreme conditions, i.e. if the available Resource(R) is only 1/5th of the Load(L) on the system (R/L = 0.20). We refer the reader to [36] for detailed results of our experiments.

VII. Status and Further Extensions

Currently, the SMM supports all of the entities and constructs described in this paper, which includes data streams, continuous queries, complex UDAs (which can define arbitrary mining methods), mining models, and mining flows. Furthermore, the system integrates a library containing wide range of mining methods, such as NBC, decision tree classifier, linear regression, ensemble based methods, DBScan clustering, windowed K-means, SWIM, SQL-TS, K*SQL, etc. Thus, SMM supports integration of a wide range of built-in and user defined algorithms over data streams along with all the data stream related extensions, such as load shedding, window synopses, QoS, etc. *Sequences and Regular Expressions*

There has been much research interest in developing new tools and languages for querying massive collections of data, in order to discover patterns in click streams, RFID data processing, asset tracking, weather forecast, fraud detection, and financial data analysis. To this end, several CEP (Complex Event Processing) systems and patterns languages have been proposed [37], [25], [14], [18], [19]. Most pattern languages provide some constructs for certain subsets of regular expressions, which are also the cornerstone of the SQL:2003 extension proposal put forth by DBMS vendors and DSMS venture companies, called SQL Match-Recognize[45]. SMM supports the K*SQL language that (i) is strictly more powerful than these languages on relational data streams but (ii) it can also support queries on XML streams and nested words documents [5], [4]. This allows SMM to search for sequential patterns using regular expressions, including complex patterns in time series. It also makes it possible to support mining of streaming XML documents, an application of interest to many data mining researchers.

The integration of K*SQL with SMM is accomplished via a UDA function: its arguments include a schema, a pattern definition, a select clause and a where clause. The support of K*SQL is a farther indication of the power and extensibility of SMM and its SQL+UDA based architecture [5], [4].

VIII. Related Work

Related work on supporting data mining in DBMS and DSMS was discussed in the introduction. Here we focus on data stream mining algorithms.

On-line data stream mining has been the focus of many research efforts, and a recent review can be found in [22].

For instance, Ester et al. [15] 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 [23]. Additionally, researchers have focused on improving the accuracy of on-line classifiers using ensemble

[1] http://infochimps.org/dataset/stocks_yahoo_NASDAQ

[2] http://lahar.cs.washington.edu/content/Download/RFIDData/rfidData.html

[3] http://www.cs.washington.edu/research/xmldatasets/

based methods [43], [11], [21]. Furthermore, there has also been significant research in finding frequent patterns and association rules [44], including frequent pattern mining over over a stream of transactions. For instance, Chi et al. [10] propose the Moment algorithm, which is a differential algorithm for closed frequent patterns over continuous windows. On the other hand, Mozafari et al. [35] proposed the SWIM algorithm to maintain frequent patterns over large sliding windows.

IX. 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 provided by (i). Foremost among the technical challenges that prevented this integration, we find the SQL-based query languages used by most DSMSs. 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, SMM compares with the Weka paragon, in terms of open architecture and extensibility, while outperforming it in terms of scalability and performance. Furthermore, we note that while in this paper we have only shown the integration of a few existing data stream mining ideas, many other mining algorithms can be easily integrated in the SMM framework. SMM supports the addition of new mining algorithms by UDAs over windows/slides, the definition of mining models and mining flows via MMDL, and user-friendly GUI. Thus, the SMM system is significant because it has delivered (i) the first data stream mining workbench, and (ii) effective techniques and a general architecture to extend a DSMS with high-level functionality for very advanced application domains.

References

[1] IBM. DB2 Intelligent Miner http://www-306.ibm.com/software/data/iminer.

[2] ORACLE. Oracle Data Miner Release 10gr2http://www.oracle.com/technology/products/bi/odm.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, 1994.

[4] Barzan Mozafari andKai Zeng and Carlo Zaniolo. From regular expressions to nested words: Unifying languages and execution for relational and xml sequences. In *VLDB 2010: 36th Int. Conference on Very Large Data Bases*, 2010.

[5] Barzan Mozafari andKai Zeng and Carlo Zaniolo. K*sql: a unifying engine for sequence patterns and xml. In *SIGMOD Conference*, pages 1143–1146, 2010.

[6] A. Arasu, S. Babu, and J. Widom. Cql: A language for continuous queries over streams and relations. In *DBPL*, pages 1–19, 2003.

[7] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *ICDE*, pages 350–361, 2004.

[8] Y. Bai, H. Thakkar, C. Luo, H. Wang, and C. Zaniolo. A data stream language and system designed for power and extensibility. In *CIKM*, 2006.

[9] Toon Calders, Bart Goethals, and Adriana Prado. Integrating pattern mining in relational databases. In *PKDD*, 2006.

[10] Y. Chi, H. Wang, P. Yu, and R. Muntz. Moment: Maintaining closed frequent itemsets over a stream sliding window. In *ICDM*, 2004.

[11] F. Chu and C. Zaniolo. Fast and light boosting for adaptive mining of data streams. In *PAKDD*, volume 3056, 2004.

[12] Coral8, inc. (n.d.). *Home — Coral8, Inc.* Retrieved July 7, 2006, from http://www.coral8.com/, 2005.

[13] Weka 3: data mining with open source machine learning software in java. http://www.cs.waikato.ac.nz.

[14] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, 2007.

[15] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Michael Wimmer, and Xiaowei Xu. Incremental clustering for mining in a data warehousing environment. In *VLDB*, 1998.

[16] C. Jin et al. Dynamically maintaining frequent items over a data stream. In *CIKM*, 2003.

[17] D. Abadi et al. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, 2003.

[18] Mohamed H. Ali et. al. Microsoft cep server and online behavioral targeting. *PVLDB*, 2009.

[19] Nihal Dindar et. al. Dejavu: declarative pattern matching over live and archived streams of events. In *SIGMOD*, 2009.

[20] Sirish Chandrasekaran et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[21] George Forman. Tackling concept drift by temporal inductive transfer. In *SIGIR*, pages 252–259, 2006.

[22] João Gama, Jesús S. Aguilar-Ruiz, and Ralf Klinkenberg. Knowledge discovery from data streams. *Intell. Data Anal.*, 12(3):251–252, 2008.

[23] Joao Gama, Ricardo Rocha, and Pedro Medas. Accurate decision trees for mining high-speed data streams. In *KDD*, 2003.

[24] UCI Machine Learning Group. UCI Machine Learning Repository http://www.ics.uci.edu/ mlearn/mlsummary.html.

[25] Daniel Gyllstrom, Jagrati Agrawal, Yanlei Diao, and Neil Immerman. On supporting kleene closure over event streams. In *ICDE*, 2008.

[26] J. Han, Y. Fu, W. Wang, K. Koperski, and O. R. Zaiane. DMQL: A data mining query language for relational databases. In *DMKD*, Montreal, Canada, 1996.

[27] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, 2000.

[28] T. Imielinski and A. Virmani. MSQL: a query language for database mining. *Data Mining and Knowledge Discovery*, 1999.

[29] Tomasz Imielinski and Heikki Mannila. A database perspective on knowledge discovery. *Commun. ACM*, 39(11):58–64, 1996.

[30] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. Data models and query language for data streams. In *VLDB*, 2004.

[31] Yan-Nei Law and Carlo Zaniolo. Improving the accuracy of continuous aggregates and mining queries on data streams under load shedding. *IJBIDM*, 3(1):99–117, 2008.

[32] Bing Liu, Wynne Hsu, and Yiming Ma. Pruning and summarizing the discovered associations. In *KDD*, 1999.

[33] R. Meo, G. Psaila, and S. Ceri. A new SQL-like operator for mining association rules. In *VLDB*, Bombay, India, 1996.

[34] Introducing Stream Mill:. User-guide to the dsms, esl and smm. http://yellowstone.cs.ucla.edu/projects/index.php/.

[35] B. Mozafari, H. Thakkar, and C. Zaniolo. Verifying and mining frequent patterns from large windows over data streams. In *ICDE*, 2008.

[36] Barzan Mozafari and Carlo Zaniolo. Optimal load shedding with aggregates and mining queries. In *ICDE*, pages 76–88, 2010.

[37] Reza Sadri, Carlo Zaniolo, Amir Zarkesh, and Jafar Adibi. Optimization of sequence queries in database systems. In *PODS*, Santa Barbara, CA, May 2001.

[38] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *SIGMOD*, 1998.

[39] Streambase. *Complex Event Processing, Event Stream Processing, StreamBase Streaming Platform.* Retrieved July 4, 2006, from http://www.streambase.com/, 2005.

[40] Z. Tang and et al. Building data mining solutions with OLE DB for DM and XML analysis. *SIGMOD Record*, 34(2):80–85, 2005.

[41] N. Tatbul, U. Setintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, 2003.

[42] H. Thakkar, B. Mozafari, and C. Zaniolo. Designing an inductive data stream management system: the stream mill experiences. In *Scalable Stream Processing Systems*, 2008.

[43] H. Wang, W. Fan, P. S. Yu, and J. Han. Mining concept-drifting data streams using ensemble classifiers. In *SIGKDD*, 2003.

[44] Xiaoyu Wang, Hongyan Liu, and Jiawei Han. Finding frequent items in data streams using hierarchical information. In *SMC*, 2007.

[45] Fred Zemke, Andrew Witkowski, Mitch Cherniak, and Latha Colby. Pattern matching in sequences of rows. In *[sql change proposal, march 2007], http://asktom.oracle.com/tkyte/row-patternrecogniton-11-public.pdf http://www.sqlsnippets.com/en/topic-12162.html*, 2007.