Designing an Inductive Data Stream Management System: the Stream Mill Experience

Hetal Thakkar, Barzan Mozafari, and Carlo Zaniolo
University of California at Los Angeles
{hthakkar, barzan, zaniolo}@cs.ucla.edu

ABSTRACT
There has been much recent interest in on-line data mining. Existing mining algorithms designed for stored data are either not applicable or not effective on data streams, where real-time response is often needed and data characteristics change frequently. Therefore, researchers have been focusing on designing new and improved algorithms for on-line mining tasks, such as classification, clustering, frequent itemsets mining, pattern matching, etc. Relatively little attention has been paid to designing DSMSs, which facilitate and integrate the task of mining data streams—i.e., stream systems that provide Inductive functionalities analogous to those provided by Weka and MS OLE DB for stored data. In this paper, we propose the notion of an Inductive DSMS—a system that besides providing a rich library of inter-operable functions to support the whole mining process, also supports the essentials of DSMS, including optimization of continuous queries, load shedding, synoptic constructs, and non-stop computing. Ease-of-use and extensibility are additional desiderata for the proposed Inductive DSMS. We first review the many challenges involved in realizing such a system and then present our approach of extending the Stream Mill DSMS toward that goal. Our system features (i) a powerful query language where mining methods are expressed via aggregates for generic streams and arbitrary windows, (ii) a library of fast and light mining algorithms, and (iii) an architecture that makes it easy to customize and extend existing mining methods and introduce new ones.

1. INTRODUCTION
Data mining has received much attention in database community over the last decade [17, 18, 42, 4, 23]. Similarly, research on data streams has also received considerable interest [8, 30, 7, 20, 41, 6, 19]. On-line data stream mining represents the confluence of these two research areas and has recently received much attention [12, 18, 13, 43, 16, 23].

This interest is largely due to the growing set of streaming applications where mining plays a critical role; these include network traffic monitoring, web click-stream analysis, highway traffic congestion analysis, market basket data mining, credit card fraud detection, etc.

Introduction of the static data mining algorithms, in 1990s, had presented the research challenge of how to best support the large variety of mining algorithms in an integrated manner. An obvious solution in the case of static data mining was supporting these algorithms in a DBMS. While the problem drew much interest from vendors and researchers in mid-90s, effective solutions did not come quickly. Indeed performing data mining tasks using DBMS-provided constructs and functions, proved to be exceedingly difficult[1] [38]. Therefore, in their visionary paper [27], Imielinski and Mannila called for a quantum leap in the functionality and usability of DBMSs, whereby mining queries can be formulated with the same ease of use as most usual queries in a relational DBMS. The notion of Inductive DBMS (IDBMS) was thus born (a.k.a. the 'high-road' approach) [27], which inspired approaches such as MSQL [26], DMQL [24], and Mine Rule [33]. These approaches feature SQL-based mining languages to specify the data to be mined and the kind of patterns to be derived. Although these proposals have made a number of research contributions, they suffer from two main limitations, generality and performance. For instance, MSQL and Mine Rule only consider association rule mining.

Therefore, instead of taking the 'high-road' approach to inductive DBs, commercial DBMS vendors have answered users' demands by less ambitious approaches that are largely based on the addition of mining libraries to their DBMSs. For example, DB2 Intelligent Miner [4], Oracle Data Miner [5], and OLE DB for DM [42]. Also these libraries are often enhanced with graphical interfaces. However, all vendor proposed approaches are closed, i.e. provide little in terms of flexibility, extensibility, and integration with SQL. This lack of openness represents a significant weakness with respect to specialized data mining systems, such as Weka [14], which have gained wide acceptance in the field. Weka is an open source tool for machine learning and data mining, implemented in Java. The main advantages of Weka are as follows:

- All algorithms follow standard Java API for extensibility,
- A comprehensive set of data pre-processing (filtering) tools, and
- Built-in implementation for many learning algorithms for classification, clustering, and frequent itemsets mining.

Therefore, while static data is managed in DBMSs, it is traditionally mined using a cache-mining approach due to the lack of suitable constructs in DBMSs.

[1] Sarawagi et al [38], attempted to implement frequent itemset mining in DB2 without any extensions. As discussed in the paper, it was very difficult to implement and the efficiency was worse than the cache-mining approach.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT'08, March 25–30, 2008, Nantes, France.
Copyright 2008 ACM 978-1-59593-926-5/08/0003...$5.00.

On-line data stream mining raises many new problems that were not encountered in static data mining. For instance, changing data characteristics represent the first such problem, often known as concept shifts/drifts. Since the data is continuously arriving the data values may experience a change in the distribution, for example the mean and the variance for a temperature reading may vary as seasons change. Furthermore, the underlying concepts that generate the data may also change, for example in credit card fraud detection, the types of frauds may evolve over time. Thus, a classifier learned over stale credit card transactions may not be accurate in predicting behavior of current transactions. Therefore, we must continuously and actively learn the data mining model over the latest training data.

Another problem in on-line mining is the real-time response requirement of data streams, which severely limits the use of complex mining algorithms requiring multiple passes over the data. In fact, these applications may sacrifice the accuracy of the mining models in order to support the real-time constraints. Therefore, these new problems present two main research challenges.

i. Finding new algorithms that suit the requirements of the on-line applications.
ii. Building systems that efficiently support such algorithms in an integrated and extensible manner.

Recent years have seen the emergence of many on-line mining algorithms [12, 18, 13, 43, 16, 23], however many research problems are still unsolved in this area, e.g. better algorithms for frequent itemsets mining over sliding windows are still being proposed [34]. On the other hand, the second challenge of investigating systems that can efficiently support a wide variety of these algorithms, has not received much research attention. Therefore, this paper proposes an Inductive Data Stream Management System (IDSMS) that precisely addresses this issue.

There are two alternatives for building such a system that supports arbitrary on-line mining algorithms over data streams.

i. Extend an existing Data Stream Management System (DSMS) with on-line mining capability
ii. Extend a cache-mining system such as Weka to handle stream mining

The second approach advocates extending an existing static data mining package, such as Weka, to support on-line data mining. Database users can select this cache-mining approach because the DBMS essentials, such as recovery, concurrency control, and data independence, etc. are not requisites to the execution of mining functions required for data mining applications. Thus, it is simple for users to provide their own mining functions by either coding them in a procedural language or by using a mining library. However, the situation could not be more different for data streams, where DSMS essentials such as scheduling, response time/memory optimizations, support for synopses, QoS, non-stop continuous queries, etc. are required by all applications, including on-line mining applications. Thus, users would prefer to rely on the system, for these basic features, rather than having to provide them directly as part of their applications. Thus, any such on-line mining system will have to provide all stream related extensions, such as windows, slides, load shedding, etc. Furthermore, the pull-based architecture expected by the static data mining packages will have to be changed to the push-based architecture posed by data streams. Finally, the complex data mining algorithms supported by these static data mining systems might be of little use, since they are not fast and light enough to satisfy the real-time response requirements. Thus, the second approach is not attractive since it requires rebuilding a complete DSMS and yet, does not reduce the effort of implementing new on-line mining algorithms.

Therefore, we have selected the first approach and extended the Stream Mill DSMS to support the rich functionality of an Inductive DSMS. While Stream Mill provides a good platform (e.g., it supports a very powerful query language), the task remains a formidable one. To the best of our knowledge on-line data stream mining has not been attempted previously by other DSMS projects. Providing a full suite of well-integrated on-line mining functions represents only the first of these challenges. The second challenge consists in making these functions *generic* i.e., independent on the number and types of attributes in the incoming data streams. Advanced techniques for on-line mining, such as ensemble based boosting and bagging, also have to be supported generically, i.e. over arbitrary mining algorithms. The third problem is to support the efficient computation of these mining functions on different types of windows, including continuous windows, sliding windows, and tumbles. The fourth challenge is to ensure that the system remains open and extensible, so that users can modify and extend it for their own application classes (a property that is lacking in most commercial systems).

Therefore, the main contributions of this paper are as follows:

- To the best of our knowledge this is the first attempt to design and implement an Inductive DSMS.
- Efficient and robust techniques have been developed for integrating key data mining functions into a DSMS. These include classification and association rule mining.
- Constructs and techniques have been proposed to support flexible windows and *generic* implementations of the on-line mining functions. Techniques, such as ensemble-based bagging and boosting, have also been developed to tackle concept-drifts and shifts [43, 13, 23].
- An open architecture, which enables the declarative specification and integration of new mining algorithms into the IDSMS, has been developed and tested.

The organization of this paper is as follows. The next section discusses related work in the areas of Inductive DBMSs, on-line mining algorithms, and DSMSs. In Section 3, we take Naïve Bayesian classification as a sample mining task and present the extensions that are required to support on-line mining in a DSMS. In Section 4, we discuss the support for advanced mining methods in the extended Stream Mill system. Section 5 and Section 6 present future work and conclusion, respectively.

2. RELATED WORK

On-line data stream mining has been the focus of many research efforts. For instance, [12] presents, Moment, a differential algorithm to maintain closed frequent itemsets over continuous windows, whereas [34] proposes, SWIM, an algorithm to maintain frequent itemsets over large sliding windows. Similarly, [43] presents an ensemble based bagging technique to improve the accuracy of classifiers in the presence of concept-drifts and shifts. [13] and [23], present other similar techniques to improve the accuracy of on-line classifiers. [16] extends a static clustering algorithm, namely DBScan, to be continuous. Thus, on-line mining algorithms represent a vibrant area of current research.

On the other hand, DSMSs have also been introduced to support continuous applications. DSMSs provide in-built support for many advance techniques, such as buffering, approximate query answering, sampling, load shedding, scheduling, windows, slides, etc., to effectively manage data streams. STREAM is one such DSMS that uses an SQL-like language and focuses on techniques for window management [7]. Instead of extending SQL, the Aurora/Borealis project defines query plans via an attractive 'boxes and arrows' based graphical interface [19]. The TelegraphCQ [20] project proposes an SQL-like language with extended window constructs, including a low-level C/C++-like for-loop, aiming at supporting more general windows such as backward windows. This paper focuses on extending the Stream Mill system, which supports the Expressive Stream Language (ESL). ESL's expressive power is superior to other data stream query languages, both theoretically [30] and practically [9].

As discussed in Section 1, Inductive DBMSs have also been the focus on many research projects. Furthermore, DBMS vendors have also added mining libraries to their respective DBMSs to provide integrated support [4, 5, 42].

At the convergence of these three research areas, namely DSMSs, IDBMSs, and on-line mining algorithms, is the research on systems that support on-line mining algorithms (i.e. Inductive DSMSs), which has not been the focus of any research projects. Therefore, this paper proposes an IDSMS that generically supports data stream mining algorithms and allows adding new algorithms declaratively.

3. DATA STREAM SYSTEMS AND MINING

Our approach consists in building an IDSMS by integrating efficient stream mining algorithms into the Stream Mill DSMS. To achieve such an integration, however a number of language and system extensions need to be added to Stream Mill. We will next discuss these extensions using Naïve Bayesian Classifier(NBC) as an example, since (i) it represents a very important classifier frequently used in applications [40], and (ii) unlike other data mining algorithms, it is simple enough to be expressible in standard SQL [44] and thus provides an excellent vehicle for explanation.

Take for instance the Play-Tennis example of Table 1, where we want to predict the **Play** attribute ('Yes' or 'No') given a training set consisting of tuples similar to the ones shown in Table 1.

RID	Outlook	Temp	Humidity	Wind	Play
1	Sunny	Hot	High	Weak	No
2	Sunny	Hot	High	Strong	Yes
3	Overcast	Hot	High	Weak	Yes
...	...	...	...	..	...

Table 1: The relation PlayTennis

Building a Naïve Bayesian classifier from a table like this (containing the training tuples) only requires (i) the count of the number of tuples, (ii) the number of tuples belonging to each class, and (iii) for each (column, value) pair the number of tuples belonging to each class. Using the statistics so collected in this descriptive phase, we can now perform the predictive task of deciding whether 'Yes' or 'No' is more probable for a new incoming tuple where all the attribute values, but the classification, are known.

Genericity. While implementing such a classifier on a table with a given schema is simple in our IDSMS, we want to implement a *generic* NBC—i.e., one that will work on any given table or data stream with an arbitrary schema. Indeed, Genericity is an important property supported in systems such as Weka or OLE DB, where algorithms can be applied on tables with arbitrary schema.

The first step toward achieving this genericity in a DSMS, is table *verticalization*. Thus our training set is verticalized into column/value pairs whereby the first two tuples in Table 1 are now represented by the eight tuples shown in Table 2.

RID	Column	Value	Dec
1	1	Sunny	No
1	2	Hot	No
1	3	High	No
1	4	Weak	No
2	1	Sunny	Yes
2	2	Hot	Yes
2	3	High	Yes
2	4	Strong	Yes
...	...	...	...

Table 2: Verticalized PlayTennis relation

Furthermore, we can convert each attribute to a real value, much in the same way as Weka, so that the vertical tuples can be passed to a UDA. Conversion of attributes with type date, int, and real to real is trivial. For attributes of type string, nominal, and relational each possible value is assigned an index, which is stored as a real. Therefore, Weka creates a generic type (real) array for each input tuple. In the case of Stream Mill we create a similar array, but instead store it as a table, since Stream Mill is a relational system. For instance, verticalizing a tuple such as

```
DataTuple(a INT, b REAL, c INT, d TIMESTAMP)
```

we get the following tuples.

```
VerticalTuple(Column INT, Value REAL, TotColumns INT)
VerticalTuple(1, a, 4)
VerticalTuple(2, b, 4)
VerticalTuple(3, c, 4)
VerticalTuple(4, d, 4)
```

This simple example tuple contains 4 attributes, two integers, one real, and one timestamp. The resulting vertical tuples always have three attributes, first attribute is an integer, namely column number, which is self-explanatory. Second attribute is a real and acts like an entry in Weka real array. The third and final attribute is the number of columns in the dataset (mainly required for book-keeping). While this verticalization can be easily achieved in Stream Mill using user defined table functions [2], for user convenience we extend Stream Mill with a built-in function called verticalize, which takes an arbitrary number of arguments and verticalizes them based on a given configuration table.

Thus, from this vertical representation, the descriptive part of the NBC is implemented by simply counting the occurrences of Yes and No with a statement such as follows:

```
SELECT ts.Column, ts.Value, t.Dec, count(t.Dec) as Cnt
    FROM traningset AS t,
        TABLE(verticalize(Outlook, Temp, Humidity, Wind)) AS ts
```

[2]User defined functions represent a very useful SQL:2003 construct now supported by most DSMSs, which are similar to DB2 table functions [2, 1]

```
GROUP BY ts.Col, ts.Value, t.Dec
```

This query (with small modifications) results in the following descriptor table, where the last attribute, normCnt, is the normalized count (normalized with respect to the number of tuples with the same Dec value).

```
DescriptorTbl(Col: int, Value: int, Dec: int, normCnt: real)
```

Then, the predictive part of the Naïve Bayesian classifier is implemented using the results of the following query:

```
SELECT t.RID, d.Dec, sum(abs(log(normCnt)))
FROM DescriptorTbl AS d, testingset AS t,
   TABLE(verticalize(Outlook, Temp, Humidity, Wind)) AS ts
WHERE d.Val=ts.Value AND d.Col=ts.Column
     GROUP BY t.RID, d.Dec
```

For each tuple, the classification with the highest value for the above sum will be predicted. Thus, a Naïve Bayesian classifier can be implemented as a set of simple SQL queries.

UDAs. However, the situation is a bit different when either the training set or the testing set is streaming. Let's first discuss the case where the testing set is streaming. In this case, for each incoming testing tuple, we must apply the above query and determine the best classification. Stream Mill/ESL provides User Defined Aggregates(UDAs) specially for this purpose, i.e. ESL UDAs allow specifying arbitrary operations to be performed over each arriving tuple. In general, UDAs provide tremendous power and flexibility as discussed in [9][3]. ESL UDAs consist of two states, INITIALIZE (executed for the first tuple of the stream) and ITERATE (executed for each sub-sequent tuple). Furthermore, ESL allows definition of tables, which are shared by the INITIALIZE and the ITERATE states, i.e. these tables can store any information needed for subsequent execution of ITERATE statements. These shared tables also make UDAs highly suitable for state-based computations. Thus, given a statistics table **DescriptorTbl**, the UDA of Example 1 performs classification for each tuple.

EXAMPLE 1. *Defining Classification Aggregate*

```
AGGREGATE classify(col INT, val CHAR(10), totCols INT) : INT {
    TABLE tmp(column INT, value CHAR(10));
    TABLE pred(dec INT, tot REAL);
    INITIALIZE: ITERATE: {
        INSERT INTO tmp VALUES (col, val);
        INSERT INTO pred SELECT d.Dec, sum(abs(log(normCnt)))
            FROM DescriptorTbl AS d, tmp AS t
            WHERE col = totCols AND d.Val=t.value
                AND d.Col=t.column
            GROUP BY d.Dec;
        INSERT INTO RETURN SELECT dec FROM pred
            WHERE col = totCols
                AND tot = (SELECT max(tot) FROM pred);
        DELETE FROM tmp WHERE col = totCols;
        DELETE FROM pred WHERE col = totCols;
    }
}
```

The UDA of Example 1 buffers the vertical tuples of each original tuple (INSERT INTO tmp statement), till it receives the last vertical tuple of an original tuple, signified by *col = totCols*. Upon arrival of the last vertical tuple of an original tuple, it computes the probability of each possible *Dec* value (INSERT INTO pred statement) and outputs the *Dec* value with the highest probability (INSERT INTO RETURN statement). This procedure is repeated for each testing tuple. Both the *tmp* table and the *pred* table are emptied after the computation is completed, i.e. at the end of the ITERATE statement for the last vertical tuple. This UDA is invoked in the same way as built-in aggregates are in SQL, as shown below.

```
SELECT classify(ts.Column, ts.Value, ts.TotColumns)
FROM testingstream as t,
   TABLE(verticalize(Outlook, Temp, Humidity, Wind)) AS ts
```

While the solution proposed above effectively solves the genericity issue, it is prone to performance overhead. Since the ITERATE statement of the UDA will now be invoked for each vertical tuple; if a tuple contains a lot of attributes, this overhead can be significant (based on our experience with real-world datasets). Therefore, we extend Stream Mill with a special solution for aggregates that are written in external programing language. This second solution consists in creating temporary arrays that are only processed by objects outside the Stream Mill system, i.e. external functions and external UDAs. Thus, the Stream Mill system does not have to deal with array types (a daunting task given that Stream Mill is a relational DSMS). Thus, a built-in function is provided that takes an arbitrary number of arguments and creates an array of reals from it. This array is directly passed to an external UDA programmed in C/C++ or Java. Thus, the mining algorithm will be implemented as an external UDA. The main advantage of this second approach is that it results in an efficient implementation compared to the first approach.

The first approach suffers from verticalization overhead and has to process large number of 'vertical' tuples. However, the second approach is only suitable when the algorithm is implemented as an external UDA. Therefore, the users are likely to use the first verticalization approach with natively defined UDAs when they are testing and fine-tuning their algorithms. However, the users will switch to UDAs written in a lower level language, such as C, and use the second approach for verticalization, during deployment for better performance. Therefore, the extended Stream Mill system allows the user flexibility to choose between declarative (thus simpler) implementation and performance. Indeed, Stream Mill employs the second technique to implement many built-in mining algorithms, which can be generically applied over arbitrary data streams.

Windows. Coming back to our Play-Tennis example, let's now suppose that the training set is also arriving continuously. In many on-line mining applications this represents a realistic scenario. For instance, consider a loan approval application in a bank, where a loan request is approved or denied based on the customer's credit rating, age, income, etc. While many cases can be marked easily via a classification tool, some cases may require human experts. Thus, the system may use these human classified examples as training set to learn new trends and to improve the accuracy. Therefore, the training model will have to be relearned to adopt to these changes. In general, in a streaming environment the mining models should be learned on the latest training dataset to cope with the changes in the data, a.k.a concept drifts and shifts. In general, the mining model should be continuously relearned from the training stream. This requires that the system maintains a mining model that is based on N most recent training tuples (or tuples that arrived in past N time units). Therefore, upon arrival of a new tuple, the oldest tuple is discarded and a new model is learned based on the most recent N tuples. Thus, ESL UDAs allow a delta maintenance strategy, where by a tuple can be added/removed from the model. ESL allows the user to define a special EXPIRE state in the UDA,

[3]While we only discuss mining related features and advantages of UDAs here, [9] presents an in depth discussion of UDAs

which is invoked for each expiring tuple. This allows the user to take any actions required for an expiring tuple. This method works very well for Naïve Bayesian classifier, as seen in Example 2 [4]. Example 2, shows the *learn* UDA, which collects the statistics required for the Naïve Bayesian classifier, i.e. updates the DescriptorTbl table upon arrival/expiration of a tuple. In other words, this UDA continuously learns the mining model for an NBC. Note, in this case the learned model is updated at the arrival/expiration of each tuple. Such windows are called continuous windows, since the window moves one tuple at a time.

EXAMPLE 2. *Defining Windowed Learning Aggregate*

```
WINDOW AGGREGATE learn(col INT, val CHAR(10),
        dec INT):INT {
    INITIALIZE: ITERATE: {
        UPDATE DescriptorTbl SET normCnt = normCnt + 1
            WHERE Col = col AND Val = val AND Dec = dec;
        INSERT INTO DescriptorTbl VALUES (col, val, dec, 1)
            WHERE SQLCODE = 0;
    }
    EXPIRE: {
        UPDATE DescriptorTbl SET normCnt = normCnt - 1
            WHERE Col = oldest().col AND Val = oldest().val
                AND Dec = oldest().dec;
    }
}
```

Such windowed UDAs are invoked in the same way as OLAP aggregates are in SQL:2003, as shown below.

```
SELECT learn(ts.Column, ts.Value, t.dec)
        OVER (ROWS 1000 PRECEDING)
FROM trainingstream AS t,
    TABLE(verticalize(Outlook, Temp, Humidity, Wind)) AS ts
```

While this represents an effective solution to continuously learn the mining model, it may not work for some classifiers. For instance, maintaining a decision tree classifier differentially, is rather complex. In such cases another technique can be applied; this technique consists in learning a new classifier every N number of tuples (or every hour), maintaining M such recent classifiers, and voting among these recent classifiers to determine the class label for the tuples. This strategy can be realized by the ESL query given below.

```
SELECT learn(T.col, T.val, TS.dec)
        OVER (ROWS 999 PRECEDING SLIDE 1000)
FROM trainingstream TS,
    TABLE(verticalize(outlook, temp, humidity, wind)) AS T
```

In ESL such windows can be applied over arbitrary UDAs, not just built-in aggregates. The OVER clause defines the size of the window. Whereas the SLIDE clause defines interval of execution, i.e. the execution of the iterate state is bundled for SLIDE number of tuples (or for tuples that arrive in SLIDE time, if a time range is specified for SLIDE). When the slide size and the window size are the same as in this case, it is called a tumble, since each subsequent window is disjoint from the previous window. More discussion of different kinds of windows supported in ESL and their optimization can be found in [9]. Thus in this example a new classifier is learned every 1000 tuples and a few such recent classifiers can vote to determine the class value for a testing tuple. Therefore, ESL UDAs and windows over UDAs naturally support such advanced mining queries. However, in some cases it is difficult to determine the optimal size of the window and the number of classifiers to keep. Indeed, smaller window sizes can lead to over-fitting, whereas larger window sizes cannot detect concept-drifts/shifts in a timely manner. Therefore many advanced techniques for improving the accuracy of classifiers, in the presence of concept-drifts and shifts, have been proposed, e.g. ensemble based bagging [43] and boosting [13]. Such advanced methods are generically supported efficiently in Stream Mill, as discussed in Section 4.

[4]Note, we have omitted the normalization of counts in this example for clarity.

In general, ESL with UDAs can support many mining algorithms efficiently. Furthermore, windows and slides over arbitrary UDAs find natural applications in on-line data mining. Additionally, mining algorithms implemented as UDAs can be applied over arbitrary streams, since these algorithms can be made *generic* based on verticalization. Finally, ESL UDAs allow the user to easily introduce new mining algorithms and modify existing ones, since ESL UDAs can be written declaratively, i.e. in SQL itself. Therefore, ESL with UDAs provides an ideal platform to perform on-line data stream mining. Next, we discuss how advanced techniques, such as ensemble based bagging and other mining algorithms are integrated in this framework.

4. ON-LINE MINING ALGORITHMS

Many existing on-line mining algorithms are provided as built-in algorithms in Stream Mill as we discuss next.

4.1 Classification

Classification is the task of predicting certain attribute(s) of testing tuples based on other attributes of the tuple and a mining model learned from the training tuples. The assumption is that both the testing and the training tuples are generated from the same underling process. Classification is also known as supervised learning, since it requires a training set. NBC represents the simplest classification algorithm and was discussed in Section 3. Many other classification algorithms, such as decision tree classifier, k-nearest neighbor classifier, etc., can be efficiently integrated in the proposed framework. We discuss decision tree classifiers below, since they are more descriptive compared to NBCs.

4.1.1 Decision Tree Classifier

Decision tree classifier represents another simple type of classifiers. Generating a decision tree classifier however, is a significantly more complicated as compared to creating a simple Naïve Bayesian classifier. The process requires close inspection of data distribution, which may require multiple passes of the data. Therefore, it is expensive to continuously learn a decision tree model over an incoming training stream. Instead a windowed approach is more suitable, where a new mining model is learned every N number of tuples, where N is a user defined number. Thus, in the windowed approach a new decision tree is created for every window of N tuples using the following 3-step procedure, starting at the root node.

1. Compute the entropy of each column (S) using a formula such as that of Equation 1 [3] – many other formulas can also be used.

$$Entropy(S) = \sum_{i=0}^{c} p_i * log_2(p_i) \quad (1)$$

where p_i is the portion of instances in the dataset that take the ith value of the target attribute and c is the number of possible distinct values the attribute can attain

2. Pick the column with the least entropy (ties broken randomly). For each distinct value of the chosen column, create a new node and an edge connecting the current node to the new node.
3. Recursively invoke this procedure on all new nodes.

Note that the entropy computed in distinct recursive calls is different, since the tuples that qualify at each node are different, i.e. while computing entropy, only tuples that match partial assignments of ascendant nodes are considered. Also note that the above algorithm terminates, i) if all 'qualifying' tuples at the node are of the same class or ii) if all columns are already assigned values. In this case, the algorithm makes a probabilistic decision based on the 'qualifying' tuples. A decision tree generated in this manner can be stored in tables such as the following. These tables store traversal edges and leaf nodes, respectively.

```
TABLE traverse(src_node INT, column INT,
      value CHAR(10), dest_node INT)
TABLE accept_node(node INT, classValue INT)
```

The evaluation of such a decision tree also presents an interesting problem, since it is a graph traversal problem. This traversal is performed via a recursive UDA that that starts at the root node and traverses the tree based on the testing tuple values. Eventually, a leaf node is reached, which stores the classification (accept_node table above). This indeed is a very elegant solution as shown in [21].

Note that the approach presented above is a constrained version of the decision tree classifier. The decision tree presented above only handles equality-splits, i.e. less than and greater than based splits are not handled. However, in general a more flexible approach is required, which can be achieved as follows. First modify the traverse table as follows, where the comparison attribute describes the operator ($=, \leq, \geq, <, >$) that needs to be satisfied in order to follow the edge.

```
TABLE traverse(src_node INT, column INT,
      value CHAR(10), dest_node INT, comparison INT)
```

Furthermore, the entropy calculation needs to consider this new paradigm. The reader is referred to [29], which discusses how to calculate entropy for continuous variables. Finally, the recursive UDA that traverses the decision tree also needs to be modified to account for this change.

4.1.2 Concept-Drifts and Shifts

One of the core issues in on-line data mining is considered to be the change in the underlying data due to gradual or sudden concept changes of the external environment [15]. As discussed before, there are two types of changes that need to be considered, change in the data distribution and change in the underlying concepts that generate the data. In general, these changing data characteristics prohibit the use of existing static data mining algorithms. A simple solution, which is used in many current approaches, is to decay the weight of data tuples as they get old. As tuples get old their weight approaches 0 and they are discarded. However this simple approach creates the following dilemma, if the weight decay rate is low, old concepts are present during classification, thus the accuracy is reduced. If the decay rate is high, the classifier over-fits the latest set of training tuples. The problem has been studied in detail by [43, 13, 23] and ensemble based bagging and boosting are proposed to improve the accuracy of classifiers over data streams. These techniques effectively and efficiently cope with concept-shifts and drifts as shown in [43, 13, 23]. It is imperative that such techniques are supported in an on-line mining system to achieve higher accuracy in on-line mining. Stream Mill system generically supports both of these extensions as discussed next.

Ensemble Based Weighted Bagging: The ensemble based weighted bagging approach was proposed in [43]. The approach is applicable in the setting described earlier, i.e. when there are two parallel streams, a training stream and a testing stream and both of these streams are generated by the same underlying concepts. The approach divides incoming training data stream into blocks of data (called tumbling windows) and learns a new classifier for each window. Learning can be performed using any arbitrary classifier, such as NBC or decision tree classifier (called the base classifier). Thus, we have an ensemble of learned classifiers, one for each recent training window. The approach uses the latest training window to determine the accuracy of existing classifiers on the currently arriving testing data. 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 tuple. Figure 1 shows this process in detail.

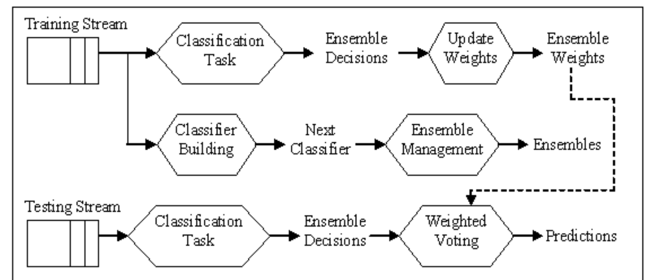

Figure 1: Generalized Weighted Bagging

As shown in Figure 1, the training stream is fed to a UDA, named 'Classifier Building', which learns the next classifier to be stored with other classifiers. The training stream is also sent to a 'Classification' UDA that predicts the classification of each tuple using each classifier in the ensemble. These predictions are then used to assign weights (based on accuracy) to the ensemble classifiers for the next set of testing tuples. The testing stream is fed to a 'Classification Task' UDA that predicts classification for each tuple based on each classifier in the ensemble. These predictions are then combined using weighted voting. Figure 1 also provides insights into how this ensemble based classification can be generalized for different classification algorithms. The general flow of data tuples, both training and testing, does not depend on the particular classification algorithm. In fact, only the boxes labeled 'classifier building' and 'classification task' are dependent on the particular classifier used. Thus, any classification algorithm that provides implementation for these two 'boxes' can be used as a base classifier for the weighted bagging approach. Of course, the 'boxes' should follow the API expected by the adjoining boxes. In Stream Mill, the boxes simply represent UDAs implemented in ESL or an external programming language. In general, built-in and arbitrary user defined classification algorithms can take advantage of weighted bagging without having to reimplement the technique. Thus, Stream Mill supports generic implementation of ensemble based weighted bagging.

Adaptive Boosting: Boosting in context of data stream classification was introduced in [13]. The assumption of two parallel

streams, one for training and one for testing, is also applicable here. As before the training stream is partitioned into tumbling windows and an ensemble of classifiers are generated. However, instead of assigning weights to the classifiers, a tuple boosting mechanism is used. A training tuple is first classified using the ensemble of classifiers. These classifications are combined using a simple voting scheme (e.g. average). If the overall classification of the training tuple is wrong, then the tuple is weighted highly during new classifier generation. Finally, the testing tuples are classified using the ensemble of classifiers and a simple voting scheme determines the eventual classification. According to experiments shown in [13], adaptive boosting handles concept-drifts and shifts more elegantly, i.e. the accuracy of the classifiers recovers more quickly with respect to concept-drifts and shifts, as compared to ensemble based bagging. As shown in Figure 2, training tuples are first predicted using the ensemble of classifiers. These predictions are combined using a simple voting mechanism. If the resulting combined prediction is incorrect, a higher weight is assigned to this tuple, so that the new classifier will increase the probability of correctly classifying this tuple. The testing tuples are predicted using the ensemble of classifiers and these predictions are combined using simple voting.

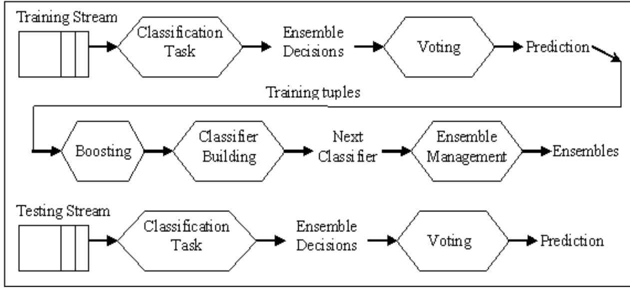

Figure 2: Generalized Boosting

Similar observation to one made before for weighted bagging, can be made here. Again only two boxes labeled 'Classification Task' and 'Classifier Building' are classifier specific. Thus, the same data flow model can be used to enable boosting on any arbitrary classification algorithm. Thus, Stream Mill generically and effectively supports such advanced techniques over data streams.

Furthermore, the extensibility provided by the Stream Mill system leaves room for implementation of other such techniques. For instance, [23], proposes a similar technique for handling concept-drifts and shifts. The technique again generates an ensemble of classifiers and augments training and testing tuples with predictions from previous classifiers. These predictions help the new classifier achieve better accuracy. Like ensemble based bagging and boosting, this technique can also be implemented as a generic built-in method is Stream Mill.

4.2 Association Rules

Association rule mining of data streams is required in many applications, including IP traffic monitoring and on-line recommendation systems. However, among the core mining methods, this presents one of the hardest research problems because of the difficulty of finding fast & light algorithms for determining frequent itemsets in an incoming stream of transactions [28]. Traditional algorithms for static data are no longer feasible here, because of the massive amount of data, the real-time requirements, bursty arrivals, and even more importantly concept shifts/drifts. To address this difficult problem we have developed a new mining algorithm called SWIM [34], that provides better performance than state-of-the-art algorithms [11, 31, 12] by optimizing incremental computation over sliding windows. SWIM has been implemented as a built-in UDA in our system, in ways that fully preserve the ability of the end-user to specialize (and optimize) its application by specifying which items and patterns should be included/excluded (in addition to support, confidence, and window size). Next, we briefly review this incremental mining algorithm.

4.2.1 SWIM (Sliding Window Incremental Miner)

SWIM [34] exploits the well-known *fp-tree* [25] data structure, which allows a compact representation of transactions. SWIM splits the window into several slides and then for each slide S, inserts the transactions in a separate *fp-tree*. Then, it computes the frequent itemsets in this small slide using any of existing static-data frequent miners (e.g. FP-growth [25]). These frequent itemsets for slide S are shown as $\sigma_\alpha(S)$ in the pseudo code, Figure 3, where α is the given minimum support. Since slides are mined separately and the occurrence of each pattern should be counted over all slides (to determine the total frequency), counting becomes a major bottleneck of the algorithm. The counting in SWIM is thus performed using a separate fast algorithm, which is based on *conditional counting* (=verification), called verifier. The verifier internally uses another *fp-tree* to store the patterns that need to be counted. This latter tree is called a Pattern Tree (PT). Then, counting is performed via conditionalizing both the pattern tree and the *fp-tree* of the slide in parallel. More details can be found in [34].

For Each New Slide S
1: For each pattern $p \in PT$
  update $p.freq$ over S
2: Mine S to compute $\sigma_\alpha(S)$
3: For each existing pattern $p \in \sigma_\alpha(S) \cap PT$
  remember S as the last slide in which p is frequent
4: For each new pattern $p \in \sigma_\alpha(S) \backslash PT$
  $PT \leftarrow PT \cup \{p\}$
  remember S as the first slide in which p is frequent
  create auxiliary array for p and start monitoring it

For Each Expiring Slide S
5: For each pattern $p \in PT$
  update $p.freq$, if S has been counted in
  update $p.aux_array$, if applicable
  report p as delayed, if frequent but not reported
    at query time
  delete $p.aux_array$, if p has existed since arrival of S
  delete p, if p no longer frequent in any of the current slides

Figure 3: SWIM pseudo code.

In the next section, we explain why and how the SWIM algorithm is provided as a built-in implementation in the extended Stream Mill system, and we discuss some of its features.

4.2.2 Built-in Support of SWIM

SWIM follows the standard windowed aggregates, by specifying computation for the EXPIRE and the ITERATE state. Furthermore, it is implemented in C/C++ for performance reasons; this takes advantage of the external UDAs supported by ESL [9]. Besides having a fast and built-in frequent itemsets algorithm, this has the following advantages.

Benefits. As discussed in [10], an inductive DBMS must be able to optimize the computation of frequent patterns and rules by tak-

ing advantage of the constraint specified by the user in the mining query, and special constructs and techniques were thus proposed to achieve that [10]. Since the extended Stream Mill system allows UDAs to access the user's post processing constraints (i.e., the *HAVING* clause) while they are running, these UDAs can exploit those constraints to optimize execution. Indeed, the built-in SWIM algorithm, utilizes these constraints to achieve better performance through better tuning.

In addition to constraints studied in [10], user can also specify the window size and the slide size. The set of the constraints that the built-in SWIM UDA can extract (and exploit) are discussed next.

- Including/Excluding Items. User may only ask for certain items to be included in (or excluded from) the mining process. In case of inclusion, such items will be the only ones that are considered in the conditionalization of the given *fp-tree* and PT, resulting in much smaller trees and better performance. For the exclusion instead, SWIM does the converse.
- Including/Excluding Patterns. User may even specify a set of interesting patterns (or association rules) as the only ones that she wants to monitor (or not monitor) over the data stream (for example, when they are permanently validated/rejected by an analyst). Such constraints can also be easily enforced and exploited by appropriately marking the corresponding nodes of the pattern tree PT with a *never-remove* (or *never-add*) flag.
- Window/Slide size. As for any other query written in ESL, the window size and slide size can be easily recognized by the query processor and be passed down to the UDA (here, SWIM algorithm) so that it adjusts its internal data structures accordingly.
- Report frequency. Instead of reporting all frequent patterns (or association rules) over the entire window, for each arrival or expiration of a slide, we only report new patterns or the expired ones (called delta reporting). Thus, the user himself does not have to determine the new/expired patterns by comparing the algorithm's output against the current ones. By extracting the frequency report from the user's query, SWIM can also be adjusted for further performance improvement by having larger slide size.

In Example 3, the query asks for continuously reporting of the new association rules and expired ones over a window worth of $1000,000$ tuples (transactions made in an on-line store), every time that it slides by $10,000$ tuples. Interesting and/or uninteresting patterns are also given by the user in two relational tables named *InterestingPatterns* and *AvoidPatterns*, respectively. The user in this example has asked only for the rules whose antecedent (left hand side) is among the set of interesting patterns and whose consequent (right hand side) is not among avoided patterns. Also the left-side of the rule should contain 'iPOD'. As described above, all these given constraints can be extracted and exploited in the SWIM algorithm transparently from the user, to achieve better optimization. The built-in swim UDA is called *AssociationRules* and it returns a 4-tuples (RuleLeft **VARCHAR**, RuleRight **VARCHAR**, support **REAL**, confidence **REAL**), where RuleLeft and RuleRight are formed by appending the single items of the corresponding rule together, in an ascending order.

EXAMPLE 3. *An ESL query with a set of constraints for frequent patterns.*

```
SELECT AssociationRules(T.Tid)
    OVER (ROWS 1000000 PRECEDING SLIDE 10000)
FROM Transactions T, Items I
WHERE I.Name = 'iPOD'
HAVING support >= 0.01 AND confidence >= 0.03
    AND RuleLeft IN (SELECT * FROM InterestingPatterns)
    AND RuleRight NOT IN (SELECT * FROM AvoidPatterns)
    AND CONTAINS(I.Id, RuleLeft);
```

As noted by previous research projects [10], this filtering is important from both the user and the system perspective. The user is likely to take full advantage of this feature, by specifying the post filtering conditions in the *HAVING* clause, since she does not want to receive a long list of uninteresting patterns. Of course, the UDA must be written to take advantage of these post conditions to constrain the search for uninteresting frequent patterns and improve its performance. Thus, while the DSMS is not responsible for these algorithm specific optimizations, since it passes the information in the *HAVING* clause to the UDA, the UDA can exploit them. Therefore, users can add new data mining algorithms as UDAs and take advantage of this feature to optimize execution without requiring any modifications to the Stream Mill compiler. This approach assures user-extensibility of the system—since, users can add new mining algorithms to the library, each with its method-specific optimizations, without touching the system internals.

4.3 Other Mining Algorithms

In addition to the classifiers, classifier ensembles, association rules, we are now building into the extended Stream Mill, a DM library with several key DM methods and improvements, such as clustering and sequential pattern detection discussed next.

4.3.1 On-line Clustering

The extended Stream Mill currently supports window versions of DBScan [17] and K-means [22]. Density-based clustering over tumbling windows is performed using DBScan [17] algorithm. The basic DBScan algorithm takes two parameters; neighborhood radius (*eps*) and number of required neighbors (*minPts*). Points within the *eps* distance of a particular point are considered its neighbors; distance can be a user defined function that gives some measure of dissimilarity between the points and follows standard distance function properties, such as non-negativity, reflexivity, symmetry, and triangle inequality. If a point has more than *minPts* neighbors then it is eligible to create (or participate in) a cluster. Given these two parameters the DBScan algorithm works as follows: pick an arbitrary point *p* and find its neighbors. If *p* has more than *minPts* neighbors then form a cluster and call DBScan on all its neighbors recursively. If *p* does not have more than *minPts* neighbors then move to other un-clustered points in the database. This can be viewed as a depth first search. Comparing the cluster results between successive windows provides an effective way to monitor trends and concept shifts and drifts [32].

The other option for on-line clustering is continuous clustering. The IncDBScan algorithm proposed in [16] modifies the original DBScan algorithm to perform this continuous clustering. The basic observation in IncDBScan is that the clustering assignments only change if number of neighbors of a point change from ($<$ *minPts*) to ($\geq$ *minPts*) or the other way around. Based on this observation IncDBScan proposes re-clustering only the required set of data tuples on tuple arrival and expiration. The algorithm is a simple extension of the DBScan algorithm, but performs much better than re-clustering all tuples on arrival/expiration of each new/old tuple. The EXPIRE state supported in ESL UDAs represents the perfect

tool to perform this delta maintenance. An ESL UDA implementing this algorithm can also be found at [21].

4.3.2 Sequence Detection

Sequence queries represent a very useful time-series analysis tool. They are useful in many practical on-line applications, such as click stream analysis, stock market data analysis, etc. A few sequence query languages have also been proposed to express such queries. For instance, SEQ [39], srql [36] SQL-LPP+ [35], and SQL-TS [37]. The SQL-TS language is based on allowing the use of Kleene-closure expressions in the FROM clause of its query. SQL-TS achieved unsurpassed levels of expressive power and optimizability [37].

An example sequence query, given a click stream such as the following, would be to find a user that went from an advertisement page to a product description page and then navigated to the product purchase page, a sequence of events signified by PageTypes ‘ad’, ‘pd’, and ‘pp’, respectively.

```
SELECT Y.PageNO, Z.ClickTime
FROM Sessions
  PARTITION BY SessNo
  ORDER BY ClickTime
  AS (X, Y, Z)
WHERE X.PageType = ‘ad’ AND Y.PageType = ‘pd’
  AND Z.PageType = ‘pp’
```

Such a query is very hard to write, and inefficient to execute, using SQL, whereas it is naturally specified in the efficiently executable SQL-TS. As demand for sequence pattern queries has grown in both database and data stream applications, SQL standard extensions are being proposed by collaborating DBMS and DSMS companies [45]. In most respects (syntax, semantics, and optimization) the standards are based on SQL-TS. A first implementation of SQL-TS is currently supported in the extended Stream Mill. We are now improving and extending it to support the proposed SQL standards [45].

As our reader might have observed, the success of SQL-TS also underscore that the UDA-based extensibility of the extended Stream Mill is not without limitations: SQL-TS and then new SQL standard suggest that some new language constructs are needed to deal effectively with special application domains. However, our experience shows that this is more of an exception than a rule. Finally our implementation of SQL-TS relied on mapping its sequence-oriented constructs into special UDAs. While the intuitive appeal of the original constructs was lost, this confirmed the basic generality of UDAs in terms of expressive power.

5. HIGH-LEVEL MINING LANGUAGE

In addition to the extensions and improvements previously described, the main focus of our future work will be to improve usability and friendliness of the system for more casual users. Indeed, while expert users would like to implement new mining algorithms or modify existing ones naïve users would find an approach such as OLE DB for DM [42] more suitable for their needs (or at least for their level of computing sophistication).

OLE DB for DM allows writing data mining queries in an intuitive language. Thus, we propose that the Stream Mill system should support a higher level language similar to OLE DB for DM. The system can internally translate such OLE DB for DM mining queries to equivalent ESL queries that call suitable aggregates. For instance, let’s consider the definition and training of a Naïve Bayesian classifier in OLE DB for DM, given in Example 4. This example works on the well-known Iris dataset, which has 4 real attributes, SL, SW, PL, and PW, and a class attribute. The class attribute can take one of three values (setosa, versicolor, or verginica).

EXAMPLE 4. *Naïve Bayesian Classifier in OLE DB for DM*

```
STREAM iris(id INT, SL REAL, SW REAL,
        PL REAL, PW REAL, class INT);
    /* Create a mining model */
CREATE MINING MODEL NaïveBayesianFlower (id INT KEY,
        SL REAL CONTINUOUS, SW REAL CONTINUOUS,
        PL REAL CONTINUOUS, PW REAL CONTINUOUS,
        class INT DISCRETE PREDICT)
    USING Microsoft_Naive_Bayes;
    /* Training the model */
INSERT INTO NaiveBayesianFlower (id, SL, SW, PL, PW, isSetosa)
    openquery (‘SELECT id, SL, SW, PL, PW, isSetosa
            FROM TrainFlowers’);
```

While the queries in Example 4 are succinct and intuitive, similar queries in ESL are not as intuitive. Therefore, we propose that given the OLE DB for DM statements for Example 4, the system should automatically translate them to ESL statements. For instance, the name of the mining model, NaiveBayesianFlower, is analogous to the name of the table, which stores the mining model. Similarly, the name of the mining algorithm, Microsoft_Naive_Bayes, is analogous to the name of the UDA that should be invoked to learn the mining model. Furthermore, the user can specify additional parameters for the UDA after the *using* clause, which is also consistent with OLE DB for DM. In general, mapping user queries, written in OLE DB for DM or other high-level mining language, to ESL queries is relatively straightforward. Furthermore, this extension greatly improves the usability of the system.

6. CONCLUSION

While DSMS and data stream mining algorithms have provided separate foci for many research projects, Inductive DSMS, which require a synergetic integration of their technologies, have received little attention until now. In this paper, we first showed that this is an important research topic that deserves much attention and poses interesting technical challenges. Then, we presented our approach based on extending the Stream Mill DSMS to support complex stream mining algorithms. Stream Mill provides a natural platform for adding the new inductive functionality, since its query language, ESL, provides (i) extensibility via user-defined aggregates and (ii) powerful window facilities for both built-in and user-defined aggregates. However, difficult technical challenges had to be solved to turn it into an Inductive DSMS. The first is the selection and careful implementation of mining algorithms that are fast and light enough to be used in continuous queries with real-time response. The second issue is how to support these algorithms generically—to assure that they can be used on streams with arbitrary schema. Another difficult design challenge is how to make the system open and extensible—to ensure that new mining algorithms can be easily introduced (or existing ones modified) by users working in the declarative framework provided by our DSMS. Finally, a high-level mining language is also being designed for our system: this will facilitate the intuitive invocation of mining methods.

7. REFERENCES

[1] Atlas user manual. http://wis.cs.ucla.edu/atlas.

[2] DB2 Universal Database http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp.

[3] Decision Tree Entropy Calculation http://decisiontrees.net/?q=node/27.

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

[5] ORACLE. Oracle Data Miner Release 10gr2 http://www.oracle.com/technology/products/bi/odm.

[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] Arvind Arasu and Jennifer Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, 2004.

[8] B. Babcock, S. Babu, M. Datar, R. Motawani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[9] Yijian Bai, Hetal Thakkar, Chang Luo, Haixun Wang, and Carlo Zaniolo. A data stream language and system designed for power and extensibility. In *CIKM*, pages 337–346, 2006.

[10] Toon Calders, Bart Goethals, and Adriana Prado. Integrating pattern mining in relational databases. In *PKDD*, volume 4213 of *Lecture Notes in Computer Science*, pages 454–461. Springer, 2006.

[11] W. Cheung and O. R. Zaiane. Incremental mining of frequent patterns without candidate generation or support. In *DEAS*, 2003.

[12] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz. Moment: Maintaining closed frequent itemsets over a stream sliding window. In *Proceedings of the 2004 IEEE International Conference on Data Mining (ICDM'04)*, November 2004.

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

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

[15] Guozhu Dong, Jiawei Han, Laks V.S. Lakshmanan, Jian Pei, HaixunWang, and Philip S.Yu. Online mining of changes from data streams: Research problems and preliminary results. In *SIGMOD*, 2003.

[16] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Michael Wimmer, and Xiaowei Xu. Incremental clustering for mining in a data warehousing environment. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 323–333, 1998.

[17] Martin Ester, Hans-Peter Kriegel, Jorg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Second International Conference on Knowledge Discovery and Data Mining*, pages 226–231, 1996.

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

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

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

[21] Stream Mill Examples. Approximate Frequent Items http://wis.cs.ucla.edu/stream-mill/examples/freq.html.

[22] E. Forgy. Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *Biometrics*, page 768, 1965.

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

[24] J. Han, Y. Fu, W. Wang, K. Koperski, and O. R. Zaiane. DMQL: A data mining query language for relational databases. In *Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD)*, pages 27–33, Montreal, Canada, June 1996.

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

[26] T. Imielinski and A. Virmani. MSQL: a query language for database mining. *Data Mining and Knowledge Discovery*, 3:373–408, 1999.

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

[28] Nan Jiang and Le Gruenwald. Research issues in data stream association rule mining. *SIGMOD Record*, 35(1):14–19, 2006.

[29] Minsoo Kim, Jae-Hyun Seo, Il-Ahn Cheong, and Bong-Nam Noh. *Fuzzy Systems and Knowledge Discovery*, chapter Auto-generation of Detection Rules with Tree Induction Algorithm, pages 160–169. Springer Berlin / Heidelberg, 2005.

[30] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. Data models and query language for data streams. In *VLDB*, pages 492–503, 2004.

[31] C.K.-S. Leung, Q.I. Khan, and T. Hoque. Cantree: A tree structure for efficient incremental mining of frequent patterns. In *ICDM*, 2005.

[32] C. Luo, H. Thakkar, H. Wang, and C. Zaniolo. A native extension of sql for mining data streams. In *SIGMOD*, pages 873–875, 2005.

[33] R. Meo, G. Psaila, and S. Ceri. A new SQL-like operator for mining association rules. In *VLDB*, pages 122–133, Bombay, India, 1996.

[34] Barzan Mozafari, Hetal Thakkar, and Carlo Zaniolo. Verifying and mining frequent patterns from large windows over data streams. In *International Conference on Data Engineering (ICDE)*, 2008.

[35] Chang-Shing Perng and D. S. Parker. SQL/LPP: A time series extension of SQL based on limited patience patterns. In *DEXA*, volume 1677 of *Lecture Notes in Computer Science*. Springer, 1999.

[36] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. Beyer, and M. Krishnaprasad. Srql: Sorted relational query language, 1998.

[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] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence query processing. In Richard T. Snodgrass and Marianne Winslett, editors, *SIGMOD*, pages 430–441. ACM Press, 1994.

[40] A. Siebes. Where is the mining in kdid? (invited talk). In *Fourth Int. Workshop on Knowledge Discovery in Inductive Databases,*, 2005.

[41] M. Sullivan. Tribeca: A stream database manager for network traffic analysis. In *VLDB*, 1996.

[42] Z. Tang, J. Maclennan, and P. Kim. Building data mining solutions with OLE DB for DM and XML analysis. *SIGMOD Record*, 34(2):80–85, 2005.

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

[44] Carlo Zaniolo. Mining databases and data streamswith query languages and rules (invited talk). In *Fourth Int. Workshop on Knowledge Discovery in Inductive Databases,*, 2005.

[45] Fred Zemke, Andrew Witkowski, Mitch Cherniak, and Latha Colby. Pattern matching in sequences of rows. Technical report, Oracle and IBM, 2007.