Classification Spanning Correlated Data Streams

Yabo Xu, Ke Wang
Computer Science School
Simon Fraser University
{yxu,wangk}@cs.sfu.ca

Ada Wai-Chee Fu
Department of Computer Science and Engineering,
The Chinese University of Hong Kong
adafu@cse.cuhk.edu.hk

Rong She, Jian Pei
Computer Science School
Simon Fraser University
{rshe, jpei}@cs.sfu.ca

ABSTRACT
In many applications, classifiers need to be built based on multiple related data streams. For example, stock streams and news streams are related, where the classification patterns may involve features from both streams. Thus instead of mining on a single isolated stream, we need to examine multiple related data streams in order to find such patterns and build an accurate classifier. Other examples of related streams include traffic reports and car accidents, sensor readings of different types or at different locations, etc. In this paper, we consider the classification problem defined over sliding-window join of several input data streams. As the data streams arrive in fast pace and the many-to-many join relationship blows up the data arrival rate even more, it is impractical to compute the join and then build the classifier each time the window slides forward. We present an efficient algorithm to build a Naïve Bayesian classifier in such context. Our method does not need to perform the join operations but is still able to build exactly the same classifier as if built on the joined result. It only examines each input tuple twice, independent of the number of tuples it joins in other streams, therefore, is able to keep pace with the fast arriving data streams in the presence of many-to-many join relationships. The experiments confirmed that our classification algorithm is more efficient than conventional methods while maintaining good classification accuracy.

Categories and Subject Descriptors
H.2.8 [**Database Applications**]: Data mining

General Terms: algorithms, management, performance

Keywords: algorithm, stream data, join, classification, Naïve Bayesian model

1. INTRODUCTION
At a time of information explosion, we see that data not only are stored in large amounts, but also keep growing quickly over time. Every day millions of bank transactions are recorded, telephone calls are registered, emails are stored, all of which keep being appended to existing databases. Such databases are therefore called data streams, as data continuously flow in and there is no particular order in which the data items arrive. Data streams are characterized as being in high volume, unbounded in size, dynamically changing and require fast response time [2]. As data are continuously evolving, so are the embedded trends and patterns. Since it is impossible to store the complete stream before the mining starts, quickly detecting evolving data characteristics is important for decision making. Any algorithm designed for data streams must have a very low computation time per input tuple in order to keep pace with the high data arrival rate.

Moreover, there are often situations in stream mining where multiple related data streams need to be examined at the same time in order to discover trends or patterns that involve features from different data streams. Indeed, there are many applications where the classification patterns span across multiple streams. For example, stock streams and news streams are related, traffic report streams and car-accident streams are related, sensor readings of different types are related. In such applications, co-occurrence of certain conditions in several related streams may jointly determine the class label, therefore related streams should be examined together to build the classifier.

To illustrate this, let us consider a simplified example. In the stock market, "favorable trading" refers to stock transactions that are favorable to the engaging party, i.e., selling before a stock plunges or buying before a stock goes up. In order to build classification models that identify patterns for "favorable trading", the stock trading stream that records all trading transactions must be examined. However, stock transactions are *not* isolated or independent events; they are related to many other data streams, e.g., phone calls between dealers and managers/staffs of public companies. Thus it is necessary to mine on multiple related data streams.

For example, the classification algorithm may need to look at all following correlated data:

Trading stream: **T** (τ, Dealer, Type, Stock, Class)
Phone call stream: **P** (τ, Caller, Callee)
Company table: **C** (Company, Stock)
Person table: **S** (Name, Org)

where τ is the timestamp, "Type" is either "sell" or "buy", "Class" ("yes"/"no") refers to the class label of being favorable trading or not. To compute the complete training set, a SQL query can be used to extract information from the above data as follows:

```
SELECT  *
FROM    P, S, T, C
WHERE   S.Name=P.Caller AND P.Callee=T.Dealer
        AND P.τ<T.τ AND T.Stock=C.Stock
```


CIKM'06, November 5–11, 2006, Arlington, Virginia, USA.

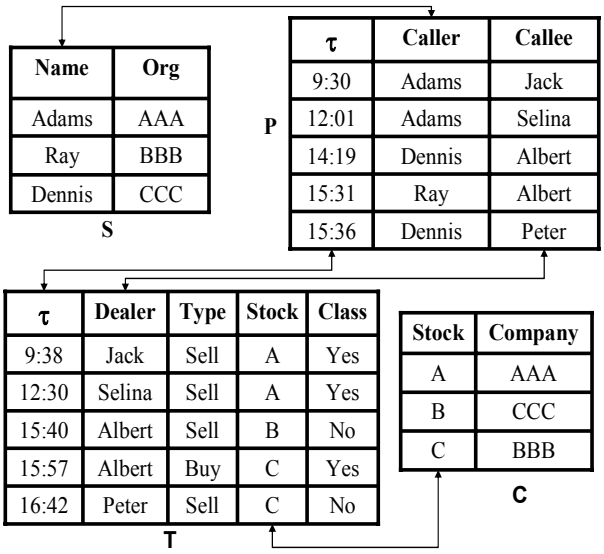

Table 1. Related streams / tables

Class	Caller	Org	Callee	Dealer	Type	Stock	Company
Yes	Adams	AAA	Jack	Jack	Sell	A	AAA
Yes	Adams	AAA	Selina	Selina	Sell	A	AAA
Yes	Ray	BBB	Albert	Albert	Sell	B	CCC
Yes	Ray	BBB	Albert	Albert	Buy	C	BBB
No	Dennis	CCC	Albert	Albert	Sell	B	CCC
No	Dennis	CCC	Albert	Albert	Buy	C	BBB
No	Dennis	CCC	Peter	Peter	Sell	C	BBB

Table 2. The join stream

Essentially, this query performs a join on all related data and the joined result is the training set used to build the classifier.

Table 1 shows a snapshot of such data. The join relationship is indicated by the arrows connecting the join attributes. Note that the join relationship between P and T is "many-to-many", which is the most general case. For example, "Albert" was called twice and traded twice, generating four tuples in the join stream in Table 2 (The timestamps for each join record are ignored because of the space.), the rule "Org=Company → Class=Yes" holds in 3 out of 4 tuples that have "Org=Company", i.e., with 75% confidence. It suggests that after getting a call, the trading on the caller's company stock tends to be more favorable. A classifier bearing rules that utilize information from multiple correlated streams/tables is likely more accurate than those built on the trading stream alone.

Motivated by the above discussions, in this paper, we will consider the problem of classification over multiple data streams with general join relationships. Such a problem is common in practice. Actually, in the later section, our experiments on a real-life dataset (UK road accident dataset) confirmed that classifiers built on multiple streams are much more accurate than those built only on a single target stream.

For such applications, the demand on efficiency of the stream applications is even more challenging. Since the size of the data stream is unbounded and new data continuously flow in, it is impossible to store the entire data stream first before the mining starts. Instead, stream applications only deal with the current part of a stream (called a *window*) which may look like a static table. As new tuples flow in, the window also slides forward to include the latest tuples while some old tuples are expired from the window. For conventional static relations, join is a natural operation to link related relations by data semantics. For online and unbounded data streams, joining the entirety of the data is impossible. Instead, a join can be defined on stream windows, called *sliding-window join* [2], which generates a new stream, called the "*join stream*", representing the join of all related streams in their current windows. When the window of any input stream slides forward, the join stream also evolves to include new joined tuples and invalidate expired joined tuples. The problem we are addressing in this paper is to build the classifier based on such a join stream, referred to as the *join stream classification* problem hereinafter. Note static tables are also allowed (as in Table 1) in the same classification problem by considering them as "streams" that never change. For join stream classification, the classifier must be updated each time any stream window is updated. The sliding of the window can be tuple-based or time-based. When the gap between two consecutive windows is small, the classifier must be rebuilt at very fast pace, i.e. the algorithm that builds the classifier must be very efficient.

For such join stream classification, a tempting straightforward solution may be to compute the join each time a window slides forward and then build the classifier based on the current join stream. Unfortunately, this is unrealistic for data streams. As the input streams are arriving in fast pace and the tuples in related streams arrive in no particular order, it is difficult to optimize the join operations, i.e. the join operations are expensive. Yet the classifier must evolve quickly each time the window slides forward, thus there may be no time to perform the join [20][6][23]. Furthermore, the join relationship can be "many-to-many" as shown in the sample data in Table 1, therefore, there are far more tuples in the join stream than in the input streams. Any method that explicitly generates the join stream will suffer from the blow-up of data arrival rates and is unlikely to be able to keep pace with the incoming data.

Some previous works have dealt with challenges in the single-stream classification problem [11][14][24][5]. To the best of our knowledge, there has been no work on such join stream classification problem. Although there have been many classification algorithms that work well on static tables, it is very difficult to adapt them to join stream classification. For example, support vector machines (SVM) require explicit generation of the training set, i.e. the join stream, in order to build the classifier. On the other hand, we notice that Naïve Bayesian Classifier (NBC) [13] has some unique properties which can be explored to avoid the join. NBC is one of the most widely used and successful classification methods. Although it assumes variables are

independent given the class label, researches show that NBC is still reliable even when this assumption is violated [12][22][19]. Thus we believe that the NBC presents a best opportunity in order to deal with the join stream classification problem. By taking advantage of the unique properties of the NBC classifier, we can efficiently address this problem.

To keep pace with the fast input streams, we propose a NBC method that does not need to join the input streams, while still producing the exact same NBC as produced on the join stream. Our insight is that the information required by NBC for the join stream can be obtained by computing "blow-up counts" directly from the input streams in linear time. Our main achievement lies in the fact that computing the blow-up counts is much cheaper than computing the sliding-window join itself. Our approach examines each input tuple in the current window twice, independent of the number of tuples it joins in the other stream windows. Note since the stream window can be held in main memory, scanning each tuple in the current window in linear time is very efficient. Thus it is highly suitable to handle high-speed streams in the presence of many-to-many join. Note the idea of computing such "blow-up counts" to avoid the expensive join operations is also applicable to some other classification algorithms that require similar statistics, for example, decision trees.

The rest of this paper is organized as follows. In Section 2, we review related works. In Section 3, we define the problem and discuss core concepts of NBC. In Section 4, we present our algorithm. We evaluate our method in Section 5. Section 6 concludes the paper.

2. RELATED WORKS

In data stream management [2], sliding-window join is proposed to answer queries involving the join of multiple data streams, such as the join size, sum [1][10], join-distinct [15]. Their focus was on how to compute these joined results under resource constraints and used techniques such as sampling [7][14] or load-shedding [6][20][23]. However, in the join stream classification problem, as we explained in the previous section, it is undesirable to first compute the join of multiple streams and then build the classifier. Thus these techniques cannot be applied.

Most stream mining algorithms consider a single stream and simple statistics such as average and standard deviation. Classification on data streams was considered in [11][14][24][5]. Other mining problems that involve multiple streams are clustering [18][4], correlation analysis [25], sequential patterns [8]. However, none of these works involves a general join among streams; thus, they do not deal with the blow-up of data arrival rates caused by a many-to-many join. For example, the correlation analysis in [25] computes the correlation coefficient of two time series, which aligns the two streams by a common key such as the timestamp. To our knowledge, the classification problem over a general sliding-window join has not been studied.

On the other hand, there have been works on classification over multiple static relations. For example, [2] presented a multi-relational decision tree, [20] and [25] studied rule inductions. In a relational learner, the training set cannot be defined by the join of multiple tables, making the problem very different from ours. In [28], a secure construction of decision tree classifiers from vertically partitioned data was presented, where the join is given by the one-to-one relationship implied by the common key identifier for all partitions. That work is not applicable to the general many-to-many join relationship. Recently, [27] proposed a secure construction for decision tree classifiers over distributed tables with the general many-to-many join relationship. Nevertheless, the work focuses on the privacy preserving aspect, not the data stream requirement. [29] proposed an efficient algorithm for building decision tree classifiers for the data given by a collection of tables related by a hierarchical structure of foreign key references, motivated by those found in relational databases, data warehouses, XML data, and biological databases. It examines the same performance issue caused by the blow-up of join. Unlike [29], this work deals with the arbitrary join relationship that is not necessarily represented by foreign key references, and focuses on stream data.

3. PROBLEM STATEMENT

In this section, we formally define the problem of join stream classification and briefly review the standard naïve Bayesian classification.

3.1 Join Stream Classification

Join stream classification refers to the problem when classification involves several related data streams S_1, …, S_n, and the classifier needs to be built on the join stream as defined by a sliding-window join. The specification of the sliding-window join over S_1, …, S_n includes the join condition, the window specification and update frequency of each input stream [2][17].

We consider the join condition that is a conjunction of equality predicates $S_i.A=S_j.B$ ($i\neq j$), where $S_i.A$ and $S_j.B$, called *join attributes,* represent one or more attributes from S_i and S_j. By allowing $S_i.A$ and $S_j.B$ to contain more than one attribute, we need at most one predicate $S_i.A=S_j.B$ between each stream pair S_i and S_j. The *join graph* is constructed in a way such that there is an edge between S_i and S_j if there is a predicate $S_i.A=S_j.B$ in the join condition. We consider *connected* and *acyclic* joins, that is, the join graph is well connected and contains no cycle. This is not a serious limitation since many joins in practice are in fact acyclic, i.e., chain joins and star joins over the commonly used star/snowflake schemas.

The window and update specification in join stream classification can be time-based or tuple-based. The term "window" refers to the collection of current windows of all streams. One of $S_1,\ldots,S_n$, called the *target stream*, contains the class column. The task is to build a classifier each time the window updates. This means that the classifier must be rebuilt whenever the window on any input stream slides forward. The speed of fastest-sliding window determines the rate of classifier updates. For the current window, the training set is the set of tuples defined by the sliding-window join. Note that the training set is not explicitly given, but specified by the input streams and the sliding-window join. This distinction is important because the training set is significantly larger than

$$P(x \mid C_i) = \prod_{j=1..n} P(x_j \mid C_i)$$

the input streams and being able to work on the latter directly has performance advantages.

3.2 Naïve Bayesian Classifiers

Consider a single table T ($X_1,\ldots, X_n$, Class), where "Class" denotes the class column whose domain is [$C_1,\ldots, C_m$].

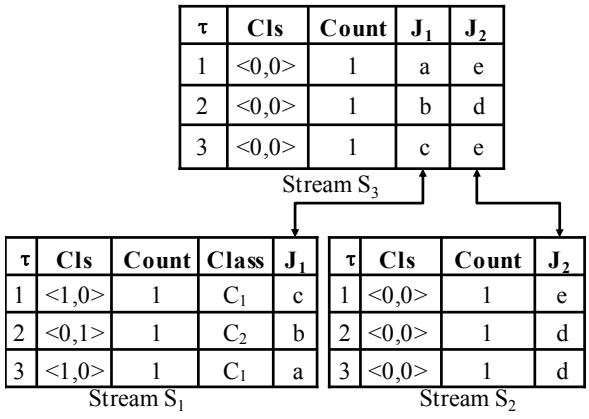

Figure 1. Example with 3 streams at initialization

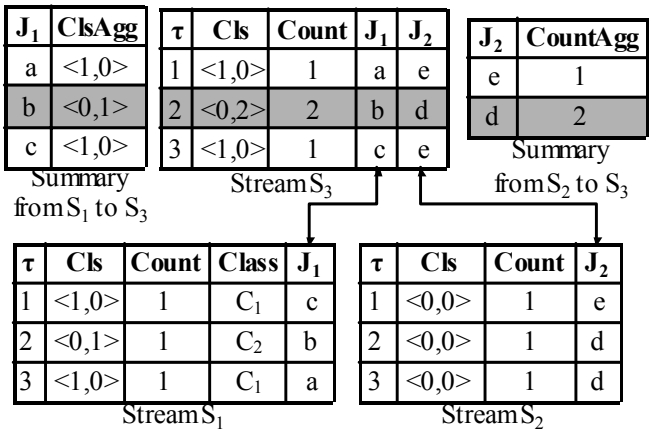

Figure 2. After bottom-up propagations

To classify a tuple $x=(x_1,\ldots, x_n)$, the Naïve Bayesian Classifier (NBC) assigns x to class C_i that maximizes the conditional class probability $P(C_i|x)$ based on the following maximum a posteriori (MAP) hypothesis:

$$\underset{C_i \in Class}{\operatorname{argmax}} P(C_i \mid x) = \underset{C_i \in Class}{\operatorname{argmax}} P(x \mid C_i) P(C_i)$$

where $P(C_i)$ is the class probability and $P(x|C_i)$ is the conditional probability of x given the class label C_i. Under the assumption that variables $X_1, \ldots, X_n$ are independent given the class label, NBC estimates $P(x|C_i)$ by

Once $P(x_j|C_i)$ and $P(C_i)$ are collected from the training data, NBC is able to assign a class label to a new tuple x.

To compute $P(x_j|C_i)$ and $P(C_i)$, we only need to compute the *class count matrix* of the form $(x_k, <N_1,\ldots, N_m>)$ for each distinct value x_k of X_j, where N_j ($1\le j\le m$) is the number of tuples that has the value x_k and the class label C_j. This data structure has a size proportional to the number of distinct values in X_j. Note NBC requires attributes to be categorical (having a small number of distinct values). Continuous attributes can be first discretized (such as equi-width or equi-depth binning) into a small number of intervals before applying NBC.

The above discussion assumes a single table T. For join stream classification, T will be the join result of the input streams in the current window. Since such T is much larger than the input streams, the challenge is to compute the class count matrix for T without generating T. In the next section, we present such a method.

4. OUR APPROACH

We assume the current windows of all input streams are held in memory. The join relationships among streams form an acyclic join graph, which is in fact a rooted tree. Any stream may be regarded as the root and the target stream may be at any position in the tree. As our method involves propagation of information along the edges of the tree, we will call this tree as the *propagation tree*.

Instead of generating the join stream, we maintain a data structure of (*Cls*, *Count*) for each tuple t in the input streams, where *Cls* is a *class vector* in the form of $<N_1,\ldots, N_m>$ (m is the number of classes, N_i records the count of occurrences of t with class label i in the join stream) and *Count* is a counter that keeps track of the total occurrences of t in the join stream. Note this data structure stores all information about a tuple t in the join stream, but with the size proportional to the size of the input stream. The class vectors (*Cls*) correspond to the class count matrix as mentioned in section 3.2, which is the only information required to build a NBC classifier. The challenge is how to compute such class vectors for each tuple in the input streams without performing the join.

We compute the class vectors by propagating the blow-up effect of the join. The propagation proceeds in two phases. In the phase of *bottom-up propagations*, all *Cls's* and *Counts* are propagated from the leaf nodes to the root. On reaching the root, the *Cls's* in the root reflect the join of all streams. Next, in the phase of *top-down propagations*, we propagate *Cls's* from the root to all leaf nodes. When reaching all leaf nodes, the class vectors (*Cls's*) in each stream have reflected the join effect of all streams. Our main achievement lies in the fact that computing the blow-up ratios through propagations is much cheaper than computing the sliding-window join itself.

The detailed process of computing such data structures is explained in the following subsections. To correctly carry out computations during propagations, we first define the *arithmetic operations on Cls* as follows: given an operator "○" and two *Cls's* (V_1: $<a_1,\ldots,a_m>$ and V_2: $<b_1,\ldots,b_m>$), $V_1 \circ V_2 = <a_1 \circ b_1,\ldots,a_m \circ b_m>$. For example, $<4,3>/<2,3>=<2,1>$.

4.1 Initialization

Initially, for any tuple in the target stream, its *Cls* ($<N_1, \ldots, N_m>$) is determined by its class label C_i such that, $N_i=1$ and $N_j=0$ where $i\neq j$. The *Count* is always initialized to 1.

For any tuple in all other streams, its *Cls* is initialized to all zeros ($<0,\ldots,0>$) and *Count* is always 1.

Figure 1 gives an example with 3 streams with initial *Cls's* and *Counts* values shown. The join relationships are specified by the arrows: S_1 and S_3 join on J_1, and S_2 and S_3 join on J_2. S_1 is the target stream containing two classes. S_3 is the root of the propagation tree. Note the root of the tree can be arbitrarily selected. We will show later that choosing the input stream with largest window size (i.e. the most number of tuples in a window) as the root can optimize the cost of scan on input streams.

Note that such initialization does not require a separate scan on the input streams and can be combined with the bottom-up propagation process as discussed in the following subsection.

4.2 Bottom-Up Propagation

This is the phase where the information of *Cls* and *Count* are propagated from leaf nodes to the root in a bottom-up order. Consider a parent node P and a child node C with join predicate $P.J_1=C.J_2$.

Observation 1: Given a tuple *t* in P, if *t* joins with *k* tuples in C, *t* will occur *k* times in the join between P and C. These occurrences can be reflected directly in P by blowing up the *Cls* and *Count* of *t* using the aggregated *Cls* and *Count*, denoted as *ClsAgg* and *CountAgg*, of the *k* joining tuples in C.

Formally, we define *blow-up summary from C to P* as the set {(*v*, *ClsAgg*, *CountAgg*)}, where *v* is a join value in C. It has the size that is proportional to the number of distinct join values in C and can be collected by one scan of C.

Observation 2: If P has *n* child nodes (*n*>1), the *Cls* and *Count* of *t* in P will be blown up by all children as in Observation 1, to reflect the join with all children streams. Note in this phase, as we propagate only in the bottom-up order, at most one of the children contains non-zero *ClsAgg'*s (the branch containing the target stream).

The following lemma follows from the above observations:

Lemma 1. With a parent node P and its *n* child nodes, for each tuple *t* in P with join values ($v_1,...,v_n$), where each v_i corresponds to the join attribute between P and the *i*th child, let (v_1, $ClsAgg_1$, $CountAgg_1$), …, (v_n, $ClsAgg_n$, $CountAgg_n$) denote the *n* corresponding summary entries from all children, *t*'s *Count* is blown up as:

$$t.Count = \prod_{j=1..n}(CountAgg_j);$$

if there's some entry $ClsAgg_i$ ($1 \le i \le n$) being non-zero (by Observation 2, there's at most one such entry), *t*'s *Cls* is updated as:

$$t.Cls = (ClsAgg_i) * \prod_{j=1..n, j \neq i}(CountAgg_j)$$ ■

Now we can propagate the blow-up summaries from child nodes to the parent node P. After receiving all children's blow-up summaries, we scan P once and update its *Count'*s and *Cls'*s as in Lemma 1. We also create the blow-up summary from P to its own parent (if any) in the same scan.

Figure 2 shows the bottom-up propagation following the example in Figure 1, where S_1 and S_2 are scanned to produce blow-up summaries to propagate to S_3. Note trivial aggregated counts (*ClsAgg* is all zeros or *CountAgg*=1) are ignored and not shown in summaries. On receiving the summaries, S_3 blows up *Cls* and *Count* of its tuples. For example, consider the tuple *t* in S_3 as grayscaled in Figure 2 (with J_1=b, J_2=d). It has two corresponding summary entries: (b,<0,1>,1) from S_1 and (d,<0,0>,2) from S_2, each containing all information on *t*'s joining tuples in that child. The blow-up of *t* by these entries in fact represents the effect of join: *t.Count* is blown up by 1*2=2, *t.Cls* is blown up by <0,1>*2=<0,2>. These results indicate that *t* occurs in the join twice, both having the class label C_2, which is exactly the same information as in the join stream.

In general, the target stream can be anywhere in the tree, thus there are two cases in the bottom-up propagation from children to a parent node P:

- If the target stream is not in P's subtree, we blow up only *Count* at P since *ClsAgg* is always empty; *Cls* will be blown up later at some ancestor node of P;
- If the target stream is in P's subtree, we blow up both *Cls* and *Count* at P; in this case either P is the target stream or one of its child nodes has non-empty *ClsAgg'*s.

4.3 Top-Down Propagation

At the end of bottom-up propagation, the *Cls* in the root stream reflects the effect of join of all streams. However, the *Cls'*s in all other streams have not reflected the joins performed at their ancestors. Thus we need to propagate in the top-down fashion to push the correct join information to all non-root streams. The propagation is based on the following observations.

Observation 3: For a parent node P and a child node C, if a tuple *t* in C joins with some tuple in P that has the join value *v*, so do *all* tuples in C that have this join value *v*. We can view all such tuples as an "*equivalence class"* on the join value *v* in C, denoted as C[*v*]. Similarly, P[*v*] is defined as the corresponding tuples in P that share the same join value *v*. The *Cls'*s of C[*v*] tuples must be updated by redistributing the aggregated count of P[*v*] tuples with following two properties: (1) the share of any tuple in its own equivalence class remains constant; (2) the aggregated counts in C after redistribution must be the same as in P.

Thus, to perform the top-down propagations properly, we define the *distribution summary from P to C* as the set {(*v*, *ClsAgg*)}, where *v* is a join value in P and *ClsAgg* is aggregated class counts of all P[*v*] tuples. Note there's one distribution summary from the parent to each child, with the size proportional to the number of distinct values of each join attribute.

Lemma 2. Given a distribution summary entry (*v*,.*ClsAgg*) from P, we redistribute the *ClsAgg* among C[*v*] tuples such that, for any tuple *t* in C[*v*]:

$$t.Cls = ClsAgg * (t.Count / \text{C[v]}.CountAgg)$$

where C[v].*CountAgg* is the aggregation of *Count'*s of all C[*v*] tuples prior to redistribution and as such, (*t.Count* / C[*v*].*CountAgg*) represents *t*'s share in C[*v*]. ■

Hence, on receiving the distribution summary from P, the *Cls'*s in C are updated as in Lemma 2, whereas the distribution summary from C to its own children (if any) are computed in the same scan.

Figure 3 shows the top-down propagation. At the root S_3, the distribution summaries to S_1 and S_2 are generated while scanning S_3 in the bottom-up propagation. On receiving these summaries, S_1 and S_2 redistribute their *Cls'*s. For example, for the tuple *t* in S_1 as grayscaled in Figure 3 (with J_1=b), *t.Cls*=<0,1> is redistributed by <0,2>*(1/1)=<0,2>, where (b,<0,2>) is the summary entry corresponding to b, and (1/1) is the share of *t* in its own equivalence class (having J_1=b). The result captures exactly the same information about *t* as in the join stream: *t* occurs twice having the class label C_2.

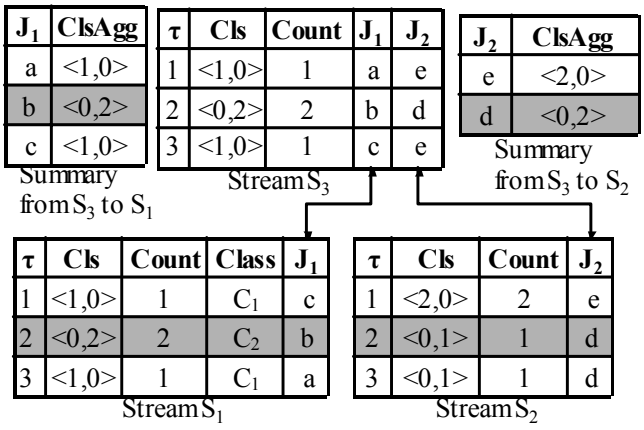

Figure 3. After top-down propagations

4.4 Cost Analysis

In the bottom-up and top-down propagation, one summary is passed between each parent/child pair and each stream (window) is scanned once. At any time, only the summaries for current parent/children are kept in memory. The size of a summary is proportional to the number of distinct join values, not the number of tuples. A summary lookup operation takes constant time in an array or hash table implementation. Therefore, the whole algorithm is linear in the stream window size, independent of the join size. This property is important because the join size can be arbitrarily large compared with the window size, due to the many-to-many join relationships.

The algorithm scans each input stream twice, once at the bottom-up propagation phase and once at the top-down propagation process. The only exception is the root stream, where the bottom-up and top-down propagations meet, two scans can be combined into one. Therefore, choosing the input stream of the largest window size (i.e., the most number of tuples) as the root will minimize the cost of scans, as it saves one scan on the largest stream window.

5. EMPIRICAL STUDIES

The objectives of our evaluations are two-folded: to verify that the classifier built on the join stream is more accurate compared with that built on a single stream; and to study the scalability of our algorithm.

We denote our algorithm as *NB_Join*, as it builds a NBC classifier whose training set is defined on the join of multiple streams. Note our algorithm does not need to actually perform the join; instead, the classification process is performed directly on the input streams. We compared it with following alternatives:

- *NB_Target*: NBC based on the target stream alone. In this case, all non-target streams are ignored.

- *DT_Join*: decision tree classifier (C4.5) on the join stream. To build the decision tree, the join stream is first computed by actually joining the input streams.

- *DT_Target*: decision tree classifier on the target stream alone.

To compare accuracy results, for each window, we train the classifier on the first 80% of data tuples within this window. The remaining 20% of data tuples in the same window are kept as testing samples for testing the classification accuracy. Note that the testing data are generated by a sliding-window join on the testing samples from all streams and constitute one join stream.

To compare the scalability, we focus on the classifiers that are built on the join stream and measure the scalability by computing "time per input tuple", i.e., time spent on each window divided by the number of tuples in the window. It gives an idea about the data arrival rate that an algorithm is able to handle. For *DT_Join*, because it has to generate the join stream before building the classifier, we measure the join time and ignore the classifier construction time since the join cost is the most expensive part.

Most of slide-window join algorithms in literature are not suitable for generating the join stream for *DT_Join* because they focus on fast computing special aggregates [10][15], or producing approximate join results [23] under resource constraints; not the *exact* join result. Therefore, we have to implement the join algorithm for slide window join. For simplicity, we implemented the nested loop join algorithm. This choice should not have a major performance effect because all tuples in the current window are kept in memory.

All programs were coded in C++ and run on a PC with 2GHz CPU, 512M memory and Windows XP.

5.1 Real-life Dataset

For experiments on real-life dataset, we obtained UK road accident data from the UK data archive[1]. It collects information about accidents, vehicles and casualties, in order to monitor road safety and determine policies to reduce the road accident casualty toll. It contains three tables: "Accident", "Vehicle" and "Casualty". The characteristics of year-2001 data are shown in Figure 4 where arrows indicate join relationships: each accident involves one or more vehicles; each vehicle has zero or more casualties. Each table can be regarded as a stream that is timestamped by "date of accident". In average, about 600 "Accident" tuples, 700 "Vehicle" tuples and 850 "Casualty" tuples are added every day. The join stream is specified by equality join on the common attributes among the streams. "Casualty" is the target stream with two casualty classes --- class 1: "fatal/serious" (13% of all tuples) or class 2: "slight" (87% of tuples).

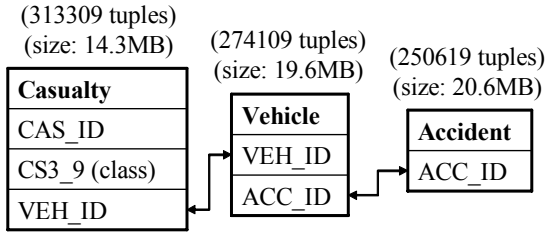

Figure 4. UK road accident data (2001)

[1] http://www.data-archive.ac.uk/

5.1.1 Accuracy

Figure 5 shows accuracies of all classifiers being compared. For all methods, the window size is the same and ranges from 10 to 50 days with no window overlapping.

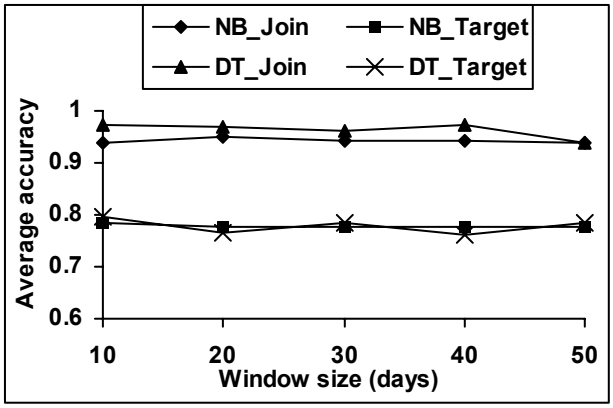

Figure 5. Classifier accuracy

It is immediately clear that classifiers built on multiple streams are much more accurate, showing that examining correlated streams is advantageous compared with building the classifier on a single stream. In fact, the accuracy obtained by examining the target stream alone is only about 80%, even lower than that obtained by a naïve classifier which simply classifies every tuple as belonging to class 2, since 87% of tuples belong to this class.

On the other hand, the results also show that, with the same training set, naïve Bayesian classifier has comparable performance as decision trees. Keep in mind that our method *NB_Join* runs directly on the input streams, while the decision tree is built on the join stream and thus is subject to the join cost. We examine the efficiency of these two methods in the next set of experiments.

5.1.2 Time per input tuple

Figure 6 compares the time per input tuple. For example, at the window size of 20 days, the join takes about 9.83 seconds whereas *NB_Join* takes only about 0.3 seconds. Therefore, the join time per input tuple is $9.83*10^6/43{,}900=224$ microseconds, where 43,900 is the total number of tuples that arrived in the 20-day window. In contrast, *NB_Join* takes only $0.3*10^6/43{,}900=6.8$ microseconds per input tuple. Thus, any method that requires computing the join will be at least 33 times slower than our method *NB_Join*. As the window size increases, the join time increases quickly due to the increased join cardinality in a larger window; whereas the time per input tuple for *NB_Join* is almost constant, indicating that our approach is linear to the window size. Thus our method can handle a much higher speed of window sliding than conventional methods.

Therefore, while both *NB_Join* and *DT_Join* classifiers exhibit similar classification accuracies, *NB_Join* is much more efficient than *DT_Join*.

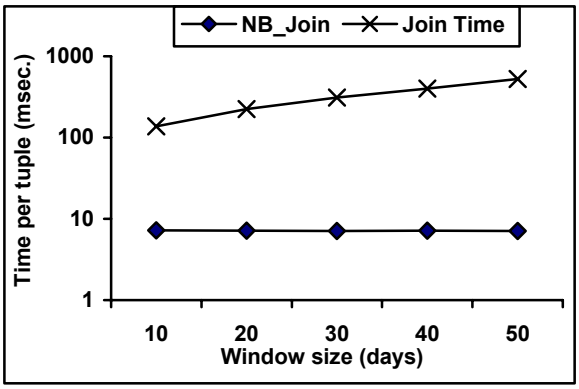

Figure 6. Time per input tuple

5.2 Synthetic Datasets

To further verify our claims, we also used synthetic datasets with various data characteristics. Similar to the experiments on real-life datasets, we want to examine whether the correlation of multiple streams yields benefits for classification under different data characteristics. We also want to evaluate if *NB_Join* can deal with streams with high data arrival rates.

As we are not aware of an existing data generator to evaluate classification spanning several related streams, we designed our own data generator.

5.2.1 Data Generator

To focus on important data characteristics, we make some simplifying assumptions. We consider the chain join of k streams $S_1, \ldots, S_k$, where S_1 is the target stream, and each adjacent pair S_i and S_{i+1} have one join predicate. All join attributes are categorical and have the same domain size D. All streams have the same number of tuples $|S|$. All streams have N numerical and N categorical attributes (excluding join attributes and the class attribute). All numerical attributes have the ranked domain $\{1,\ldots,10\}$, i.e., values are treated as being discretized into 10 categorical intervals. Categorical values are drawn randomly from a domain of size 20.

To verify our claim that classifiers built on the join stream are more accurate when there are correlations among streams, we need the dataset to contain certain structure rather than randomly generating the data tuples. To this end, we construct datasets such that the class label in the join stream is determined by whether at least q numerical attributes have "high" values, where q is a percentage parameter. A numerical value is "high" if it belongs to the top half of its ranked domain. Since the numerical attributes are distributed among multiple input streams, to ensure the desired property on the join stream, the input streams $S_1,\ldots,S_k$ are constructed as follows.

- ***Join values***. Each stream S_i consists of D groups: from 1st to Dth group. All tuples in the jth ($1 \le j \le D$) group of S_i join with all tuples in the jth group of S_{i+1}, but not any other tuples. The *jth join group* refers to the set of join tuples produced by the jth groups. The size Z_j of the jth group is the same in all streams $S_1,\ldots,S_k$, and follows Poisson distribution with the mean $\lambda=|S|/D$. The jth join group has the size Z_j^k, with λ^k

being the mean. The *blow-up* of the join is defined as $\lambda^k/\lambda=\lambda^{k-1}$, i.e., the ratio between the mean of group size on the join stream and that on input streams.

- ***Numerical values***. We generate the numerical attributes such that all join tuples in the *j*th join group have the same class label, by having "high" values in the *same* number of numerical attributes, say *hj*. To ensure this property, we distribute the number *hj* among $S_1,\ldots,S_k$ randomly, say $hj_1,\ldots,hj_k$, such that $hj=hj_1+\ldots+hj_k$, and all tuples in the *j*th group for S_i are "high" in hj_i numerical attributes. *hj* follows uniform distribution in the range $[0,k*N]$, where $k*N$ is the total number of numerical attributes.
- ***Class labels***. If $hj \geq q*k*N$, for some percentage parameter *q*, we assign the "Yes" class label to every tuple in the *j*th group of S_1, otherwise, assign the "No" class label.

Finally, to simulate the concept drifting in data streams, we change the class distribution every time after generating *W* tuples. This is done by varying the parameter *q*: let *w* be the window size, for every *W* tuples (*W*>>*w*), we randomly determine a *q* value in the range [0.25, 0.75) following the uniform distribution.

Thus a dataset generated as above can be characterized by the set of parameters (N,$|S|$,D,λ,W), where $\lambda=|S|/D$ is the mean of group size and determines the *blow-up ratio* of join.

5.2.2 Accuracy

We generated three streams S_1, S_2 and S_3 with parameters N=10, $|S|$=1,000,000, D=200,000, λ=5, W=100,000. Figure 7 shows the accuracy results with 50% window overlapping. *DT_Join* and *NB_Join* are more accurate than their counterparts on the single stream, while both having similar accuracies.

Figure 8 shows another experiment, where we fixed the window size *w* at 20,000 and decreased *W* from 100,000 to 20,000, in order to simulate situations where classification patterns change more frequently. Since the previous experiments have confirmed that classifiers built on the join stream have better accuracies, in this experiment, we only show the accuracy results of *NB_Join* and *DT_Join*. As expected, the accuracy drops slowly as *W* decreases, since there are more windows spanning data with different characteristics, making it difficult for a classifier to correctly identify the classification pattern.

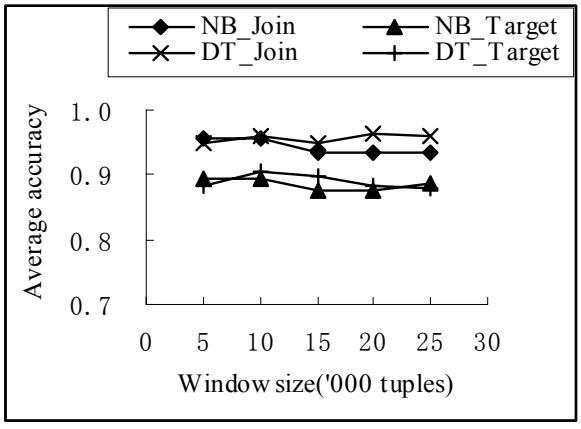

Figure 7. Classifier accuracy

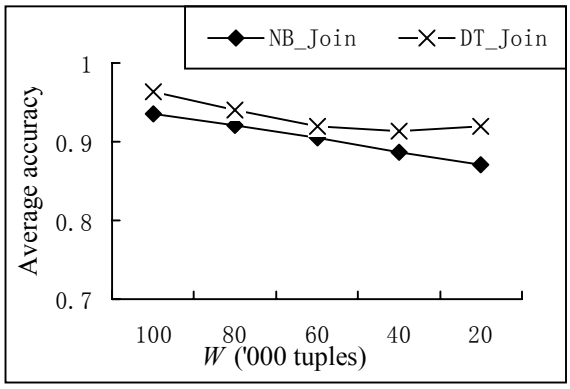

Figure 8. Classifier accuracy with changing data patterns

5.2.3 Time per input tuple

Figure 9 shows the time per tuple on the same dataset as in Figure 7. The join time is much larger than the time of *NB_Join.* As the window size increases, the join time increases due to the blow-up effect of join, while *NB_Join* spends almost constant time per tuple for any window size.

Figure 10 shows the time per tuple vs. the blow-up of join. All parameters are the same as previous except λ. For the join of three streams, the blow-up ratio is λ^2. By varying λ from 2 to 7, the blow-up varies from 4 to 49. The window size is fixed at 20,000. Again, *NB_Join* shows a much better performance and is flat with respect to the blow-up of join. This is because it scans the window exactly twice, independent of the blow-up ratio of the join. On the other hand, the join takes more time per tuple with a larger blow-up ratio because much more tuples are generated.

Figure 11 shows the time per tuple vs. the number of streams. All parameters are still the same as in Figure 9. The window size is fixed at 20,000 tuples. We vary the number of steams from 1 to 5. The blow-up ratio for k-stream join is determined by $5^{(k-1)}$. The comparison of the results is similar to Figure 10.

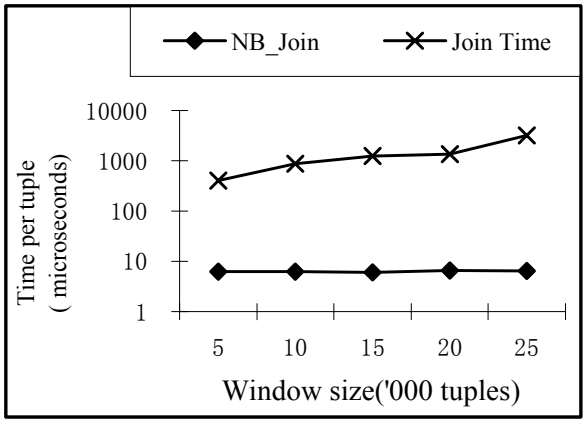

Figure 9. Time per input tuple vs. window size

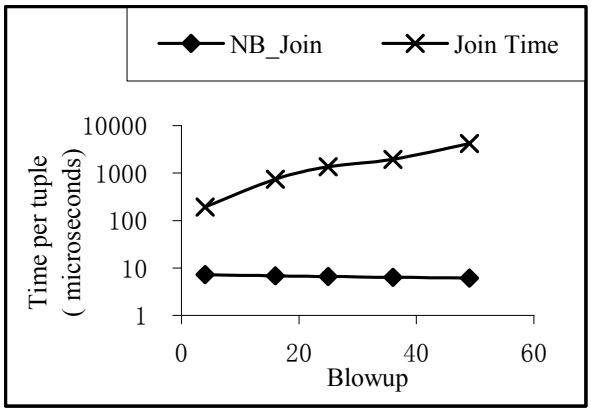

Figure 10. Time per input tuple vs. blow-up ratio

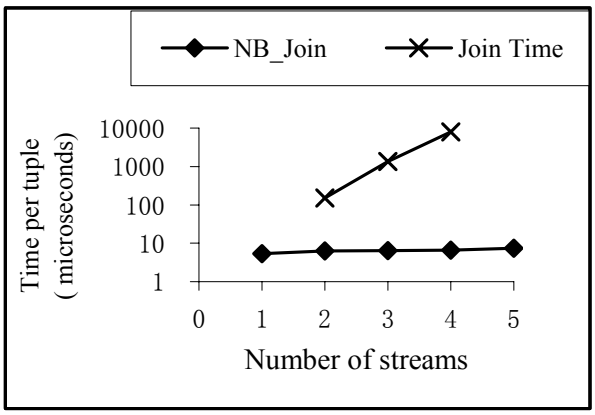

Figure 11. Time per input tuple vs. number of streams

5.3 Discussion

On both the real life and synthetic datasets, our empirical studies showed that when the features for classification are contained in several related streams, the proposed join stream classification has significant accuracy advantage over the conventional method of examining only the target stream. Thus a classification algorithm should examine as much related information as possible.

The main challenge is how such classification can be performed in pace with the high-speed input streams, given that the join stream has an even higher data arrival rate than that of the input streams, due to the arbitrary join blow-up ratios. To this end, our experiments showed that our proposed algorithm has the cost linear to the size of input streams, independent of the join size. Thus our algorithm is scalable and superior to all other alternative methods.

It is worthy of note that the classifier must be rebuilt each time the window on any input stream slides forward. This is reasonable when there is no overlap or only small overlaps between slide windows. However, when windows are significantly overlapped, this strategy tends to repeat the work on the overlapped data. In this case, a more efficient strategy may be incrementally updating the NBC by working only on the difference due to the window sliding. We did not pursue in this direction further because even overlapped tuples still need to be joined with *new* tuples in the other streams, which means that the scan of overlapped tuples cannot be avoided. Since our algorithm scans the current window only twice, the benefit of being incremental is limited, especially considering the overhead added.

6. CONCLUSIONS

Real life classification often involves multiple related data streams. Due to the online and high volume nature of data streams, with the join that blows up the data arrival rate on top of the rapidly arriving streams, it is prohibitive to perform the sliding-window join and then conduct classification analysis on the join. To solve this problem, we explored the property of Naïve Bayesian classifiers and proposed a novel technique for rapidly obtaining the essential join statistics without actually computing the join. With this technique, we can build exactly the same Naïve Bayesian classifier as using the join stream, but with a processing cost that is linear to the size of the input streams and independent of the join size. Empirical studies supported our two claims: examining several related streams indeed benefits the quality of classification; and the proposed method has much lower processing time per input tuple, thus, is able to handle much higher data arrival rates, even in the presence of many-to-many join relationships.

7. REFERENCES

[1] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. Tracking Join and Self-Join Sizes in Limited Storage. *In ACM PODS*, Philadeplphia, Pennsylvania, 1999.

[2] A. Atramentov, H. Leiva and V. Honavar, A multi-relational decision tree learning algorithm – implementation and experiments. ILP 2003.

[3] B.Babcock, S. Babu, M. Datar, R. Motwani, J. Widom. Model and issues in data stream systems. *In ACM PODS*, Madison, Wisconsin, 2002.

[4] J. Beringer and E. Hullermeier. Online clustering of parallel data streams. In press for *Data & Knowledge Engineering,* 2005.

[5] Y. D. Cai, D. Clutter, G. Pape, J. Han, M. Welge and L. Auvil. MAIDS: Mining alarming incidents from data streams. *In Proc. SIGMOD*, demonstration paper, 2004.

[6] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. *In Proc. VLDB,* 2002.

[7] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. *In Proc. SIGMOD,* 1999.

[8] G. Chen, X. Wu, X. Zhu. Sequential pattern mining in multiple streams, *In Proc. ICDM,* 2005.

[9] A. Das, J. Gehrke and M.Riedewald. Approximate join processing over data streams. *In Proc. SIGMOD*, Madison, Wisconsin, 2003.

[10] Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. *In Proc. SIGMOD*, Madison, Wisconsin, 2002.

[11] P.Domingos and G. Hulten. Mining high-speed data streams. *In Proc. SIGKDD,* 2000.

[12] Pedro Domingos and Michael Pazzani. On the optimality of the simple Bayesian classifier under zero-one loss. *Machine Learning,* 29:103-130, 1997.

[13] R. O. Duda and P. E. Hart. *Pattern classification and scene analysis*. New York: John Wiley & Sons, 1973.

[14] J. Gama, R. Racha, P.Medas. Accurate decision trees for mining high-speed data streams. *In Proc. SIGKDD,* 2003.

[15] S. Ganguly, M. Garofalakis, A. Kumar and R. Rastogj. Join-distinct aggregate estimation over update streams. *In Proc. ACM PODS*, Baltimore, Maryland, 2005.

[16] J. Gehrke, R. Ramakrishnan and V. Ganti. Rainforest – A framework for fast decision tree construction of large datasets. *In Proc. VLDB*, San Francisco, 1998.

[17] L. Golab, M. Tamer Ozsu. Processing sliding window multi-joins in continuous queries over data streams. *In Proc. VLDB*, 2003.

[18] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. *In FOCS,* 2000.

[19] DJ. Hand and K. Yu, Idiot's Bayes - not so stupid after all? *International Statistical Review*. 69(3), 385-399, 2001.

[20] G. Hulten, P. Domingos and Y. Abe, Mining massive relational databases, 18th International Joint Conference on AI - Workshop on Learning Statistical Models from Relational Data, Acapulco, Mexico, 2003.

[21] J. Kang, J. Naughton, S.Viglas. Evaluating window joins over unbounded streams. *In Proc. ICDE,* 2003.

[22] Irina Rish. An empirical study of the naive Bayes classifier. *IJCAI 2001 Workshop on Empirical Methods in Artificial Intelligence*, 2001.

[23] U. Srivastava, J. Widom. Memory-limited execution of windowed stream joins. *In Proc. VLDB,* 2004.

[24] H. Wang, W. Fan, P. Yu and J. Han. Mining concept-drifting data streams using ensemble classifiers. *In Proc. SIGKDD*, Washington DC, USA, 2003.

[25] X. Yi, J. Han, J. Yang, and P. Yu. Crossmine: efficient classification across multiple database relations. ICDE 2004.

[26] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. *In Proc. VLDB,* 2002.

[27] K. Wang, Y. Xu, R. She, P. Yu. Classification Spanning Private Databases. *AAAI*, 2006.

[28] W. Du and Z. Zhan. Building decision tree classifier on private data. *ICDM Workshop on Privacy, Security and Data Mining*, 2002

[29] K. Wang, Y. Xu, P. S. Yu, and R. She. Building decision trees on records linked through key references. *SIAM International Conference on Data Mining (SDM),* 2005.