Scalability via summaries: Stream Query Processing Using Promising Tuples

M. H. Ali

Walid G. Aref
Purdue University, aref@cs.purdue.edu

M. Y. IlTabakh

Report Number:
05-005
SCALABILITY VIA SUMMARIES: STREAM QUERY PROCESSING USING PROMISING TUPLES

M. H. Ali
Walid G. Aref
M. Y. ElTabakh

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #05-005
March 2005

[Image of the document page]
Scalability via Summaries: Stream Query Processing using Promising Tuples

M. H. Ali
Walid G. Aref
Y. ElTaibakh

Department of Computer Science, Purdue University, West Lafayette, IN
{mhali, aref, meltabakh}@cs.purdue.edu

Abstract
In many data streaming applications, streams may contain data tuples that are either redundant, repetitive, or that are not "interesting" to any of the standing continuous queries. Processing such tuples may waste system resources without producing useful answers. To the contrary, some other tuples can be categorized as promising. This paper proposes that stream query engines can have the option to execute on promising tuples only and not on all tuples. We propose to maintain intermediate stream summaries and indices that can direct the stream query engine to detect and operate on promising tuples. As an illustration, the proposed intermediate stream summaries are tuned towards capturing promising tuples that (1) maximize the number of output tuples, (2) contribute to producing a faithful representative sample of the output tuples (compared to the output produced when assuming infinite resources), or (3) produce the outliers or deviant results. Experiments are conducted in the context of Nile [24], a prototype stream query processing engine developed at Purdue University.

1 Introduction

Recently, many applications, e.g., monitoring network traffic, sensor network applications and retail store online transaction processing, have moved from the traditional database paradigms to a more challenging paradigm in which data evolves infinitely and unpredictably over time to form streams of data. Although traditional DBMSs have reached some level of maturity in processing various types of queries over their persistent data layer, data stream systems are still evolving to cope with the challenging nature of the streaming environment. Scalability in terms of the number of streams, stream rates, and number of standing continuous queries that the system can handle is still a major challenge for data stream systems. In this paper, we propose a framework for stream query processing engines that achieves scalability via the notion of promising tuples. Promising tuples limit the attention of the query processor to a smaller subset of the input tuples that preserve the output features with respect to a specific query preference.

1.1 Motivation

By observing the distribution of tuples in the answer of various queries, we notice that tuples contribute with different frequencies to the output. The behavior of the output is usually dominated and shaped by a smaller subset of the tuples. Focusing on these tuples, the promising tuples, utilizes the available time budget effectively and produces output with desirable features.

Consider a join operation between the two streams in Figure 1 over the depicted time-window. Assume that due to scarcity of resources, we are limited to processing only three tuples from Stream 1. What would be a smart choice of those three tuples? We need to assume a specific query preference before we decide which tuples to choose. For example, consider the following cases: (1) If we take all the three tuples to be the "5"s, we maximize the number of output tuples (a total of 6 output tuples). (2) If we take two tuples to be "5" and one tuple to be "g", we get a total output of 5 tuples. However, in the latter case, the output looks more representative to the exact output of the join operation. i.e., comes close to a random sample over the exact output. (3) If we take the value "3" that rarely appears in the streams as one of the tuples, the output sounds good to some applications that keep track of irregular behavior to detect outliers.

To map the above example to real life imagine that the two streams are coming from the online transactions of two retail stores, where the tuple values rep-
Figure 1: A two stream example

Figure 2: Promising-tuple based processing

1.2 Contributions

The main contributions of this paper can be summarized as follows:

1. We propose the notion of promising tuples to utilize the available budget of CPU time effectively in processing tuples that contribute heavily to the output.
2. We propose an intermediate stream representation that is capable of indexing the stream's promising tuples.
3. We tune various stream operations to adopt the notion of promising tuples.
4. We conduct an experimental study to evaluate the effectiveness of promising tuples inside the Nile stream query processing engine [24].

The rest of this paper is organized as follows. Section 2 introduces the concept of promising tuples. Section 3 presents the proposed intermediate stream representation. Section 4 explains how various types of operations are performed on top of the intermediate stream representation. An experimental study that is based on a real implementation of the promising tuple approach inside Nile is given in Section 5. Section 6 overviews related work. Finally, the paper is concluded in Section 7.

2 Promising Tuples

In this section, we introduce the concept of promising tuples and present a framework for query processing using the notion of promising tuples. Promising tuples can be defined as follows:

Definition 1 Let Q be a query over stream X = \{x_1, x_2, x_3, \ldots\} and let O = \{o_1, o_2, o_3, \ldots\} be the answer to the query Q (i.e., Q(X) = O). Let \( P_Q \) be a preference function of query Q that is defined over its answer set O such that \( P_Q(O) = O_Q \) where \( O_Q \subseteq O \). A promising tuple is a tuple \( x_p \in X \) such that \( Q(x_p) = \psi \) where \( \psi \in O_Q \).
CREATE SUMMARY summary-name ON stream (attr) WITH GRANULARITY G1 AND COLLAPSE(m, rate, depth)

(b)

SELECT attr1, attr2, ...
FROM stream1, stream2, ...
WHERE conditions
WITH PREFERENCE [scope: preference, collapse: P_i, factor: f]
WINDOW w

Figure 3: Extended SQL statements

response to a create-summary statement as shown in Figure 2a. The create-summary statement instructs the summary manager to divide the streams into time granularities and to summarize each granularity. High-resolution summaries are built over the most recent granularity that is of length $T_1$ time units. The collapse clause specifies two parameters: the rate and the depth. The quality of the summaries decreases or collapses as we go to older granularity according to the rate parameter. The depth parameter avoids the indefinite growth of the summaries and specifies how far the summaries are maintained over the past.

Sets of promising tuples are extracted from summaries in response to an SQL continuous query over a sliding window of size $w$ with a preference clause as shown in Figure 2b. The preference clause specifies how sets of promising tuples are extracted and prioritized from the stream summaries. The priority of a promising tuple reduces or collapses as we move in the past. Notice that the stream summaries are stream-dependent structures and are constructed regardless of the standing queries while the stream summaries are maintained over the past.

Sets of promising tuples are extracted from summaries in response to an SQL continuous query over a sliding window of size $w$ with a preference clause as shown in Figure 2b. The preference clause specifies how sets of promising tuples are extracted and prioritized from the stream summaries. The priority of a promising tuple reduces or collapses as we move in the past. Notice that the stream summaries are stream-dependent structures and are constructed regardless of the standing queries while the stream summaries are maintained over the past.

Sets of promising tuples are extracted from summaries in response to an SQL continuous query over a sliding window of size $w$ with a preference clause as shown in Figure 2b. The preference clause specifies how sets of promising tuples are extracted and prioritized from the stream summaries. The priority of a promising tuple reduces or collapses as we move in the past. Notice that the stream summaries are stream-dependent structures and are constructed regardless of the standing queries while the stream summaries are maintained over the past.

Promising tuples focus on the CPU processing cycles as a primary resource. The interarrival time between two consecutive tuples is our processing budget where useful computations can take place. To quantify how many output tuples can be generated during one interarrival time period, we introduce the following measure of performance:

$$N_{out} = \frac{\tau - t_{in}}{t_{in}}$$  \(1\)

where $N_{out}$ is the number of output tuples per input tuple, $\tau$ is the average interarrival time between two consecutive tuples, $t_{in}$ is the time required to insert the incoming tuple into the intermediate stream representation, and $t_{in}$ is the average time required by the intermediate representation to guide the search to one output tuple. Initially, we invest a little portion of the budget interarrival time ($\tau$) to reduce the output tuple hit time ($t_{out}$). Smart guidance to the stream promising regions reduces effectively the output hit time, which in turn increases the number of generated output tuples ($N_{out}$).

3 Stream Intermediate Representation

Given the create-summary statement of Figure 2a, it is required to build an intermediate stream representation that indexes the incoming stream tuples. The intermediate stream representation keeps track of incoming tuples plus some summaries to direct stream operations to their promising regions. We propose an intermediate stream representation that has the following two properties:

1. The support of multiple time granularities, $G_i$, $G_2$, ..., $G_m$ (Figure 4a) such that:

   (a) The number of granularities $m$ corresponds to the depth of the collapse clause in the create-summary statement (i.e., $m = depth$).

   (b) Granularities span regions in the stream's timeline with sizes that are multiples of each other (i.e., $G_{m+1} = rate \times G_m$, $G_1 = T_1$), where $T_1$ and $T_2$ are specified in the create-summary statement.

2. The capability to index each time granularity using a multi-resolution index structure as illustrated in Figure 4b (Section 3.4).

Multiple-time granularities are desirable to make up for the bounded memory requirements. Notice that the summary size is fixed in each granularity despite the increase in the size of the region that is covered by that granularity. As data gets older, it moves from one granularity to the next and a coarser summarization is applied to the data due to the increase in the granularity size. This behavior results in a gradual degradation in the resolution of summaries as we go farther in the past. The collapse clause in the create-summary statement specifies the rate at which summaries collapse and how deep summaries can go in the past. Summaries in each granularity are indexed using a multi-resolution index structure in order to allow coarser resolutions to guide the search down the hierarchy to finer resolutions.

3.2 Two-dimensional Representation of Data Streams

Each granularity in the stream is divided into smaller partitions in order to build more accurate summaries over those partitions. There are two ways to partition a stream:

1. Partition the stream granularity into sub-set-based buckets by hashing on the values of incoming tuples.
3.2 Summary Information

In each granularity cell, we maintain summaries to capture the behavior of the stream tuples in this cell. A summary abstract data type (summary-ADT) is added to the system to encapsulate the functionalities of summarization techniques. Various types of summaries have been studied in the literature, e.g., histograms [21], wavelets [18], samples [7], and aggregates [34]. In our work, the summarization technique is considered to be a black box as long as it provides the following set of interface functions:

1. **Summarize(x)**, to insert a tuple x and to add the effect of its value to the summaries.

2. **Partition the stream granularity into time-based zones.** Each zone corresponds to a contiguous time zone of the stream.

   In the first approach, buckets that are based on the tuple values span the timeline of the stream. Consequently, summaries may suffer inaccuracies due to the change in the stream's behavior over time. In the second approach, a time zone contains all tuple values in this portion of the stream. Summaries may suffer inaccuracies due to mixing the summaries of a wide range of values in the same zone.

3. Instead of using a one-dimensional space, we propose to hash the stream tuples over two dimensions: (a) the value dimension and (b) the time dimension (Figure 4c). The two-dimensional representation of one granularity can be summarized as follows:

   1. The value dimension is divided into buckets based on a suitable hash function over the tuple values.

   2. The time dimension is divided into time zones based on the stream behavior. A change in the stream's behavior implies a new time zone (Section 3.3).

   3. A granularity cell is the intersection of a value bucket with a time zone (Figure 4d).

   4. For each granularity cell:
      (a) Tuple values are materialized and are linked in the order of their arrival time.
      (b) Summaries are built to provide a rough but quick estimation of the tuples in that cell (Section 3.2).
      (c) A multi-resolution index is built to speed up the access to summaries (Section 3.4).
      (d) Sets of promising tuples are extracted based on query preferences (Section 4.1).

4. **Estimate(x)**, to estimate various properties of a stream tuple x from the summaries. These estimations are used in the context of the query preference function to specify how promising tuples are extracted.

5. **Confidence()**, to report how much confident we are about the stream summaries. This function is used as a measure of accuracy.

6. **Add(r1, r2)**, to merge the summaries of region r1 and r2 into one summary structure. This function is used to build coarser summaries upon merging the two regions.

7. **Subtract(r1, r2)**, to subtract the summaries of region r2 from the summaries of region r1. This function is used to remove the effect of the summaries of region r2 once it expires, i.e., goes outside the time-window of interest.

For implementation purposes, we adopt the countsketch technique as presented in [12]. Countsketches provide an estimation of both the absolute and the relative frequencies of the stream tuples. Countsketches are efficient with respect to the update and the search operations. The Confidence function takes the variance of the items in the countsketch as a measure of accuracy. Another interesting feature of countsketches is that they are additive. Two countsketches can be combined together in linear time simply by adding the two sketches. Similarly, the effect of one countsketch can be subtracted from another countsketch in linear time. Other forms of summarizations, e.g., histograms, can be addressed to define the same interface functions.

3.3 Self-adjusting Time Zones

As mentioned in Section 3.1, the stream is partitioned over the infinite time dimension into time zones. Each zone can have a fixed predefined size. However, fixing the size of each zone may result in zones containing various behaviors of the stream which in turn degrades the accuracy of the summaries.

To improve the accuracy of the summaries, we partition the timeline into zones based on the change...
in the stream's behavior. We sense a change in the stream's behavior through the Confidence function of the summation technique. Once a tuple arrives, it is inserted into the summaries of the current zone. Then, the Confidence function is evaluated. If the confidence is below a certain threshold (\( \theta \)) or the length of a time zone exceeded \( M_{L \text{LENGTH}} \), a new time zone is started and its summaries are initialized. For a more efficient implementation of detecting changes in the stream's behavior, the reader is referred to [26].

3.4 Multi-resolution Index

Our intermediate stream representation makes use of a multi-resolution index to provide an efficient access to stream tuples. Coarser resolutions of summaries are formed by the aggregation of the fine-resolution summaries. The top-level summarization maintains the coarsest summaries that provide a rough estimation of how much a tuple isprecedented. Then, we follow the hierarchy of the multi-resolution summaries till we reach the exact regions where the tuple resides.

For each stream granularity, we maintain a pyramid-like structure [31] to index this granularity (Figure 4b). The lowest-level cells of the pyramid contain the stream tuples plus their summaries. The lowest-level cells (LLC) are on the form:

\[
\text{LLC (tuple-list: i) Summary-info)
\]

as we go up in the pyramid, the summary information are aggregated using the add function and pointed to their original cells are maintained. Upper level cells (ULC) are on the form:

\[
\text{ULC (Aggreg-Summar-info, cell-pointer ... j)
}\]

3.5 Multiple Granularities

As mentioned at the beginning of this section, we have \( m \) time granularities: \( G_1, \ldots, G_m \), with \( G_1 \) being the most recent granularity and \( G_m \) being the oldest granularity (Figure 4a). Each granularity (except \( G_1 \)) is divided over the time dimension into \( n \) time zones. \( G_i \) is divided into \( x \) time zones based on the stream's behavior as explained in Section 3.3. \( G_1 \) spans the most recent portion of the stream that is of length \( T_1 \). \( T_1 \) is a parameter that is specified by the create-summary statement (Figure 4a). This parameter defines the base granularity of the stream on which other granularities will be based. Granularity \( G_2 \) spans a portion of the stream that is of length \( T_2 \), such that:

\[
T_i = n \times T_{i-1}, i = 2, \ldots, m.
\]

Notice that old granularities span larger portions of the stream than recent granularities. As time proceeds, all the cells of \( G_{m-1} \) are combined and their summaries are compressed into one cell of \( G_1 \). Figure 5 illustrates the movement of data between two granularities of the stream. Let \( G_i \) be granularity number \( i \),

**Procedure COLLAPSE(CZ \(_{i+1}, CZ_i \))**

1. \( CZ_i = CZ_i + 1 \mod n \)
2. Add(CZ \(_{i+1}. \text{summary-info}, CZ_i. \text{summary-info})
3. Concatenate(CZ \(_i+. \text{tuple-list}, CZ_{i+1}. \text{tuple-list})
4. CZ_i. Summary-info= NULL
5. CZ_i. tuple-list= NULL
6. update the multi-resolution indices over \( G_i \) and \( G_{i+1} \)

**END**

Figure 5: Data movement between granularities of length \( T_i \) and let \( CZ_i \) be the current zone in granularity \( G_i \) that receives the incoming tuples. All \( CZ_i \) are initially set to 1 and are advanced in a round robin fashion (\( CZ_i = CZ_i + 1 \mod n \), where \( i \) is the number of time zones in a granularity). \( CZ_i, t \equiv 2, \ldots, m \) advances to the next time zone once the current time zone elapses, i.e., every \( T_i \) units of time. \( CZ_i \) advances to the next time zone once a change in the stream's behavior is detected.

Figure 5b lists the steps required to move \( CZ_i \) to the next time zone (\( CZ_i = 1 \mod n \)). In Step 1, we advance \( CZ_i \) (where \( i \geq 2 \)) to the next time zone. Notice that \( CZ_i \) advances based on a change in the stream behavior and returns to the first time zone in the granularity if the granularity size becomes \( T_i \). This is not mentioned in the figure for simplicity. In Steps 2 and 3, the summaries and data of \( CZ_i \) are pushed to the current time zone of granularity \( G_{i+1} \) (\( CZ_{i+1} \)). We free \( CZ_i \) from its summaries and data in Steps 4 and 5. Finally, the multi-resolution index of \( G_i \) is updated to reflect the added summaries, which are subtracted from higher level resolutions of granularity \( G_{i+1} \) (Step 6).

4 Data Stream Operations

Promising tuples can be applied widely to various stream operations. In this section, we explain how stream operations gain benefit from the promising tuple approach. We explore the join, selection, and aggregation operations as example operations. 
SELECT St.ItemNo, St.TimeStamp, St2.TimeStamp
FROM Store St, Store St2
WHERE St.ItemNo = St2.ItemNo
WITH PREPREFERENCE (desc|asc|weighted)
SUMMARY EstimateCount(St.ItemNo)
FADE FACTOR α
WINDOW w

Figure 6: An example window join query

4.1 Window Join

Given two streams and a window specification, Figure 6 gives the SQL query that performs a window join over the two streams. The two streams are generated from the online transactions of two retail stores. Each stream contains items at a stream number and its associated timestamp. The preference function is based on the summary-estimated counts of the item number in both streams. One out of three preference specifiers (i.e., desc, asc, or weighted) is used to tune the preference function (Section 4.1.1). The fade factor reduces the priority of the time granularities by a factor of α as we move in the past. The window clause specifies the size of the sliding window of interest. If no window clause is specified, the window is assumed to span the whole stream that is seen so far. The details of the query preferences and the join algorithm are given in the next sections.

4.1.1 Query Preferences

The preference clause in the query syntax bridges the gap between the general-purpose stream summaries and the query-specific processing tuples. Consider the following three preference specifiers and how they contain the way the priorities are handled in the SQL query of Figure 6:

1. Desc. The promising tuples are extracted and processed based on the descending order of the tuple. Count estimates to give frequent tuples more priority. Focusing on frequent tuples maximizes the number of output tuples.

2. Weighted. The CPU time is divided among the promising tuples based on the ratio of their Count estimates. This approach avoids the case where the CPU gets monopolized by high frequency tuples and deviant tuples. Our notion of outliers considers a tuple to be an outlier if it occurs with a low frequency, e.g., a sensor triggering a false alarm or the purchase of a rare item. Similarly, other notions of outliers can be extracted from various preference functions.

3. Asc. The positioning tuples are extracted and processed based on the ascending order of the tuples' Count estimates to give the least frequent tuples more priority, which leads to producing outliers and deviant tuples. Our notion of outliers considers a tuple to be an outlier if it occurs with a low frequency, e.g., a sensor triggering a false alarm or the purchase of a rare item. Similarly, other notions of outliers can be extracted from various preference functions.

The query processor may consider time in determining the priority of a tuple, i.e., by favoring output tuples with more recent timestamps. The fade factor α in clause the query syntax instructs the query processor to decrease the priority of a promising tuple based on its time granularity. If a promising tuple is extracted from granularity i, its priority is decreased by a factor of αi.

4.1.2 Window Join Algorithm

Our window join algorithm is a variation of the symmetric hash join algorithm [33] that is tuned to benefit from the promising tuple approach. When a tuple arrives at the system, (1) it is inserted into the summary of its stream, and (2) it probes the summaries of the other stream looking for join matches. This process is symmetric with respect to each stream. When there are hundreds of queries in the system, our window join algorithm considers the join probes only once stream (i.e., tuples of stream S2 probe stream S1). In Step 1, the tuple is inserted into its stream summaries. In step 2, a pointer is placed at the first granularity of the other stream. If the granularity intersects with the join window (Step 3), we investigate all its time zones (Step 3.a). We estimate the priority of ξk in these zones and insert (ξk, the time zone, the priority) into a priority queue. We weight the priority by how old the granularity is (i.e., by a factor of αi). Notice that the multi-condition index on top of each granularity stores many time zones where the tuple is likely to occur. This step is not shown in the figure for simplicity. We advance the pointer to the next granularity (Step 3.b). In step 4, the priority queue (PQ) is traversed and the actual join probes take place until a time out occurs. The join process time out the joining process once another tuple arrives at the system. The query processor revisits the priority queue again once it becomes idle. The query processor is operated over standing queries in a round robin fashion. The priority queue is traversed in a no time order of their timestamps.

4.2 Selection

Consider the simplest case of restricting the selection problem to one selection predicate that is applied over one stream. If the stream is the system, then the optimal solution is to apply the selection predicate over each individual tuple once it arrives at the system. However, if we have hundreds of queries such that each query has an additional selection predicate, each query will lose some of the input tuples due to the limited CPU time that is allocated to each query. The major problem with the above approach is the random
Procedure JOIN-PROBE
Input. given two streams $S_1$ and $S_2$, a time-window $w$, a tuple $t_1 \in S_1$
Output. the join pairs on the form $(t_1, t_2)$ $S_1 \times S_2$ and $t_1, t_2 \in [t_1: t_1 + w]$

1. $S_1$.Summarize(t1)
2. $=$
3. while($S_2.[] [\tau - w + 1, \tau) \neq 0$)
   (a) for $j \in \{1, \ldots, \tau\}$
      i. $\text{Priority} \leftarrow S_2.[] \{j\} \cdot \text{Estimate}(t_2)$
      ii. $PQ.\text{Insert}(i, S_2.[] \{j\}, \text{Priority})$
   (b) if $i = \tau$
      i. while (NOT TimeOut()) $PQ.\text{Probe}()$

Figure 7: Window Join Algorithm.

Procedure SELECTION

1. $t_{max} \leftarrow \infty$
2. while (TRUE)
   (a) $t_{start} \leftarrow t_{end}$
   (b) $\text{Process the selection predicate on top of summarizations over the region } \{t_{start}, t_{end}\}$
   (c) $\text{Prioritize regions based on their promise.}$
   (d) $\text{Process the selection predicate over the identified promising regions.}$
   (e) $t_{max} \leftarrow t_{end}$
END.

Figure 8: The selection operation.

4.3 Aggregates

The behavior of an aggregate is usually dominated by a small subset of tuples, i.e., the promising tuples in our notion. The challenge is to specify which set of tuples dominates the behavior of the aggregate. For example, the behavior of the average and the median aggregates seems to be preserved under a faithful representative subset of tuples. The preference of an average or a median query may look like: [WITH PREFERENCE weighted SUMMARY.Estimate.Count(Bl.ItemNo)]. Similarly, other aggregates can define their own notions of promising tuples. In response to the aggregate queries, the query processor will extract sets of promising tuples. Then, the regions that are associated with these sets of promising tuples are visited and aggregated.

5 Experiments

In this section, we provide an experimental evidence that the promising tuple approach enhances the performance of stream query processing engines. Four sets of experiments are conducted. In Section 5.1, we compare the performance of different forms of intermediate stream summaries. Then, we devote a section for each stream operation. In Section 5.2, we compare the efficiency of the join selection and aggregate operations is addressed in Sections 5.2, 5.3, and 5.4, respectively. Unless mentioned otherwise, data streams are generated using logs of online retail transactions on the form (StoreID, TransactionID, TimeStamp, ItemID, Price). Retail transactions are extracted from three-month logs of Walmart retail stores. We playback the data stream logs such that the interarrival time between two consecutive tuples follows the exponential distribution with an average of 0.1 second. Stream operations are interacted in a one-minute sliding window over the most recent portion of the stream. All the experiments in this system are based on a real implementation of the promising tuple approach inside the Nile stream query processing engine [24]. The Nile engine executes on a machine with Intel Pentium IV, 2.4GHz with 512MB RAM running Windows XP.

5.1 Performance of intermediate stream summaries

In this section, we compare the performance of four intermediate stream summaries: (1) T1, the naïve approach where no summaries are maintained. (2)
T2, where a one-dimensional summary structure is built over the both the time and the value dimension. (4) T4, where a two-dimensional summary structure is built over the both the time and the value dimension. (4) T4, where a two-dimensional summary structure is built over the both the time and the value dimension.

Our major measure of performance is the number of generated output tuples per input tuple as mentioned in Section 2. Equation 2 gives the number of output tuples per input tuple in terms of the tuple insertion time \( t_{\text{ins}} \) and the tuple hit time \( t_{\text{hit}} \). In Figure 9, we perform a join operation between two streams of retail transactions and we monitor both the \( t_{\text{ins}} \) (Figure 9a) and the \( t_{\text{hit}} \) (Figure 9b) parameters along with the number of output tuples per an input tuple (Figure 9c). Notice that as techniques get more sophisticated, the insertion cost increases. However, the output tuple hit time improves and dominates the overall performance by increasing the total number of output tuples.

In some cases, it is cheaper and more efficient to operate on the two streams and to avoid the summarization overhead. For very high rate streams, the system may end up being all its processing cycles in data summarization. In Figure 10a, we conduct an experiment to measure the performance of the join operation between two streams with small average interarrival times (\(< 20\,\text{msec}\) ). Less sophisticated techniques with small insertion cost become more efficient than the techniques with high insertion cost. Similarly, if we are interested in a small window size, it may be cheaper to traverse the window tuple by tuple. Figure 10b evaluates the performance of the join operation under a time window that varies from 10 seconds till 2 minutes with 20 second increments. Notice that the performance gains are obtained from summary-based structures as the window gets larger.

5.2 Performance of Window Join

In this section, we study the performance of the join operation under various notions of promising tuples. The average join selectivity between the two data streams (that are drawn from real data sets) is found to be 6.7%. We compare the performance of the promising tuple approach to the symmetric hash join (SIMPLE-SHJ) that is presented in [33]. Consider the following notions of promising tuples:

1. **Maximizing the number of output tuples**

Figure 11 evaluates the number of output tuples that are generated by both the simple symmetric hash join (SIMPLE-SHJ) and the promising-tuple join (PT-Join) over a one-minute sliding window. Our measure of performance is the percentage of the exact output tuples that is obtained at run time. The figure shows that the promising tuple approach uses its processing time effectively and focuses on the tuples that maximizes the number of output tuples.

Figure 12 evaluates the performance of the join operation under a whole-stream window. In addition to the number of output tuples (Figure 12a), we monitor the freshness of the data as another measure of goodness. (Figure 12b) assesses the average difference in timestamps between the two join components as our measure of freshness (\( \text{avg} \cdot \text{interarrival time} \)). This measure indicates how far a tuple goes in the past to look for a join match. From the figure, notice that the promising tuple approach favors recent and fresh tuples over old ones. This behavior is achieved by reducing the window size of old granularities by a factor of \( e \) (so it is set to be 80% in this experiment). SIMPLE-SHJ has no way to know in advance whether an incoming tuple will join with a recent tuple or with an old one.

2. **Providing a faithful representation of the...**
The performance of aggregates is tested on top of the hundred selection queries that are experimented in Section 5.3. Aggregates are computed using both the traditional approach that executes each query separately over each stream and the promising-tuple approach. The mean square error (MSE) between the computed aggregate and the exact aggregate is used as our measure of performance.

For the sake of illustration, we show the performance of the selection operation in Figure 15. Figure 15a shows hundred queries, each performs a different selection predicate over the same stream. The selectivity of each predicate ranges uniformly from 5% to 10%. The experiment is conducted using both the naive selection approach (SIMPLE-Selection) and feeds the stream to each query separately, and the promising tuple approach which makes use of summaries. The promising tuple approach aims at maximizing the number of output tuples. The percentage of retrieved tuples relative to the size of the exact result is calculated for each query and the average over all queries is taken as our measure of performance. The figure shows that the promising tuple approach produces up to 33% output tuples over the SIMPLE selection (at 100 msec average interarrival).

In Figure 15c, the number of concurrent queries is varied from 20 to 200 to explore the performance of the selection operation under various query loads. The SIMPLE-Selector is efficient at low system loads. However, the promising tuple approach outperforms the SIMPLE-Selector with the increase in the query load (more than 80 concurrent queries).

5.4 Performance of Aggregates

The performance of aggregates is tested on top of the hundred selection queries that are experimented in Section 5.3. Aggregates are computed using both the traditional approach that executes each query separately over each stream and the promising-tuple approach. The mean square error (MSE) between the computed aggregate and the exact aggregate is used as our measure of performance.

For the sake of illustration, we show the perfor-
Figure 13: Output faithful representation

Figure 14: Outlier detection

6 Related Work

In this section, we overview related work in the context of data stream processing through three major lines. First, we list major data stream systems that have been developed by various research groups. Second, we survey summarization techniques that are deployed widely to enhance query performance. Third, we give examples of stream operations that are explored in literature.

Prototype data stream systems have recently gained a lot of research interest. Stanford STREAM [8, 28] addresses the problem of resource management in the context of data stream processing. Quality of service (QoS) has been investigated in the AURORA project [1]. The Niagara project [14] adopts the notion of continuous queries and how group-optimizations can be performed over those queries. Telegraph [11] has an adaptive query engine where the execution order of query operators can change during execution. A framework for query execution plans over data streams is suggested by the Fjord project [27]. COUGAR [9] introduces a new data type for sensors. Gigascope [15] is another data stream system developed at AT&T to process streams of network traffic.

With the limited CPU time and bounded memory constraints [5], approximate answers that are obtained via summaries are accepted in place of the exact ones. Sampling [7], histograms [21], and wavelets [10, 32] can represent a stream using lower memory requirements. Streams are also summarized using sketches [4]. Some of the sketching techniques have the capability to maintain a stream's most frequent items [12]. Sketch-based processing and sketch sharing among multiple queries are presented in [19]. The concept of multiple granularities over data streams is proposed in [35]. The work in [29] suggests a gradual degradation of the summary resolution as data tuples move from coarse granularity to the next. This gradual degradation is referred to by the term amnesic stream approximation. Statistical models are utilized in [18] to query sensors interactively. Based on outstanding queries, statistical models provide an estimate for a sensor reading and tell how much this estimate is accurate.
Promising tuples are those tuples that contribute heavily to satisfying a specific preference in the query answer. Example query preferences include: (1) the maximization of the number of output tuples, (2) producing a faithful representative sample of the output tuples, or (3) producing the outlier or deviant tuples in the result. Other notions of promising tuples can be flexibly defined to satisfy the query requirements. The promising tuples may also consider the freshness of the stream tuples. We proposed stream intermediate summaries that are capable of identifying the promising tuples of a stream. The intermediate summaries are organized in a multi-resolution index that slides in steps over the stream. We proved the applicability of the promising tuple approach in the context of various stream operations, e.g., join, selection, and aggregate operations. Experimental results show that the notion of promising tuples increases the effectiveness and resource utilization of a stream query processing engine. Experiments are based on a real implementation of the proposed summaries as part of the summary manager inside the Nile stream query processing engine.

References


