

2005

# Scalability via summaries: Stream Query Processing Using Promising Tuples

M. H. Ali

Walid G. Aref

*Purdue University*, [aref@cs.purdue.edu](mailto:aref@cs.purdue.edu)

M. Y. IITabakh

Report Number:

05-005

---

Ali, M. H.; Aref, Walid G.; and IITabakh, M. Y., "Scalability via summaries: Stream Query Processing Using Promising Tuples" (2005).  
*Computer Science Technical Reports*. Paper 1620.  
<http://docs.lib.purdue.edu/cstech/1620>

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact [epubs@purdue.edu](mailto:epubs@purdue.edu) for additional information.

**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**

# Scalability via Summaries: Stream Query Processing using Promising Tuples

M. H. Ali

Walid G. Aref

M. Y. ElTabakh

Department of Computer Science, Purdue University, West Lafayette, IN  
{mhali, aref, meltabak}@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 outlier 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 paradigm 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 “9”, 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-

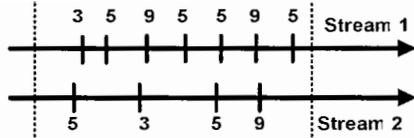


Figure 1: A two stream example

resent identifiers of the sold items. Maximizing the number of output tuples gives as much points of similarity as possible between the two stores. A faithful representative output avoids excessive duplication of high frequency elements and produces an output that spans the exact output set to give a near-random sample of the output. Low-frequency tuples may express outliers among the two stores, e.g., selling a very rare or a very expensive item in the two stores over a short period of time.

## 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 we 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  $X = \{x_1, x_2, x_3, \dots\}$  and let  $O = \{o_1, o_2, o_3, \dots\}$  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  $O$  such that  $P_q(O) = O_p$  where  $O_p \subset O$ . A promising

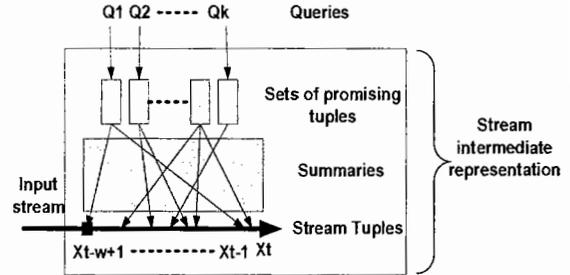


Figure 2: Promising-tuple based processing

*tuple* is a tuple  $x_p \in X$  such that  $Q(x_p) = o_p$  where  $o_p \in O_p$ .

The definition above indicates that a query  $Q$  may have a preference function that is defined over its exact answer  $O$  to yield a preferred answer set  $O_p$ . A promising tuple is the tuple that contributes to producing an output tuple that is part of the query preferred answer set. A promising region is a region in the stream that contains plenty of promising tuples. These promising regions are the regions that are worth processing. We aim at investing the system's resources in processing promising regions to satisfy the query preferences.

Figure 2 illustrates the promising-tuple approach. In the promising tuple approach, we maintain an intermediate stream representation for the largest portion of the stream that is of interest to any query. Once a new tuple arrives, it is inserted in the intermediate representation where summaries are built on top of the stream tuples to capture the behavior of the stream. Sets of promising tuples are extracted from the stream summaries based on the preferences of standing queries. An entry in the promising tuple set is on the form

*(promising tuple, promising region, priority)*

Each promising tuple points to the stream regions where it can be found with a specific priority to indicate how much a region is rich in this promising tuple. The framework of query processing using the notion of promising tuples is a three-step process and can be summarized as follows:

1. Identify and extract from summaries the query's promising regions.
2. Prioritize the query's promising regions
3. Execute query on promising regions in the order of their priorities.

Notice that the promising tuple approach results in an out-of-order query answer because stream regions are processed based on their promise regardless of their arrival order.

The stream intermediate summaries are capable of indexing a stream's promising regions and are built in

```
CREATE SUMMARY summary-name ON stream (attr)
WITH GRANULARITY  $T_1$  AND COLLAPSE(rate,
depth)
```

(a)

```
SELECT attr1, attr2, ...
FROM stream1, stream2, ...
WHERE conditions
WITH PREFERENCE [asc|desc|weighted]  $P_q$ 
FADE FACTOR  $\alpha$ 
WINDOW  $w$ 
```

(b)

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 stream 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 tuple 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 *fades* with a factor of  $\alpha$  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 promising tuple sets are query dependent and are constructed based on a query preference ( $P_q$ ).

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:

$$N_{o/p} = \frac{\tau - t_{ins}}{t_{hit}} \quad (1)$$

where  $N_{o/p}$  is the number of output tuples per input tuple,  $\tau$  is the average interarrival time between two consecutive tuples,  $t_{ins}$  is the time required to insert the incoming tuple into the intermediate stream representation, and  $t_{hit}$  is the average time required by the intermediate representation to guide the search to one output tuple. Basically, we invest a little portion of the budget interarrival time ( $\tau$ ) in data summarization

( $t_{ins}$ ) to reduce the output tuple hit time ( $t_{hit}$ ). Smart guidance to the stream promising regions reduces effectively the output hit time, which in turn increases the number of generated output tuples ( $N_{o/p}$ ).

### 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,  $G_1, G_2, \dots, G_m$  (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.,  $m = \text{depth}$ ).
  - (b) Granularities span regions in the stream's timeline with sizes that are multiples of each other (i.e.,  $|G_{i+1}| = \text{rate} \times |G_i|$ ,  $|G_1| = T_1$ ), where *rate* and  $T_1$  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.

#### 3.1 Two-dimensional Representation of Data Streams

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

1. Partition the stream granularity into *value-based* buckets by hashing on the values of incoming tuples.

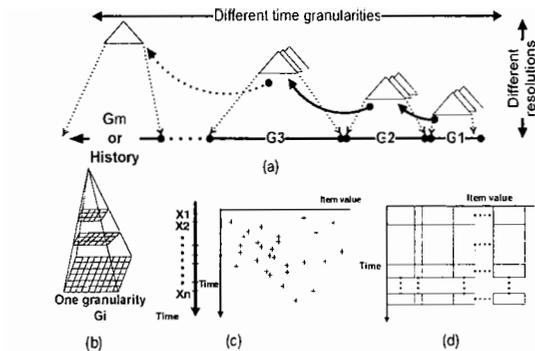


Figure 4: Stream intermediate representation

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.

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).

### 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 [10], 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. *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.
3. *Confidence()*, to report how much confident we are about the stream summaries. This function is used as a measure of accuracy.
4. *Add( $r_1.r_2$ )*, to merge the summaries of regions  $r_1$  and  $r_2$  into one summary structure ( $r_1$ ). This function is used to build coarser summaries upon merging the two regions.
5. *Subtract( $r_1.r_2$ )*, to subtract the summaries of region  $r_2$  from the summaries of region  $r_1$ . This function is used to remove the effect of the summaries of region  $r_2$  once it expires, i.e., goes outside the time-window of interest.

For implementation purposes, we adopt the counts-ketch 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 summarization 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 *MAXLENGTH*, 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 is promising. 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:

$$LLC (tuple-list[\dots], Summary-info).$$

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

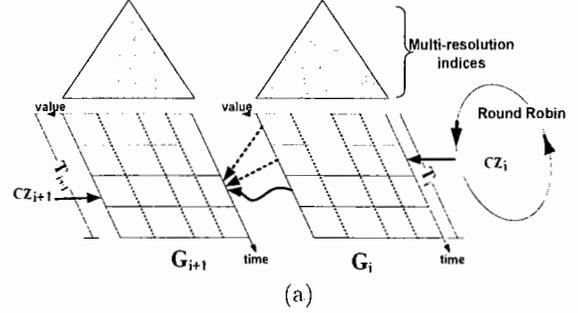
$$ULC (Aggregate-Summary-Info, cell-ptrs[\dots])$$

### 3.5 Multiple Granularities

As mentioned at the beginning of this section, we have  $m$  time granularities:  $G_1, \dots, G_m$  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.  $G_1$  is divided into  $n'$  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 2a). This parameter defines the basic granularity of the stream on which other granularities will be based. Granularity  $G_i$  spans a portion of the stream that is of length  $T_i$  such that:

$$T_i = n \times T_{i-1}, i = 2, \dots, m.$$

Notice that old granularities span larger portions of the stream than recent granularities. As time proceeds, all the cells of  $G_{i-1}$  are combined and their summaries are compressed into one cell of  $G_i$ . 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.  $CZ_i = CZ_i + 1 \bmod n$
2. Add( $CZ_{i+1}$ .summary-info,  $CZ_i$ .summary-info)
3. Concatenate( $CZ_{i+1}$ .tuple-list,  $CZ_i$ .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.

(b)

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  $CZ_i$ ’s are initially set to 1 and are advanced in a round robin fashion ( $CZ_i = CZ_i + 1 \bmod n$ , where  $n$  is the number of time zones in a granularity).  $CZ_i, \forall i = 2, \dots, m$ . advances to the next time zone once the current time zones elapses, i.e., every  $\frac{T_i}{n}$  units of time.  $CZ_1$  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 ( $CZ_i + 1 \bmod n$ ). In Step 1, we advance  $CZ_i$  (where  $i \geq 2$ ) to the next time zone. Notice that  $CZ_1$  advances based on a change in the stream behavior and returns to the first time zone in the granularity if the granularity size becomes  $T_1$ . 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  $G_{i+1}$  is updated to reflect the added summaries, which are subtracted from higher level resolutions of granularity  $G_i$  (Step 6).

## 4 Data Stream Operations

Promising tuples can be applied widely to various stream operations. In this section, we explain how stream operations can benefit from the promising tuple approach. We explore the join, selection, and aggregation operations as example operations.

```

SELECT S1.ItemNo, S1.TimeStamp, S2.TimeStamp
FROM Store1 S1, Store2 S2
WHERE S1.ItemNo= S2.ItemNo
WITH PREFERENCE [desc|asc|weighted]
      SUMMARY.Estimate.Count(S1.ItemNo).
      SUMMARY.Estimate.Count(S2.ItemNo)
FADE FACTOR  $\alpha$ 
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 tuple consists of an *item 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  $\alpha$  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 promising tuples. Consider the following three preference specifiers and how they control 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 tuples' *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 produces a near-optimal random sample.
3. **Asc.** The promising 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 fire 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 promise of a tuple, i.e., by favoring output tuples with more recent timestamps. The *fade factor*  $\alpha$  clause in 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  $\alpha^{i-1}$ .

### 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 summaries 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.

Figure 7 summarizes our join algorithm considering the join probes of only one stream (i.e., tuples of stream  $S_1$  probes stream  $S_2$ ). 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 priorities of  $x_1$  in these zones and we insert  $(x_1, \text{the time zone, the priority})$  into a priority queue. We weight the priority by how old the granularity is (i.e., by a factor of  $\alpha^{i-1}$ ). Notice that the multi-resolution index on top of each granularity prunes many time zones where the tuple is less 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 query processor times 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 iterates over standing queries in a round robin fashion. The priority queue is traversed in ascending, descending, or weighted order based on the preference specifier of the query (i.e., *asc*, *desc*, or *weighted*). As a drawback of this traversal, the output tuples are generated in no specific 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 this is all what we have in 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 a different 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  $x_1 \in S_1$ .

**Output.** The join pairs on the form  $(x_1, x_2)$  S.T.  $x_2 \in S_2$  and  $|x_1.ts - x_2.ts| \leq w$

1.  $S_1$ .Summarize( $x_1$ )
  2.  $i=1$
  3. while( $S_2.G_i \cap (\tau - w + 1, \tau) \neq \emptyset$ )
    - (a) for  $j=1$  to  $n$ 
      - i.  $Prio = \alpha^{j-1} \times S_2.G_i[j].Estimate(x_1)$
      - ii.  $PQ.Insert(x_1, S_2.G_i[j], Prio)$
    - (b)  $i = i + 1$
  4. while (NOT TimeOut())  $PQ.Probe()$
- END.

Figure 7: Window Join Algorithm.

### Procedure SELECTION

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

Figure 8: The selection operation

dropping of tuples. There is no control over which tuples a query will lose. Queries may be losing the most interesting tuples that satisfy their predicates.

The proposed summary-guided selection allocates some portion of its budget time to summarize the incoming stream tuples. Then, each query is guided by the summaries to its promising regions that are rich in terms of tuples that satisfy its search predicate. Following the same framework of the promising tuple approach, the system iterates over standing selection queries as described in Figure 8. To avoid reporting duplicate tuples in the answer, the system remembers the last tuple that is processed by each query. The next time a query is dispatched to the query processor, the query processor resumes the selection process from where it stopped last time.  $T_{start}$  and  $t_{end}$  mark the selection boundaries of the current iteration.

Notice that the selection operation need not have a *window* clause. In this case, the selection operation is a filter over the stream tuples. However, if a window clause with a window of size  $w$  is specified, the selected tuples are buffered for  $w$  time units. A tuple is reported to be expired and is invalidated from the query answer if its timestamp  $ts$  gets older than  $\tau - w$ , where  $\tau$  is the current timestamp.

### 4.3 Aggregates

The behavior of an aggregate is usually dominated by a smaller subset of tuples, i.e., the promising tuples in our notion. The challenge is to specify which set of tuples dominate 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(S1.ItemNo)*).

The summation aggregate introduces a new notion for promising tuples. It assumes that the behavior of the sum is dominated by the tuples that have a high *value*  $\times$  *frequency* product. The preference of a summation query may look like: (WITH PREFERENCE *desc SUMMARY.Estimate.Count(S1.ItemNo) \times S1.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 various forms of intermediate stream summaries. Then, we devote a section for each stream operation. The performance 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 interested in a one-minute sliding window over the most recent portion of the stream. All the experiments in this section 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, CPU 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 naive approach where no summaries are maintained. (2)

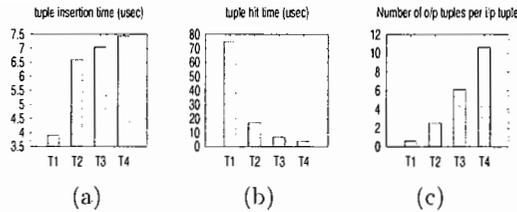


Figure 9: Cost parameters for various structures

T2, where a one-dimensional summary structure is built over the time dimension. (3) T3, where a two-dimensional summary structure is built over the both the time and the value dimensions. (4) T4, where a two-dimensional summary structure with self-adjusting time zones (as proposed in Section 3.3) is utilized.

Our major measure of performance is the number of generated output tuples per input tuple as mentioned in Section 2. Equation 1 gives the number of output tuples per input tuple in terms of the tuple insertion time ( $t_{ins}$ ) and the tuple hit time ( $t_{hit}$ ). In Figure 9, we perform a join operation between two streams of retail transactions and we monitor both the  $t_{ins}$  (Figure 9a) and the  $t_{hit}$  (Figure 9b) parameters along with the number of output tuples per an input tuples (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 raw stream and to avoid the summarization overhead. For very high rate streams, the system may end up losing *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 ( $< 50msec$ ). Less sophisticated techniques with small insertion cost become more efficient than the techniques with high insertion cost. Similarly, if we are interested in small window sizes, 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

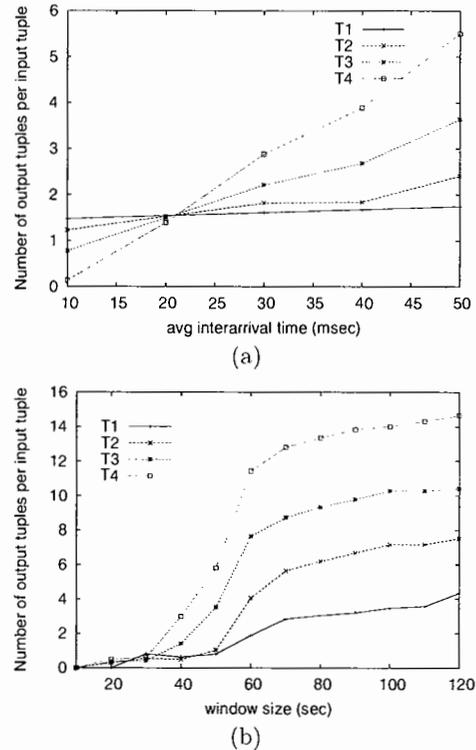


Figure 10: Effect of small interarrival time and small window size

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 that is obtained at run time. The figure shows that the promising tuple approach utilizes its processing time effectively and focuses on the tuples that maximize 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 ( $|ts1 - ts2|$ ). 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 priority of old granularities by a factor of  $\alpha$  ( $\alpha$  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

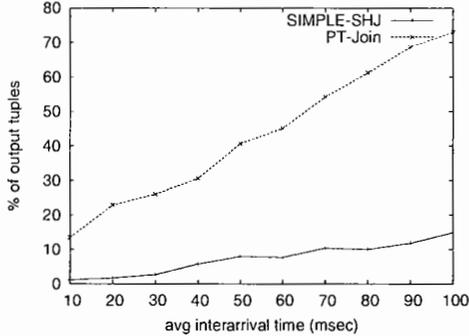


Figure 11: Sliding window join

### output

In Figure 13, we tune the promising tuples to provide a faithful representative set of the output. We take the *Hist-MSE* or the mean square error between the histograms of both the exact and the obtained answers as our measure of faithfulness. Let  $H_1$  be the histogram of the exact answer and let  $H_2$  be the histogram of the obtained answer. Each histogram is an equi-width histogram of  $k$  intervals ( $k$  is set to 100).  $H_1$  is divided into  $H_{11}, H_{12}, \dots, H_{1k}$  and  $H_2$  is divided into  $H_{21}, H_{22}, \dots, H_{2k}$ . Let  $N_1$  be the size of the exact answer and let  $N_2$  be the size of the obtained answer ( $N_1 \geq N_2$ ). The *Hist-MSE* is defined as follows:

$$Hist - MSE = \sum_{i=1}^k MSE\left(\frac{h_{1i}}{N_1}, \frac{h_{2i}}{N_2}\right) \quad (2)$$

Figure 13a shows that *PT-Join* still provides more output than *SIMPLE SHJ* while Figure 13b illustrates the superiority of *PT-Join* with respect to the *Hist-MSE* measure of performance.

### 3. Detecting low-frequency tuples and outliers

In Figure 13, we tune the promising tuples to detect low-frequency tuples. This notion of promising tuples has two desired features: (1) Less number of tuples are produced (Figure 14a). (2) Most of the produced tuples are true outliers (Figure 14b). In other words, the algorithm detects outliers and only outliers without being distracted by other tuples. In this experiment, a tuple is an outlier if its frequency is in the least 5% percentile of tuple frequencies.

### 5.3 Performance of the Selection Operation

In this section, we show the applicability of the promising tuple approach to the selection operation. 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*) that feeds the stream to each query separately, and the

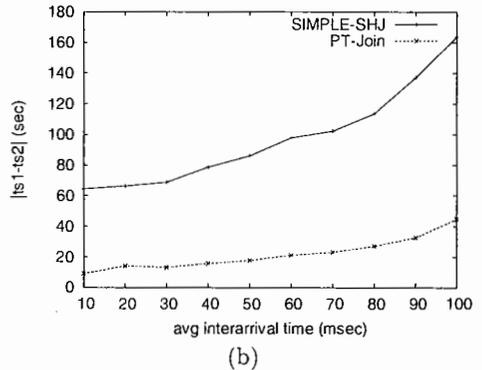
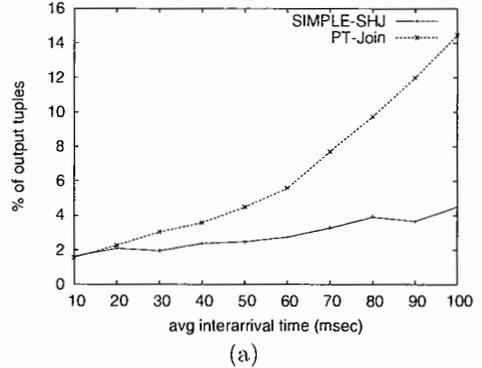


Figure 12: Whole-stream window join

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 15b, 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-Selection* is efficient at low system loads. However, the promising tuple approach outperforms the *SIMPLE-Selection* 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-

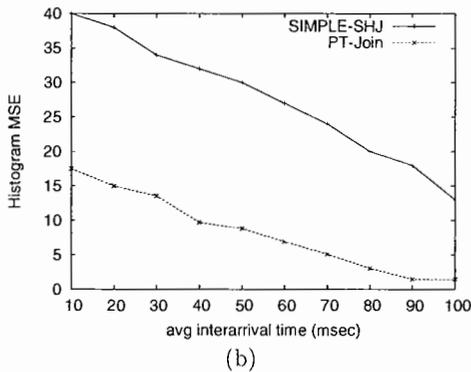
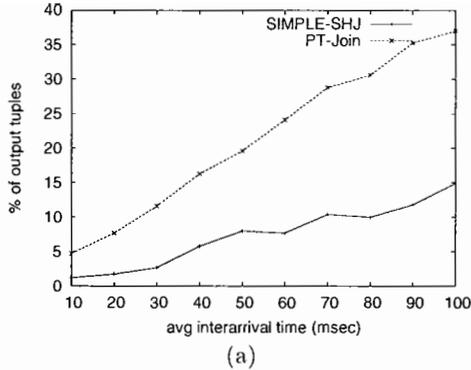


Figure 13: Output faithful representation

mance of two example aggregates under various stream rates. The performance of the *average* and the *sum* operations is illustrated in Figure 16 a, b, respectively. The figure shows that the promising tuple approach exhibits a lower mean square error (MSE) between its answer and the exact answer for both the *average* and the *sum* operations. Notice that as we increase the average interarrival time between consecutive tuples, the MSE decreases because the system gets less loaded and more input tuples are processed by the system.

## 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 Niagra project [14] adopts the notion of continuous queries and how group optimizations can

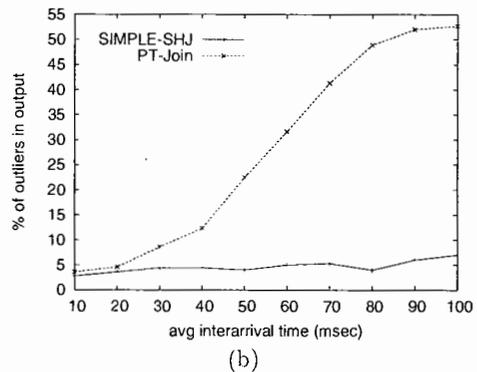
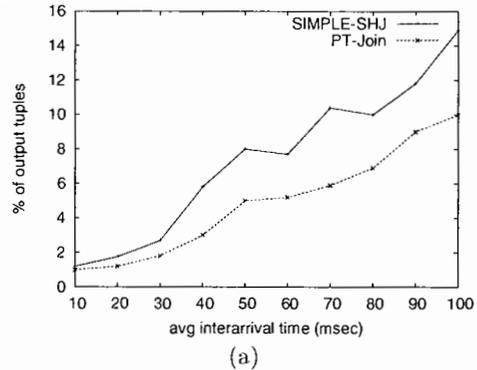


Figure 14: Outlier detection

be performed over these 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 one 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. Consequently,

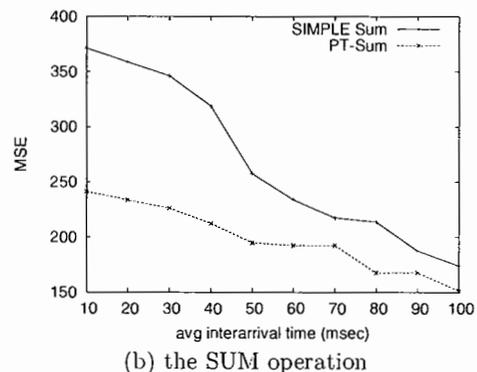
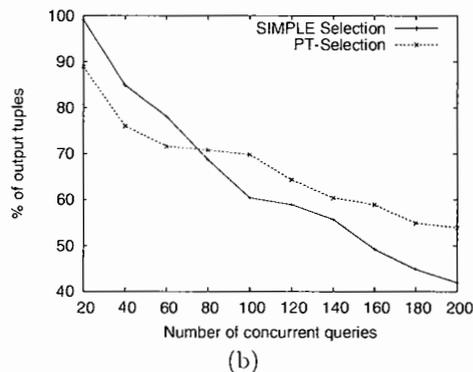
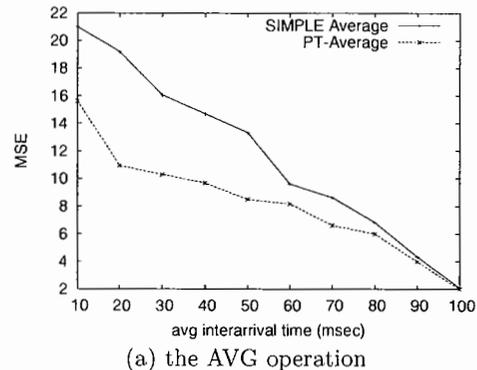
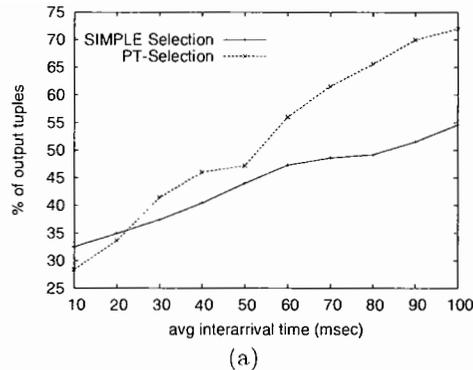


Figure 15: Performance of the selection operation

Figure 16: Performance of aggregates

one may decide to query the sensor for a fresh reading.

From the query side, various stream operators have been studied in the literature. For example, the window join operation is addressed in [16, 20, 22, 23, 25, 30]. Maintaining statistics over sliding windows is investigated in [17]. Incremental maintenance of temporal aggregates over windows is presented in [34]. There has been some ongoing research on random sampling for some special operations, e.g., the join operation [3, 13] and the group-by operation [2]. The work in [6] presents some optimizations for conjunctive queries over sliding windows.

## 7 Conclusions

In this paper, we introduced the notion of promising tuples. 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, i.e., 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

- [1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 2:120–139, August 2003.
- [2] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *Proceedings of ACM SIGMOD*, pages 487–498, May 2000.
- [3] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *Proceedings of ACM SIGMOD*, pages 275–286, June 1999.
- [4] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In

- Proceedings of the annual ACM symp. on Theory of computing*, pages 20–29, May 1996.
- [5] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. *TODS*, 29(1):162–194, March 2004.
- [6] A. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *Proceedings of ACM SIGMOD*, pages 419–430, June 2004.
- [7] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *Proceedings of the Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 633–634, Jan. 2002.
- [8] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of PODS*, pages 1–16, June 2002.
- [9] P. Bonnet, J. E. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proceedings of the Intl. Conf. on Mobile Data Management*, pages 3–14, Jan. 2001.
- [10] K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. In *Proceedings of VLDB*, pages 111–122, Sept. 2000.
- [11] S. Chandrasekaran, O. Cooper, A. Deshpande, and et al. Telegraphicq: Continuous dataflow processing for an uncertain world. In *Proceedings of CIDR*, Jan. 2003.
- [12] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proceedings of the Intl. Colloquium on Automata, Languages and Programming*, pages 693–703, July 2002.
- [13] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *Proceedings of ACM SIGMOD*, pages 263–274, June 1999.
- [14] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaraq: A scalable continuous query system for internet databases. In *Proceedings of ACM SIGMOD*, pages 379–390, May 2000.
- [15] C. D. Cranor, T. Johnson, O. Spatscheck, and et al. Gigascope: A stream database for network applications. In *Proceedings of ACM SIGMOD*, pages 647–651, June 2003.
- [16] A. Das, J. Gehrke, and M. Riedewald. Semantic approximation of data stream joins. *IEEE Trans. Knowl. Data Eng.*, 17(1):44–59, 2005.
- [17] M. Datar, A. Gionis, P. Indyk, and et al. Maintaining stream statistics over sliding windows. In *of the Thirteenth Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 635–644, Jan. 2002.
- [18] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *Proceedings of VLDB*, pages 588–599, August 2004.
- [19] A. Dobra, M. N. Garofalakis, J. Gehrke, and R. Rastogi. Sketch-based multi-query processing over data streams. In *Proceedings of EDBT*, pages 551–568, March 2004.
- [20] L. Golab and M. T. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proceedings of VLDB*, pages 500–511, Sept. 2003.
- [21] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proceedings of the annual ACM symp. on Theory of computing*, pages 471–475, July 2001.
- [22] M. A. Hammad, W. G. Aref, and A. K. Elmagarmid. Stream window join: Tracking moving objects in sensor-network databases. In *Proceedings of SSDBM*, pages 75–84, July 2003.
- [23] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *Proceedings of VLDB*, pages 297–308, Sept. 2003.
- [24] M. A. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Eltabakh, M. G. Elfeky, T. Ghanem, R. G. andlhab F. Ilyas, M. Marzouk, and X. Xiong. Nile: A query processing engine for data streams. In *Proceedings of ICDE*, page 851, April 2004.
- [25] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *Proceedings of ICDE*, pages 341–352, March 2003.
- [26] D. Kifer, S. Ben-David, and J. Gehrke. Detecting change in data streams. In *Proceedings of VLDB*, pages 180–191, 2004.
- [27] S. Madden and M. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of ICDE*, pages 555–566, Feb. 2002.
- [28] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of CIDR*, Jan. 2003.
- [29] T. Palpanas, M. Vlachos, E. J. Keogh, D. Gunopulos, and W. Truppel. Online amnesic approximation of streaming time series. In *Proceedings of ICDE*, pages 338–349, April 2004.
- [30] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *Proceedings of VLDB*, pages 324–335, 2004.
- [31] S. Tanimoto and T. Pavlidis. A hierarchical data structure for picture processing. *Computer Graphics and Image Processing*, 4(2):104–119, June 1975.
- [32] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proceedings of ACM SIGMOD*, pages 193–204, June 1999.
- [33] A. N. Wilschut and E. M. G. Apers. Pipelining in query execution. In *Proceedings of the International Conference on Databases, Parallel Architectures and their Applications*, 1991.
- [34] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In *Proceedings of ICDE*, pages 51–60, April 2001.
- [35] D. Zhang, D. Gunopulos, V. J. Tsotras, and B. Seeger. Temporal aggregation over data streams using multiple granularities. In *In Proceedings of EDBT*, pages 646–663, March 2002.