Chapter 5

FILTERING, PUNCTUATION, WINDOWS AND SYNOPSES

David Maier[1], Peter A. Tucker[2] and Minos Garofalakis[3]
[1]OGI School of Science & Engineering at OHSU, 20000 NW Walker Road, Beaverton, OR 97006;[2]Whitworth College, 300 W Hawthorne Road, Spokane, WA 99251; [3]Bell Laboratories, Lucent Technologies, Murray Hill, NJ 07974

Abstract: This chapter addresses some of the problems raised by the high-volume, non-terminating nature of many data streams. We begin by outlining challenges for query processing over such streams, such as outstripping CPU or memory resources, operators that wait for the end of input and unbounded query state. We then consider various techniques for meeting those challenges. *Filtering* attempts to reduce stream volume in order to save on system resources. *Punctuations* incorporate semantics on the structure of a stream into the stream itself, and can help unblock query operators and reduce the state they must retain. *Windowing* modifies a query so that processing takes place on finite subsets of full streams. *Synopses* are compact, efficiently maintained summaries of data that can provide approximate answers to particular queries.

Key words: data stream processing, disordered data, stream filtering, stream punctuation, stream synopses, window queries

1. INTRODUCTION: CHALLENGES FOR PROCESSING DATA STREAMS

The kinds of manipulations users would like to perform on data streams are reminiscent of operations from database query processing, OLAP and data mining: selections, aggregations, pattern-finding. Thus, one might hope that data structures and algorithms developed for those areas could be carried over for use in data stream processing systems. However, existing approaches may be inadequate when confronted with the high-volume and unbounded nature of some data streams, along with the desire for near-real time results for stream operations.

The data rate for a stream might outstrip processing resources on a steady or intermittent (bursty) basis. Thus extensive CPU processing or secondary storage access for stream elements may be infeasible, at least for periods of time. Nor can one rely on buffering extensive amounts of the stream input in memory. For some applications, such as network monitoring, a few seconds of input may exhaust main memory. Furthermore, while buffering might handle bursty input, it does so at a cost of delaying results to users.

The potentially unbounded nature of data streams also creates problems for existing database query operators, or for particular implementations of them. Blocking operators, such as group-by, difference and sort, and blocking implementations, such as most join algorithms, cannot in general emit any output until the end of one or more of the inputs is reached. Thus on a continuous data stream, they will never produce any output. In the case of join, there are alternative implementations, such as symmetric hash join (Wilschut and Apers, 1991) that are non-blocking, and hence more suitable for use with streams. But the other operators mentioned are inherently blocking for any implementation. Even if an operator has a non-blocking implementation, if it is stateful, such as join and duplicate elimination, it will accumulate state without limit, eventually becoming memory bound.

Thus, extensions or alternatives for current query processing and data analysis techniques are needed for streams. In this chapter, we survey several approaches to these challenges, based on data reduction, exploiting semantics of data streams, and approximation. We first cover exact and lossy filtering techniques, which attempt to reduce data stream volumes early in the processing chain, in order to reduce the computational demands on later operations. We then consider the use of stream "punctuation" to incorporate knowledge about the internal structure in a data stream that might be useful in unblocking operators or limiting the amount of state that must be retained. We then consider "windowed" versions of classical operators, which can be viewed as a continuous user query being approximated by a series of queries over finite subsequences of an unbounded stream. In this context we also briefly consider issues with disordered inputs. The final class of techniques we cover are synopses, which in the stream case can be considered as representations of data streams that a) summarize the stream input, b) can be maintained online at stream input rates, c) occupy much less space the full data, and d) can be used to provide exact or approximate answers to some class of user queries.

2. STREAM FILTERING: VOLUME REDUCTION

Faced with stream volumes beyond what available resources allow processing in their entirety, a stream processor can simply abort, or somehow reduce the volume to a manageable level. Such reduction can take several forms: precise filtering, data merging, or data dropping.

2.1 Precise Filtering

Precise filtering extracts some portion of a stream query for application nearer the stream source, with the expectation of reducing stream volume while not changing the final query answer. Filtering operations generally need to be simple, such as selection or projection, and applicable on an item-by-item basis, so as not to consume extensive processing cycles or memory. Filtering should also avoid introducing long delays into time-critical data streams. This filtering may happen at the stream source, near the stream-processing system, or in between.

A source may support subscription to a substream of the full data stream. For example, the Virtual Object Ring Buffer (VORB) facility of the RoadNet project (Rajasekar et al., 2004) supports access to real-time information from an environmental-sensing network. A VORB client can request a substream of this information restricted on (geographic) space, time and attribute. Financial feeds also support filtering, such as on specific stocks or currencies.

Hillston and Kloul (2001) describe an architecture for an online auction system where active network nodes serve as filters on the bid stream. Such a node can filter out any bid for which a higher bid has already been handled for the same item. (It is also possible that highest bid information is periodically disseminated from the central auction server to the active network nodes, as otherwise an active node is only aware of bid values that it handles.) Such processing is more complex than item-at-a-time filtering. It essentially requires an anti-semijoin of incoming bids with a cache of previous bids. However, the space required can be reduced from what is required for a general semijoin by two considerations. First, only one record needs to be retained for each auction item – the one with the maximum price so far. (Actually, just the item ID and bid price suffice.) Second, the cache of previous bids does not need to be complete – failure to store a previous bid for an item only means that later items with lower prices are not filtered at the node. Thus an active node can devote a bounded cache to past information, and select bids to keep in the cache based on recency or frequency of activity on an item.

Gigascope (Johnson et al., 2003) is a stream-processing system targeted at network monitoring and analysis. It supports factoring of query conditions that can be applied to the raw data stream arriving at the processor. These conditions can be applied in the network interface subsystem. In some versions, these filter conditions are actually pushed down into a programmable network interface card (NIC).

2.2 Data Merging

Data merging seeks to condense several data items into one in such a way that the ultimate query can still be evaluated. Consider a query that is computing the top-5 most active network flows in terms of bytes sent. (Here a flow is defined by a source and destination IP address and port number.) Byte-count information for packets from the same flow can be combined and periodically transferred to the stream-processing system. This approach is essentially what routers do in generating Netflow records (Cisco Systems, 2001), reducing the volume of data that a network monitoring or profiling application needs to deal with. Of course, only certain queries on the underlying network traffic will be expressible over the aggregated Netflow records. A query looking for the most active connections is expressible, but not an intrusion-detection query seeking a particular packet signature. Merging can be viewed as a special case of synopsis. (See Section 6.)

2.3 Data Dropping

Data dropping (also called load shedding) copes with high data rates by discarding data items from the processing stream, or limiting the processing of selected items. Naïve dropping happens in an uncontrolled manner – for example, items are evicted without processing from an overflowed buffer. More sophisticated dropping schemes introduce some criterion that identifies which data items to remove, based for example, on the effect upon the accuracy of the answer or an attempt to get a fair sample of a data stream.

The simplest approaches can be termed *blind* dropping: the decision to discard a data item is made without reference to its contents. In the crudest form, blind dropping discards items when CPU or memory limits are exceeded: Data items are dropped until the stream-processing system catches up. Such a policy can be detrimental to answer quality, with long stretches of the input being unrepresented. Better approaches attempt to anticipate overload and spread out the dropped data items, either randomly or uniformly. For example a VORB client can throttle the data flow from a data source, requesting a particular rate for data items, such as 20 per minute.

Dropping can take place at the stream source, at the leaves of a stream query, or somewhere in the middle of a query plan. The Aurora data stream manager provides an explicit **`drop`** operator that may be inserted at one or more places in a network of query operators (Tatbul et al., 2003). The **`drop`** can eliminate items randomly or based on a predicate (which is termed *semantic* dropping). Another approach to intra-query dropping is the modification of particular operators. Das et al. (2003) and Kang and Naughton (2003) present versions of window join (see Section 4) that shed load by either dropping items or avoiding the join of particular items.

Whatever mechanism is used for dropping data items, key issues are determining how much to drop and maximizing answer quality for a given drop rate. The Aurora system considers essentially all placements of **`drop`** operators in an operator network (guided by heuristics) and precomputes a sequence of alternative plans that save progressively more processing cycles, called a *load-shedding road map* (LSRM). For a particular level of cycle savings, Aurora selects the plan that maximizes the estimated quality of service (QoS) of the output. QoS specifications are provided by query clients and indicate, for example, how the utility of an answer drops off as the percentage of full output decreases, or which ranges of values are most important. The two window-join algorithms mentioned above attempt to maximize the percentage of join tuples produced for given resource limits. Das et al. point out that randomized dropping of tuples in a join can be ineffective by this measure. Consider a join between r tuples and s tuples on attribute A. Any resources expended on an r tuple with $r.A = 5$ is wasted if the only s tuple with $s.A = 5$ has been discarded. They instead collect statistics on the distribution of join values, and retain tuples that are likely to contribute to multiple output tuples in the join. Kang and Naughton look at how to maximize output of a window join given limitations on computational or memory resources. They demonstrate, for example, with computational limits, the operator should favor joining in the direction of the smaller window to the larger window. For limited memory, however, it is better to allocate that resource to storing tuples from the slower input.

There are tensions in intelligent data dropping schemes, however. On one hand, one would like to select data items to discard carefully. However, a complicated selection process can mean more time is spent selecting a data item to remove than is saved by removing it. Similarly, the value of a data item in the answer may only be apparent after it passes through some initial operators. For example, it might be compared to frequent data values in a stream with which it is being joined. However, discarding a data item in the middle of a query plan means there are “sunk costs” already incurred that cannot be reclaimed.

2.4 Filtering with Multiple Queries

For any of the filtering approaches – precise filtering, data merging and data dropping – the situation is more complicated in the (likely) scenario that multiple queries are being evaluated over the data streams. Now, the combined needs of all the queries must be met. With precise filtering, for example, the filter condition will need to be the union of the filters for the individual queries, which means the processing of the raw stream may be more complex, and the net reduction in volume smaller. In a semantic data-dropping scheme, there may be conflicts in that the least important data items for one query are the most important for another. (In the multi-query case, Aurora tries to ensure different users receive answers of approximately equal utility according to their QoS specifications.)

3. PUNCTUATIONS: HANDLING UNBOUNDED BEHAVIOR BY EXPLOITING STREAM SEMANTICS

Blocking and stateful query operators create problems for a query engine processing unbounded input. Let us first consider how a traditional DBMS executes a query plan over bounded data. Each query operator in the plan reads from one or more inputs that are directly beneath that operator. When all data has been read from an input, the operator receives an end of file (EOF) message. Occasionally a query operator will have to reread the input when it receives EOF (e.g., a nested-loops join algorithm). If not, the query operator has completed its work. A stateful query operator can purge its state at this point. A blocking operator can output its results. Finally, the operator can send the EOF message to the next operator along in the query plan.

The EOF message tells a query operator that the end of the entire input has arrived. What if a query operator knew instead that the end of a subset of the input data set had arrived? A stateful operator might purge a subset of the state it maintains. A blocking operator might output a subset of its results. An operator might also notify the next operator in the query plan that a subset of results had been output. We will explain how “punctuations” are included in a data stream to convey knowledge about ends of data subsets.

For example, suppose we want to process data from a collection of environmental sensors to determine the maximum temperature each hour using a DBMS. Since data items contain the time they were emitted from the sensor, we can assume that data from each sensor is sorted (non-decreasing) on time. In order to calculate the maximum temperature each hour from a single sensor, we would use the following query (in SQL):

```
SELECT MAX(temp)
FROM sensor
GROUP BY hour;
```

Unfortunately, since group-by is blocking and the input is unbounded, this query never outputs a result. One solution is to recognize that hour is non-decreasing. As data items arrive, the group-by operator can maintain state for the current hour. When a data item arrives for a new hour, the results for the current hour can be output, and the query no longer blocks.

This approach breaks down when the input is not sorted. Even in our simple scenario, data items can arrive out-of-order to the group-by operator for various reasons. We will discuss disorder in data streams in Section 5. By embedding punctuations into the data stream and enhancing query operators to exploit punctuations, the example query will output results before receiving an EOF, even if data arrive out-of-order.

3.1 Punctuated Data Streams

A *punctuation* is an item embedded into a data stream that denotes the end of some subset of data (Tucker et al., 2003). At a high level, a punctuation can be seen as a predicate over the data domain, where data items that pass the predicate are said to *match* the punctuation. In a *punctuated stream*, any data item that matches a punctuation will arrive before that punctuation. Given a data item *d* and a punctuation *p*, we will use *match(d,p)* as the function that indicates whether a *d* matches *p*.

The behaviors exhibited by a query operator when the EOF message is received may also be partially performed when a punctuation is received. Clearly, EOF will not arrive from unbounded inputs, but punctuations break up the unbounded input into bounded substreams. We define three kinds of behaviors, called *punctuation behaviors*, to describe how operators can take advantage of punctuations that have arrived. First, *pass behavior* defines when a blocking operator can output results. Second, *keep behavior* defines when a stateful operator can release some of its state. Finally, *propagate behavior* defines when an operator can output punctuations.

In the environmental sensor example, data output from each sensor are sorted on time. We can embed punctuations into the stream at regular intervals specifying that all data items for a particular prefix of the sorted stream have arrived. For example, we can embed punctuations at the end of each hour. This approach has two advantages: First, we do not have to enhance query operators to expect sorted input (though we do have to enhance query operators to support punctuations). Second, query operators do not have to maintain sorted output.

3.2 Exploiting Punctuations

Punctuation behaviors exist for many query operators. Non-trivial behaviors are listed in Tables 5-1, 5-2, and 5-3. The pass behavior for group-by says that results for a group can be output when punctuations have arrived that match all possible data items that could participate in that group. The keep behavior for group-by says that state for a group can be released in similar circumstances. Finally, the propagate behavior for group-by says that punctuations that match all possible data items for a group can be emitted (after all results for that group have been output). For example, when group-by receives the punctuation marking the end of a particular hour, the results for that hour may be output, state required for that hour can be released, and a punctuation for all data items with that hour can be emitted. Notice that ordering of data items on the hour attribute does not matter. Even if data arrives out of order, as long as the punctuation correctly denotes the end of each hour, the results will still be accurate.

Many query operators require specific kinds of punctuations. We saw above that the pass behavior for group-by was to output a group when punctuations had arrived that matched all possible data items that can participate in that group. A set of punctuations *P describes* a set of attributes *A* if, given specific values for *A*, every possible data item with those attribute values for *A* matches some punctuation in *P*. For example, punctuations from the environment sensors that denote the end of a particular hour describe the hour attribute, since they match all possible data items for a particular hour.

Table 5-1. Non-trivial pass behaviors for blocking operators, based on punctuations that have arrived from the input(s).

Group-by	Groups that match punctuations that describe the grouping attributes.
Sort	Data items that match punctuations that have arrived covering all possible data items in a prefix of the sorted output.
Difference (S_1-S_2)	Data items in S_1 that are not in S_2 and match punctuations from S_2.

Table 5-2. Non-trivial propagation behaviors for query operators, based on punctuations that have arrived from the input(s).

Select	All punctuations.
Dupelim	All punctuations.
Project$_A$	The projection of *A* on punctuations that describe the projection attributes.
Group-by	Punctuations that describe the group-by attributes.
Sort	Punctuations that match all data in a prefix of the sorted output.
Join	The result of joining punctuations that describe the join attributes.
Union	Punctuations that equal some punctuation from each other inputs.
Intersect	Punctuations that equal some punctuation from each other inputs.
Difference	Punctuations that equal some punctuation from each other inputs.

Table 5-3. Non-trivial keep behaviors for stateful query operators, based on punctuations that have arrived from the input(s).

Dupelim	Data items that do not match any punctuations received so far.
Group-by	Data items that do not match punctuations describing the grouping attributes.
Sort	Data items that do not match any punctuations covering all data items in the prefix of the sorted output defined in the pass behavior.
Join	Data items that do not match any punctuations from the other input that describe the join attributes.
Intersect	Data items that do not match any punctuations from the other input.
Difference	Data items that do not match any punctuations from the other input.

3.3 Using Punctuations in the Example Query

Suppose in the environmental sensor example each sensor unit outputs data items that contain: sensor id, temperature, hour, and minute. Thus an example stream from sensor 3 might contain: [<3,75,1,15>, <3,78,1,30>, <3,75,1,45>, <3,76,2,0>, <3,75,2,15>, …]. We would like to have the sensors emit punctuations that denoted the end of each hour, to unblock the group-by operator. We treat punctuations as stream items, where punctuations have the same schema as the data items they are matching and each attribute contains a pattern. Table 5-4 lists the patterns an attribute in a punctuation can take.

Table 5-4. Punctuation patterns

Pattern	Representation	Match Rule
wildcard	*	All values.
constant	*c*	The value *c*.
list	$\{c_1, c_2, \ldots\}$	Any value c_i in the list.
range	(c_1, c_2)	Values greater than c_1 and less than c_2.

We want punctuations embedded into the data stream denoting the end of data items for a specific hour. One possible instantiation of such a stream might be (where the punctuation is prefixed with *P*): [<3,75,1,15>, <3,78,1,30>, <3,75,1,45>, <3,76,2,0>, *P*<*,*,1,*>, <3,75,2,15>]. All data items containing the value 1 for hour match the punctuation.

How will punctuations that mark the end of each hour help our example query, where we take input from many sensors? We examine each operator in turn. Suppose our query plan is as in Figure 5-1, and each sensor emits punctuations at the end of an hour. As data items arrive at the union operator, they are immediately output to the group-by operator. Note that union does not attempt to enforce order. Due to the propagation invariant for union, however, punctuations are not immediately output as they arrive. Instead, union stores punctuations in its state until all inputs have produced equal punctuations. At that point, a punctuation is output denoting the end of data items for that hour.

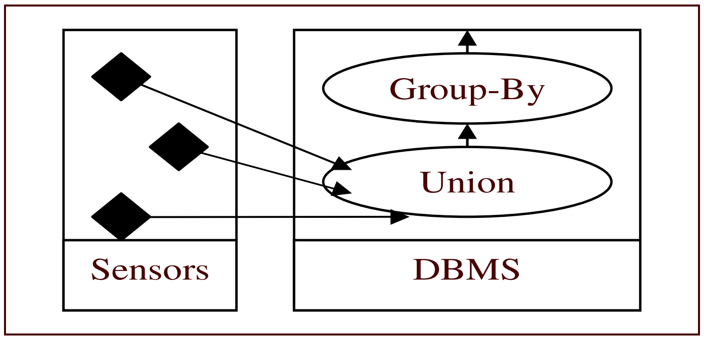

Figure 5-1. Possible query tree for the environment sensor query.

When a data item arrives at group-by, the appropriate group is updated, in this case, the maximum temperature for a specific hour. When a punctuation denoting the end of an hour arrives, group-by can output results for that hour, clear out its state for that hour, and emit a new punctuation denoting the end of data items for that hour. Thus, the query is unblocked, and the amount of state required has been reduced, making it more appropriate for unbounded data streams.

3.4 Sources of Punctuations

We have seen how punctuated streams help query operators. However, we have not explained how punctuations get into a data stream. We posit a logical operator that embeds punctuations and can occur in various places: at the stream source, at the edge of the query processor, or after query operators within the query. We call this operator the *insert punctuation* operator. There are many different schemes for implementing the insert punctuation operator. Which scheme to choose depends on where the information resides for generating punctuation. We list some alternatives below:

- **Source or sensor intelligence**: The stream source may know enough to emit a punctuation. For example, the individual environmental sensors produced data sorted on time. When an hour ended, the sensor emitted punctuation that all reports for that hour had been output.
- **Knowledge of access order**: Scan or fetch operations may know something about the source, and generate punctuations based on that knowledge. For example, if scan is able to use an index to read a source,

it may use information from that index to tell when all values for an attribute have been read.

- **Knowledge of stream or application semantics**: An insert punctuation operator may know something about the semantics of its source. In the environmental example, temperature sensors might have temperature limits, say -20F and 125F. An insert punctuation operator can output two punctuations immediately: One that says there will not be any temperature reports below -20F and another that says there will not be any reports above 125F.
- **Auxiliary information**: Punctuation may be generated from sources other than the input stream, such as relational tables or other files. In the environmental example, we might have a list of all the sensor units. An insert punctuation operator could use that to determine when all sensors output results for a particular hour, and embed the punctuation itself. This approach can remove punctuation logic from the sensors.
- **Operator semantics**: Some query operators impose semantics on output data items. For example, the sort operator can embed punctuations based on its sort order. When it emits a data item, it can follow that data item with a punctuation stating that no more data items will appear that precede that item in order.

3.5 Open Issues

We have seen that punctuations can improve the behavior of individual query operators for processing unbounded data streams. One issue not addressed yet is how to determine if punctuations can improve the behavior of entire queries. There are two questions here: First, what kinds of queries can be helped by punctuations? Not all queries can be improved by punctuations; we would like to be able to characterize those that can. The second question is, given a query (that we believe can be improved by punctuations), what kinds of punctuations will help that query? We refer to the set of punctuations that will be emitted from a stream source as the *punctuation scheme* of that source. In the sensor query, a punctuations scheme that describes the hour attribute helps the query, but so do schemes that punctuate every 20 minutes, or at the end of every second hour.

A related question is, of the kinds of punctuation schemes that will improve the behavior of a query, which are most efficient? Again referring to the environmental query, if punctuations are emitted at the end of each hour, memory usage is minimized since state is purged as soon as possible. However, this choice maximizes the number of punctuations in the stream. If instead punctuations are embedded every six hours, then memory usage is increased but the number of punctuations in the stream is reduced and the processing time for them is reduced.

One final issue relates to query optimization. Given a logical query, do two (or more) equivalent query plans exist that exhibit different behaviors based on the same input punctuation scheme? For example, if one query plan is unblocked by the scheme and another is not, then choosing the unblocked query plan is most logical. Optimizing for state size is more difficult, since punctuation schemes do not give guarantees on when a particular punctuation will arrive. However, it would be useful for a query optimizer to choose the query plan with the smallest predicted requirement for memory.

3.6 Summary

Punctuations are useful for improving the behavior of queries over unbounded data streams, even when the input arrives out-of-order. Query operators act on punctuations based on three kinds of behaviors: Pass behavior defines when a blocking operator can output results. Keep behavior defines what state must be kept by a stateful operator. Propagation behavior defines when an operator can emit punctuation.

4. WINDOWS: HANDLING UNBOUNDED BEHAVIOR BY MODIFYING QUERIES

Windowing operates on the level of either a whole query or an individual operator, by changing the semantics from computing one answer over an entire (potentially unbounded) input streams to repeated computations on finite subsets (windows) of one or more streams. Two examples:

1. Consider computing the maximum over a stream of temperature readings. Clearly, this query cannot emit output while data items are still arriving. A windowed version of this query might, for example, compute the maximum over successive 3-minute intervals, emitting an output for each 3-minute window.
2. Consider a query that matches packet information from two different network routers. Retaining all items from both sources in order to perform a join between them will quickly exhaust the storage of most computing systems. A windowed version of this query might restrict the matching to packets that have arrived in the last 15 seconds. Thus, any packet over 15 seconds old can be discarded, once it has been compared to the appropriate packets from the other input.

There are several benefits from modifying a query with windows.

- An operation, such as aggregation, that would normally be blocking can emit output even while input continues to arrive.
- A query can reduce the state it must retain to process the input streams.
- Windowing can also reduce computational demands, by limiting the amount of data an operation such as join must examine at each iteration.

There have been many different ways of defining windows proposed. The size of a window can be defined in terms of the number of items or by an interval based on an attribute in the items, such as a timestamp. The relationship between successive window instances can vary. In a *tumbling* window (Carney et al., 2002), successive window instances are disjoint, while in a *sliding* window the instances overlap. Window instances may have the same or different sizes. For example, in a *landmark* window (Gehrke et al., 2001), successive instances share the same beginning point (the landmark), but have successively later endpoints.

5. DEALING WITH DISORDER

Stream query approaches such as windowing often require that data arrive in some order. For example, consider the example from Section 3, where we want the maximum temperature value from a group of sensors each hour. This query can be modified to a window query that reports the maximum temperature data items in each hour interval is output, as follows (using syntax similar to CQL (Arasu et al., 2003)):

```
SELECT MAX(temp)
FROM sensor [RANGE 60 MINUTES];
```

In a simple implementation, when a data item arrives that belongs to a new window, the results for the current window is “closed”, its maximum is output, and state for a new window is initialized. However, such an implementation assumes that data arrive in sorted order. Suppose the data items do not quite arrive in order. How can we accurately determine if a window is closed?

5.1 Sources of Disorder

A data stream is in *disorder* when it has some expected arrival order, but its actual arrival order does not follow the expected arrival order exactly. It may be nearly ordered, but with a few exceptions. For example, the following list of integers is in disorder: [1,2,3,5,4,6,7,9,10,8]. Clearly the list is close to being in order, and can be put back in order with buffering.

Disorder can arise in a data stream for several reasons: Data items may take different routes, with different delays, from their source; the stream might be a combination of many sources with different delays; the ordering attribute of interest (e.g., event start time) may differ from the order in which items are produced (e.g., event end time). Further, an operator in a stream processing system may not maintain sort order in its output, even if the data items arrive in order. For a simple example, consider the union operator. Unless it is implemented to maintain sorted order, its output will not necessarily be ordered.

5.2 Handling Disorder

A query operator requiring ordered data can be modified to handle data streams in disorder. First, it must know the *degree of disorder* in the stream: how far away from sorted order each data item in the stream can be. There are two approaches we discuss: *global disorder properties* and *local disorder properties*. Once the operator can determine the degree of disorder, it has a least two choices on how to proceed. It can put its input into sorted order, or it can process the input out of order.

5.2.1 Expressing the Degree of Disorder in a Data Stream

The degree of disorder can be expressed using global or local stream constraints. A global disorder property is one that holds for the entire stream. Several systems use this approach. In Gigascope (Johnson et al., 2003), the degree of disorder can be expressed in terms of the position of a data item in the stream, or in terms of the value of the sorting attribute in a data item. A stream is *increasing within δ* if, for a data item *t* in stream *S*, no data item arrived δ items before *t* on *S* that precede *t* in the sort order. Thus, disorder is expressed in terms of a data item's position in the stream. Similarly, a stream is *banded-increasing (ε)* for an attribute *A* if, for a data item *t* in stream *S*, no data item precedes *t* in *S* with a value for *A* greater than $t.A + \varepsilon$..

Related to these notions from Gigascope are *slack* in Aurora (Carney et al., 2002) and *k-constraints* in STREAM (Babu et al., 2004). In Aurora, an operator that requires sorted input is given an ordering specification, which contains the attribute on which the order is defined and a slack parameter. The slack parameter specifies how out of order a data item might arrive, in terms of position. In STREAM, a *k*-constraint specifies how strictly an input adheres to some constraint. One kind of k-constraint is *k-ordering*, where *k* specifies that out-of-order items are at most *k* positions away from being in order. Note that $k = 0$ implies sorted input.

There are two advantages to using a global disorder property approach. First, it is relatively simple to understand in that it is generally expressed with a single integer. Second, it generally gives a bound on the amount of state required during execution and the amount of latency to expect in the output. However, global disorder properties also have disadvantages. First, it is not always clear what the constraint should be for non-leaf query operators in a query plan. For example, suppose a query has a windowed aggregate operator above the union of five inputs. We may know the degree of disorder of each input to the union, but what is the degree of disorder for the output of union? A second disadvantage is that it is generally not flexible. A bursty stream will likely have a higher degree of disorder during bursts and a lower degree during lulls. If we want accurate results, we must set global disorder constraint to the worst-case scenario, increasing the latency at other times.

A second way to express the degree of disorder is through local disorder properties (Tucker and Maier, 2003). In this method, we are able to determine through properties of the stream the degree of disorder during execution. One method to determining local disorder is to use punctuations. Appropriate punctuation on an ordering attribute can be used, for example, to close a window for a windowed operator. Punctuations are propagated to other operators higher up in the query plan. Thus, there is not the problem of how disorder in lower query operators translates to disorder in operators further along in a query tree. In STREAM, the *k* value for a *k*-constraint can dynamically change based on data input, similar to a local disorder property. A monitoring process checks the input data items as they arrive, and tries to detect when the *k* value for useful constraints changes during execution.

The main advantage of using a local disorder property approach is its flexibility. The local disorder property approach can adapt to changes in the stream, such as bursts and lulls. However, since the degree of disorder may not remain static throughout execution, we cannot determine a bound for the state requirement as we can with global disorder properties.

5.2.2 Processing Disordered Data Streams

Once an operator knows the degree of disorder in its input, it can begin processing data from the input stream. One approach in handling disorder is to reorder the data as they arrive in the leaf operators of the query, and use order-preserving operators throughout the query. In Aurora, disordered data streams are ordered using the *BSort* operator. BSort performs a buffer-based sort given an ordering specification. Suppose *n* is the slack in the ordering specification. Then the BSort operator sets up a buffer of size *n+1*, and as data items arrive they are inserted into the buffer. When the buffer fills, the minimum data item in the buffer according to the sort order is evicted. Note that if data items arrive outside the slack parameter value, they are still placed in the buffer and output as usual. Thus, the BSort operator is only an approximate sort, and its output may still be in disorder.

As data are sorted (at least approximately), later operators should preserve order. Some operators, such as select and project, already maintain the input order. It is a more difficult task for other operators. Consider an order-preserving version of union, and suppose it is reading from two inputs already in order. Union outputs the minimum data item, according to the sort order, from the two inputs. This implementation is simple for reliable inputs, but data streams are not always reliable. Suppose one of the inputs to union stalls. The union operator cannot output data items that arrive on the other input until the stalled input resumes. Maintaining order in other operators, such as join, is also non-trivial.

Instead of forcing operators to maintain order, an alternative is for data to remain disordered, and process each data item as it arrives. Many operators (again select and project are good examples) do not require data to arrive in order. However, operators that require some sort of ordered input must still determine the degree if disorder in the input. If we use one of the global disorder property approaches, then we must estimate the degree of disorder of the output based on the global disorder properties of the input. However, if we use punctuations, then disorder information is carried through the stream automatically using each operator's propagation behaviors.

5.3 Summary

Many operators, such as window operators, are sensitive to window order. However, as streams are not always reliable data sources, disorder may arise. To handle disorder, an operator must first determine the degree of disorder in its inputs. Once the degree of disorder is determined, then the operator can either resort the data process the data out-of-order. We have presented different ways to express disorder in a stream, and the advantages and disadvantages of sorting data compared to processing data out-of-order.

6. SYNOPSES: PROCESSING WITH BOUNDED MEMORY

Two key parameters for processing user queries over continuous, potentially unbounded data-streams are (1) the amount of *memory* made available to the on-line algorithm, and (2) the *per-item processing time* required by the query processor. Memory, in particular, constitutes an important design constraint since, in a typical streaming environment, only limited memory resources are available to the data-stream processing algorithms. In such scenarios, we need algorithms that can summarize the underlying streams in concise, but reasonably accurate, *synopses* that can be stored in the allotted amount of memory and can be used to provide *approximate answers* to user queries along with some reasonable guarantees on the quality of the approximation. Such approximate, on-line query answers are particularly well suited to the exploratory nature of most data-stream processing applications such as, e.g., trend analysis and fraud or anomaly detection in telecom-network data, where the goal is to identify generic, interesting or "out-of-the-ordinary" patterns rather than provide results that are exact to the last decimal.

In this section, we briefly discuss two broad classes of data-stream synopses and their applications. The first class of synopses, termed *AMS sketches*, was originally introduced in an influential paper by Alon, Matias, and Szegedy (1996) and relies on taking random linear projections of a streaming frequency vector. The second class of synopses, termed *FM sketches*, was pioneered by Flajolet and Martin (1985) and employs hashing to randomize incoming stream values over a small (i.e., logarithmic-size) array of hash buckets. Both AMS and FM sketches are small-footprint, randomized data structures that can be easily maintained on-line over rapid-rate data streams; furthermore, they offer tunable, probabilistic accuracy guarantees for estimating several useful classes of aggregate user queries. In a nutshell, AMS sketches can effectively handle important aggregate queries that rely on *bag semantics* for the underlying streams (such as frequency-moment or join-size estimation), whereas FM sketches are useful for aggregate stream queries with *set semantics* (such as estimating the number of distinct values in a stream). Before describing the two classes of sketches in more detail, we first discuss the key elements of a stream-processing architecture based on data synopses.

6.1 Data-Stream Processing Model

Our generic data-stream processing architecture is depicted in Figure 5-2. In contrast to conventional DBMS query processors, our query-processing engine is allowed to see the data tuples in relations $R_1,...,R_r$ *only once* and in the fixed order of their arrival as they stream in from their respective source(s). Backtracking over a stream and explicit access to past tuples is impossible; furthermore, the order of tuples arrival for each streaming relation R_i is arbitrary and duplicate tuples can occur anywhere over the duration of the stream. (In general, the stream rendering each relation R_i can comprise tuple *deletions* as well as insertions, and the sketching techniques described here can readily handle such *update streams*.)

Consider an aggregate query Q over relations $R_1,...,R_r$ and let N denote an upper bound on the total number of streaming tuples. Our data-stream processing engine is allowed a certain amount of memory, typically significantly smaller than the total size of its inputs. This memory is used to continuously maintain a concise *sketch synopsis* of each stream R_i (Figure 5-2). The key constraints imposed on such synopses are that: (1) they are much smaller than the size of the underlying streams (e.g., their size is logarithmic or poly-logarithmic in *N*); and, (2) they can be easily maintained, during a single pass over the streaming tuples in the (arbitrary) order of their arrival. At any point in time, the approximate query-processing engine can combine the maintained collection of synopses to produce an approximate answer to query *Q*.

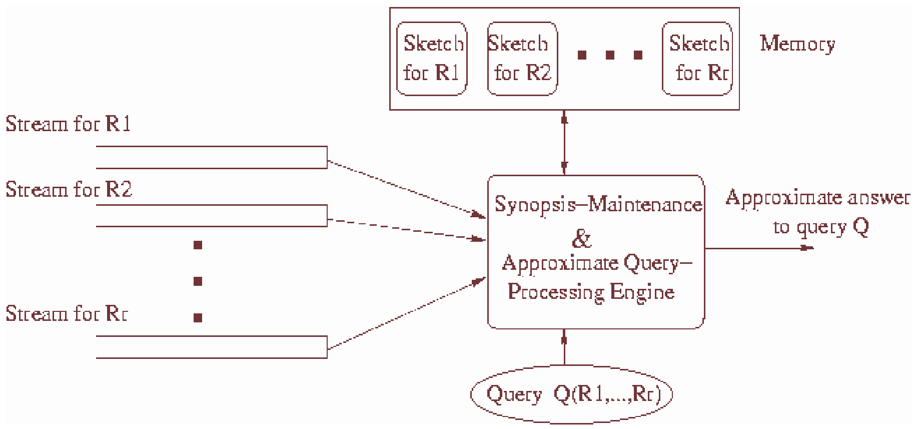

Figure 5-2. Synopsis-based stream query processing architecture.

6.2 Sketching Streams by Random Linear Projections: AMS Sketches

Consider a simple stream-processing scenario where the goal is to estimate the size of a binary equi-join of two streams R_1 and R_2 on join attribute *A*, as the tuples of R_1 and R_2 are streaming in. Without loss of generality, let $[M] = \{0,\ldots,M-1\}$ denote the domain of the join attribute *A*, and let $f_k(i)$ be the frequency of attribute value *i* in R_k. Thus, we want to produce an estimate for the expression $Q = \sum_i f_1(i) \cdot f_2(i)$. Clearly, estimating this join size exactly requires space that is at least linear in *M*, making such an exact solution impractical for a data-stream setting.

In their influential work, Alon et al. (1996, 1999) propose a randomized join-size estimator for streams that can offer strong probabilistic accuracy guarantees while using space that can be significantly sublinear in *M*. The basic idea is to define a random variable *X* that can be easily computed over the streaming values of $R_1.A$ and $R_2.A$ such that: (1) *X* is an *unbiased* (i.e., correct on expectation) estimator for the target join size, so that *E[X] = Q*; and, (2) *X*'s variance can be appropriately upper-bounded to allow for probabilistic guarantees on the quality of the *Q* estimate. This random variable *X* is constructed on-line from the two data streams as follows:

- Select a family of *four-wise independent binary random variables* $\{\xi_i : i = 0,\ldots,M-1\}$, where each ξ_i assumes a value of either +1 or –1, each with probability ½. Informally, the four-wise independence condition means that for any 4-tuple of ξ_i variables and for any 4-tuple of {+1, -1} values, the probability that the values of the variables coincide with those in the {+1, -1} 4-tuple is exactly 1/16 (the product of the equality probabilities for each individual ξ_i). The crucial point here is that, by employing known tools for the explicit construction of small sample spaces supporting four-wise independence, such families can be efficiently constructed on-line using only $O(\log M)$ space.
- Define $X = X_1 \cdot X_2$, where $X_k = \sum_i f_k(i) \cdot \xi_i$, for *k=1,2*. The scalar quantities X_1 and X_2 are called the *atomic AMS sketches* of streams R_1 and R_2, respectively. Each X_k is simply a random linear projection (i.e., an inner product) of the frequency vector of attribute $R_k.A$ with the random vector of ξ_i's that can be efficiently generated from the streaming values of $R_k.A$: Initialize a counter with $X_k = 0$ and simply add ξ_i to X_k whenever value *i* is observed in the $R_k.A$ stream.

Using the four-wise independence property for the ξ_i's, it is easy to verify that the atomic estimate *X* constructed using the process above is an unbiased estimate for *Q* and its variance can be appropriately upper bounded (Alon et al., 1996, 1999). Furthermore, note that, by virtue of linearity, handling deletions in the stream(s) becomes straightforward: To delete an occurrence of value *i*, simply *subtract* ξ_i from the running counter.

As an example, suppose the $R_1.A$ and $R_2.A$ streams comprise, in order, the data values [1, 1, 2, 3, 1, 3] and [3, 1, 3, 1, 1], respectively. Projecting on the family of random variables ξ_i, the atomic sketches of the two streams are $X_1 = \xi_1 + \xi_1 + \xi_2 + \xi_3 + \xi_1 + \xi_3 = 3\xi_1 + \xi_2 + 2\xi_3$ and $X_2 = 3\xi_1 + 2\xi_3$, respectively. Using a specific family of binary random variates, say $\xi = \{\xi_1 = -1, \xi_2 = +1, \xi_3 = -1\}$, we get the atomic AMS sketches X_1 = -3+1-2 = -4 and X_2 = -3-2 = -5, and the atomic estimate X = (-4)(-5) = 20, which approximates the true size of the binary join, i.e., 13.

The approximation guarantees of the randomized AMS join-size estimate can be improved using standard boosting techniques that maintain several independent instantiations of the above-described process, and use averaging and median-selection operators over the atomic *X* estimates to boost accuracy and probabilistic confidence (Alon et al. 1996, 1999). Thus, the AMS sketch for each stream (Figure 5-2) essentially comprises several independent atomic AMS sketch instances (constructed by simply selecting independent random seeds for generating the families of four-wise independent ξ's for each instance).

Extensions of the Basic Method and Applications. The basic ideas of AMS (more generally, random-linear-projection) sketches have found applications in a number of important data-stream processing problems. Dobra et al. (2002, 2004) extend the techniques and results of Alon et al. to handle the estimation of complex, multi-join aggregate queries over streams; they also develop algorithms for effectively processing *multiple* such queries concurrently over a collection of streams by intelligently sharing sketching space and processing. Feigenbaum et al. (1999) and Indyk (2000) use random linear projections to accurately estimate L_p norms over vectors rendered as streams of item arrivals. AMS sketches are also employed by Charikar et al. (2002) to efficiently process top-k queries over a stream of items, and Gilbert et al. (2001, 2002) to build approximate histograms and wavelet decompositions over streams. Recent work has also demonstrated the utility of AMS sketching in dealing with more complex stream-processing scenarios, such as approximating queries with spatial predicates (e.g., overlap joins) over streams of multi-dimensional spatial data (Das et al., 2004), or estimating tree-edit-distance similarity joins over streaming XML documents (Garofalakis and Kumar, 2003).

6.3 Sketching Streams by Hashing: FM Sketches

Consider the problem of estimating the number of *distinct* values in a stream of arriving attribute values *R.A*, where the domain of the attribute is again assumed, without loss of generality, to be $[M] = \{0, \ldots, M-1\}$. (Here, *R* can denote the union of any subset of the R_i streams in Figure 5-2.) As a simple example, for the stream [1, 3, 1, 3, 5, 3, 7] the exact number of distinct values is 3; note that, unlike joins, this query has *set semantics* (i.e., the multiplicity of values appearing in the stream is unimportant). Once again, this estimation problem can be solved exactly in space that is linear in *M*, which could be impractical in a data-stream setting.

To build a small-space estimate for the number of distinct values in a stream, Flajolet and Martin (1985) employ a combination of: (1) a hash function *h()* that maps incoming data values uniformly and independently over the collection of binary strings in the input data domain [*M*]; and, (2) the *lsb()* operator that returns the position of the least-significant 1-bit in its input binary string. The basic idea in their scheme is to map each incoming data value *i* to *lsb(h(i))*. Obviously, $lsb(h(i)) \in \{0, \ldots, \log M - 1\}$ and, furthermore, it is easy to verify that *lsb(h(i))=k* with probability $2^{-(k+1)}$ for each $k = 0,..., \log M - 1$.

An *atomic FM sketch* maintained by the basic Flajolet-Martin scheme is simply a bit-vector of size $O(\log M)$. This bit-vector is initialized to all zeros and, for each incoming stream value *i*, the bit located at position *lsb(h(i))* is switched on. The key observation here is that, by virtue of the exponentially-decaying probabilities for the *lsb(h())* values, we expect a fraction of $2^{-(k+1)}$ of the distinct values in the stream to map to location k in the bit-vector; in other words, if *D* denotes the number of distinct values in the stream, we expect *D/2* values to map to bit 0, *D/4* values to map to bit 1, and so on. Thus, intuitively, at any point in the stream, the location *l* of the leftmost zero in the FM bit-vector sketch provides a good basic estimate of $\log D$, or $2^l \approx D$.

Again, the accuracy and probabilistic confidence of FM-sketching estimates can be boosted using several independent instantiations of the process above (i.e., several atomic FM sketches with independently-chosen hash functions). Detailed analyses and formal results for FM-sketching techniques can be found in (Alon et al., 1996; Flajolet and Martin, 1985; Ganguly et al., 2003). FM sketches can also handle deletions in the stream: The basic idea is to maintain a *counter* (instead of a bit) for each location of the synopsis vector, and simply increment (decrement) the counter at location *lsb(h(i))* for each insertion (respectively, deletion) of value *i*.

Extensions of the Basic Method and Applications. Recent work has extended the ideas of FM (i.e., hashing-based) sketches and explored their use in different data-stream processing domains. Gibbons (2001) employs the idea of hashing into buckets with exponentially decaying probabilities to obtain a *distinct sample* summary for estimating SQL aggregates with a `DISTINCT` clause. Ganguly et al. (2003) extend the basic FM sketch synopsis structure and propose novel estimation algorithms for estimating *general set-expression* cardinalities over streams of updates. Finally, Considine et al. (2004) propose FM-sketching techniques for approximate, communication-efficient aggregation over wireless sensor networks.

6.4 Summary

AMS and FM sketches represent two important classes of randomized synopsis data structures for streaming data with several applications in stream-processing problems. Besides having a small memory footprint and being easily computable in the streaming model, these sketch synopses can also easily handle deletions in the streams. An additional benefit of both

AMS and FM sketches is that they are *composable;* that is, they can be individually computed over a distributed collection of sites (each observing only a portion of the stream) and then combined (e.g., through simple addition or bit-wise OR) to obtain a sketch summary of the overall stream.

Several other types of (deterministic and randomized) stream synopses have been proposed for different streaming problems. Vitter's reservoir-sampling scheme for constructing a uniform random sample over an insert-only stream (Vitter, 1985) is probably one of the first known stream-summarization techniques. Greenwald and Khanna (2001) and Manku and Motwani (2002) propose deterministic, small-footprint stream synopses for computing approximate quantiles and frequent itemsets, respectively. Datar et al. (2002) consider the problem of maintaining approximate counts over a sliding window of an input stream; their proposed (deterministic) *exponential histogram* synopses employ histogram buckets of exponentially-growing sizes and require space that is only poly-logarithmic in the size of the sliding window. Other stream-synopsis structures for sliding-window computation have been recently proposed by Gibbons and Tirthapura (2002), and Arasu and Manku (2004).

7. DISCUSSION

We wish to raise two points in closing. The first is that there are areas of overlap among the various techniques described in this chapter. For example, a windowed aggregate query is not that different from a group-by query on the window attribute with appropriate punctuation. Both serve to unblock a normally blocking operation, and both limit the amount of state the operations in a query must maintain. The second is that these techniques can sometimes be used in combination. For example, the Data Triage architecture of the TelegraphCQ system switches to computing a synopsis of an incoming data stream when it must drop tuples because it cannot keep up with the current data rate (Reiss and Hellerstein, 2004).

ACKNOWLEDGEMENTS

We would like to thank Leonidas Fegaras, Jin Li, Vassilis Papadimos, Tim Sheard and Kristin Tufte for discussions on punctuations, window queries and disorder, as well as Rajeev Rastogi for numerous discussions on stream synopses. The first two authors were supported in part by DARPA through NAVY/SPAWAR contract N66001-99-108908 and by NSF ITR award IIS 0086002.

REFERENCES

Alon, N., Gibbons, P., Matias, Y., Szegedy, M., 1999, Tracking join and self-join sizes in limited storage, in *Proceedings of ACM PODS Conference*, pp. 10–20.

Alon, N., Matias, Y., Szegedy, M., 1996, The space complexity of approximating the frequency moments, in *Proceeding of ACM STOC Conference*, pp. 20–29.

Arasu, A., Babu, S., Widom, J., 2003, The CQL continuous query language: semantic foundations and query execution, Stanford University TR No. 2003-67 (unpublished).

Arasu, A., Manku, G. S., 2004, Approximate counts and quantiles over sliding windows, in *Proceedings of ACM PODS Conference,* pp. 286-296.

Babu, S., Srivastava, U., Widom, J., 2004, Exploiting k-constraints to reduce memory overhead in continuous queries over data streams, *ACM TODS*, **29**(3):545–580.

Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S., 2002, Monitoring streams – A new class of data management applications, in *Proceedings of VLDB Conference*, pp. 215–226.

Charikar, M., Chen, K., Farach-Colton, M., 2002, Finding frequent items in data streams, in *Proceedings of ICALP Conference*, pp. 3–15.

Cisco Systems, 2001, *Netflow Services Solutions Guide*.

Considine, J., Li, F., Kollios, G., Byers J., 2004, Approximate aggregation techniques for sensor databases, in *Proceedings of IEEE ECDE Conference*, pp. 449–460.

Das, A., Gehrke, J., Riedewald, M., 2003, Approximate join processing over data streams, in *Proceedings of ACM SIGMOD Conference,* pp. 40–51.

Das, A., Riedewald, M., Gehrke, J., 2004, Approximation techniques for spatial data, in *Proceedings of ACM SIGMOD Conference*, pp. 695–706.

Datar, M., Gionis, A., Indyk, P., Motwani, R., 2002, Maintaining stream statistics over sliding windows, in *Proceedings of SODA Conference,* pp. 635-644.

Dobra, A., Garofalakis, M., Gehrke, J., Rastogi, R., 2002, Processing complex aggregate queries over data streams, in *Proceedings of ACM SIGMOD Conference*, pp. 61–72.

Dobra, A., Garofalakis, M., Gehrke, J., Rastogi, R., 2004, Sketch-based multi-query processing over data streams, in *Proceedings of EDBT Conference*, pp. 551–568.

Feigenbaum, J., Kannan, S., Strauss, M., Viswanathan, M., 1999, An approximate L^1-difference algorithm for massive data streams, *Proc. IEEE FOCS Conference*, p. 501.

Flajolet, P., Martin, N., 1985, Probabilistic counting algorithms for data base applications, *JCSS Journal*, **31**(2):182–209.

Ganguly, S., Garofalakis, M., Rastogi, R., 2003, Processing set expressions over continuous update streams, in *Proceedings of ACM SIGMOD Conference*, pp. 265–276.

Garofalakis, M., Kumar, A., 2003, Correlating XML data streams using tree-edit distance embeddings, in *Proceedings of ACM PODS Conference*, pp. 143–154.

Gehrke, J., Korn, F., Srivastava, D., 2001, On computing correlated aggregates over continual data streams, in *Proceedings of ACM SIGMOD Conference*, pp. 13–24.

Gibbons, P., 2001, Distinct sampling for highly-accurate answers to distinct values queries and event reports, in *Proceedings of VLDB Conference*, pp. 541–550.

Gibbons, P., Tirthapura, S., 2002, Distributed streams algorithms for sliding windows, in *Proceedings of ACM SPAA Conference,* pp. 63-72.

Gilbert, A. C., Kotidis, Y., Muthukrishnan, S., Strauss, M., 2001, Surfing wavelets on streams: one-pass summaries for approximate aggregate queries, in *Proceedings of VLDB Conference*, pp. 79–88.

Gilbert, A. C., Guha, S., Indyk, P., Kotidis, Y., Muthukrishnan, S., Strauss, M., 2002, Fast, small-space algorithms for approximate histogram maintenance, in *Proceedings of ACM STOC Conference*, pp. 389–398.

Greenwald, M. B., Khanna, S., 2001, Space-efficient online computation of quantile summaries, in *Proceedings of ACM SIGMOD Conference,* pp. 58-66.

Hillston, J., Kloul, L., 2001, Performance investigation of an on-line auction system, *Concurrency and Computation: Practice and Experience*, 13:23-41.

Indyk, P., 2000, Stable Distributions, Pseudorandom generators, embeddings, and data stream computation, in *Proceedings of IEEE FOCS Conference*, p. 189.

Johnson, T., Cranor, C., Spatscheck, O., Shkapenyuk, V., 2003, Gigascope: A stream database for network applications, in *Proc. ACM SIGMOD Conference*, pp. 647–651.

Kang, J., Naughton, J. F., Viglas, S. D., 2003, Evaluating window joins over unbounded streams, in *Proceedings of ICDE*.

Manku, G. S., Motwani, R., 2002, Approximate frequency counts over data streams, in *Proceedings of VLDB Conference*, pp. 346-357.

Rajasekar, A., Vernon, F., Hansen, T., Linquist, K., Orcutt, J., 2004, Virtual object ring buffer: A framework for real-time data grid, in *Proceedings of HDPC Conference*.

Reiss, F., Hellerstein, J. M., 2004, Data triage: An adaptive architecture for load shedding in TelegraphCQ, Intel Research Berkeley Report IRB-TR-04-004.

Tatbul, N., Çetintemel, U., Zdonik, S., Cherniack, M., Stonebraker, M., 2003, Load shedding in a data stream manager, in *Proceedings of VLDB Conference*, pp. 309–320.

Tucker, P. A., Maier, D., 2003, Dealing with disorder, in *MPDS Workshop*.

Tucker, P. A., Maier, D., Fegaras, L., T. Sheard, 2003, Exploiting punctuation semantics in continuous data streams, *IEEE TKDE*, 15(3):555–568.

Vitter, J. S., 1985, Random sampling with a reservoir, *ACM Trans. on Math. Softw*., 11(1):37-57.

Wilschut, A. N., Apers, P. M. G., 1991, Dataflow query execution in a parallel main-memory environment, in *Proceedings of PDIS Conference*, pp. 68-77.