Sampling Algorithms in a Stream Operator

Theodore Johnson
AT&T Labs Research
johnsont@research.att.com

S. Muthukrishnan
Rutgers University
muthu@cs.rutgers.edu

Irina Rozenbaum
Rutgers University
rozenbau@cs.rutgers.edu

ABSTRACT

Complex queries over high speed data streams often need to rely on approximations to keep up with their input. The research community has developed a rich literature on approximate streaming algorithms for this application. Many of these algorithms produce *samples* of the input stream, providing better properties than conventional random sampling. In this paper, we abstract the stream sampling process and design a new *stream sample operator*. We show how it can be used to implement a wide variety of algorithms that perform sampling and sampling-based aggregations. Also, we show how to implement the operator in Gigascope - a high speed stream database specialized for IP network monitoring applications. As an example study, we apply the operator within such an enhanced Gigascope to perform *subset-sum sampling* which is of great interest for IP network management. We evaluate this implemention on a live, high speed internet traffic data stream and find that (a) the operator is a flexible, versatile addition to Gigascope suitable for tuning and algorithm engineering, and (b) the operator imposes only a small evaluation overhead. This is the first operational implementation we know of, for a wide variety of stream sampling algorithms at line speed within a data stream management system.

1. INTRODUCTION

Many applications, such as network monitoring, financial monitoring, sensor networks, and the processing of large scale scientific data feeds, produce data in the form of high-speed streams. A query set which analyzes these streams might and often does resort to approximation algorithms in order to keep up with the worst case load. The research community has developed a large body of work to approximate expensive functions on data streams. Examples include approximation algorithms for quantiles, heavy hitters, set resemblance, count distinct, and so on. See [4] for an overview of data stream research.

We focus on sampling methods for data streams. A *sample* is a small-sized representative of the data suitable for different purposes. Sampling has a rich history in statistics with several variants: sampling with/without replacement, biased sampling, fixed or variable size sampling etc. There is also extensive use of sampling in databases with many modified methods such as stratified, congressional, outliner or distance-based sampling etc. [6]. Sampling in the context of data streams shares some common aspects with sampling in statistics and databases, but has additional constraints. In stream sampling, typically one is interested in sampling in one pass over a high speed data that can not be stored at its matching rate. As a result, when an item repeats on the stream, it is difficult to sample based on whether or not it has been seen before. So, even uniform sampling of the *distinct* items in the data stream is tricky. Further, one may need to obtain fixed-sized sample when the size of the stream is unknown. Finally, stream input has many attributes and items are often ``weighted'' and it is difficult to ensure that the sample has desirable properties - such as it captures the heavy hitters or sub-range aggregates - accurately for various subset combinations of attributes and cumulative weights on these combinations. The past few years have seen the design of many effective stream sampling methods for estimating specific aggregates such as quantiles [18], heavy hitters [3], distinct counts [19], subset-sums [2], set resemblance and rarity [10] etc. as well as generic sampling such as fixed-size reservoir sampling [1], adaptive geometric sampling [7][8], etc.

The focus of this paper is not to design new stream sampling methods. Instead, we address the problem of how these widely varied and quite sophisticated sampling methods can be implemented within an operational data stream management system and scale in performance to line speeds in IP network monitoring applications. The problem we address in this paper is to incorporate approximate streaming algorithms into a DSMS, specifically sampling-based algorithms.

Possible Approaches: There are several approaches to doing this integration, which we discuss here.

The first approach is to incorporate the different sampling algorithms directly into the DSMS kernel, and make the option of using them available to the user through several keywords. This approach is attractive when the special techniques being incorporated into the database engine are mature, for example data mining keywords in SQL Server 2005 [11], windowing keywords in SQL 99 [12], and so on. However, stream sampling algorithms is an active research area with new techniques being continually developed. Incorporating new techniques into the kernel is cumbersome and does not promote experimentation. In addition, the query language is burdened with a keyword explosion. Aurora incorporates a DROP operator which performs random sampling to shed load [13]; STREAM also provides operator-level sampling via a SAMPLE keyword [19].

The second approach is to implement individual stream sampling algorithms with User Defined Aggregate Functions (UDAF). This approach was explored in [14] for one of the methods, namely, approximating heavy hitter frequency counts by sampling [3]. While the UDAF approach is useful for obtaining point values (e.g., the median packet length), it is cumbersome at best for obtaining set values. For example, to obtain set of destination IP addresses responsible for at least 1% of traffic using the UDAF approach, we could write a query with 100 references to the heavy hitters UDAF (one for each of the possible 100 heavy hitters) in the SELECT clause, *pivot* [15] the result to get the set value, and filter out invalid values. While set results are not inherently better than point results, many applications require set results as their input. In addition, some algorithms, such as subset-sum sampling [2] are better expressed as a sampling query. ATLaS [22] is a system in which a UDAF is specified in SQL. Its set-oriented nature makes set-valued return results possible. As will be evident later, our operator is in some ways a highly structured version of an ATLaS UDAF. The structure we impose enables the simple expression of many algorithms, and a highly efficient evaluation process.

The third and related approach is to provide for User Defined Operators (UDOs) which consume input streams and produce output streams, one for each of the stream sampling methods. Some DSMSs provide a mechanism to incorporate UDOs, including Aurora [13] and Gigascope [21]. Aurora is built as a system of interconnecting operators, and by nature supports UDOs. Gigascope has special facilities for incorporating UDOs into a query set. However, writing and supporting a DSMS operator is a difficult and error-prone task and does not scale with the number of different stream sampling methods of interest. Our discussions with the Gigascope implementers indicated that few ODOs had been written, and only as a last resort.

Our Approach: The approach we take in this paper is to develop a *single* operator which can be specialized to implement a wide variety of stream sampling algorithms. The advantage of this approach is that it encourages experimentation and development of new streaming algorithms and their rapid deployment for practical applications. The functions which support the streaming algorithm using the operator for different problems can be written by the algorithmic expert, following a simple API. The developer is not burdened with the details of kernel integration or stream operator development. Our contributions are as follows.

- *We abstract an operator construct and define its semantics*. We show that this generic operator can be used to implement wide variety of stream sampling algorithms including the reservoir sampling [1], subset-sum sampling [2], min-wise hash sampling [10], heavy hitter algorithm [3], and many others.
- *We show how to implement the generic operator in a data stream management system (DSMS).* The sample operator is invoked using special keywords in a grouping and aggregation query. We detail an efficient templatized implementation of the sample operator. These constructs, as well as STATEFUL functions we introduce, may be of independent interest in conventional data warehouse DBMSs because of their ability to support approximation queries.
- *We perform an experimental study by implementing our sampling operator in the Gigascope DSMS.* We use this implementation to present a detailed study of one of the stream sampling algorithms of great interest to IP network management, namely, subset-sum sampling [2] that is operationally used for performance monitoring in AT&T's IP backbone and for customer reports. Our implementation works at line speed and is now part of the Gigascope release; it shows that the computational and memory overhead is very small. In addition, our experience with real data revealed its burstiness and led to a small fix in the subset-sum stream sampling algorithm that substantially improved its performance. The ability to do such easy tuning and engineering is one of the attractions of our approach.

Our operator is specifically targeted at stream sampling algorithms and can be used to implement scores of them. In the paper, we have chosen to focus on four representatives: reservoir sampling for standard fixed-size sampling on streams, heavy hitter algorithm from the database community, min-wise hash sampling from the algorithms community and subset-sum sampling from the networking community. We believe our work is the first to operational implementation we know of, for a widely variety of stream sampling algorithms at line speed within a DSMS.

Map: In Section 2, we discuss related work. In Section 3, we provide an overview of the Gigascope DSMS. In Section 4, we present an overview of the four stream sampling methods above and describe their common framework. In Section 5, we present our operator and show how it can be used generically to implement different stream sampling algorithms. In Section 6 we discuss STATEFUL functions, SUPERGROUPs and show how to implement our operator in Gigascope. In Section 7, we present our experimental study. Conclusions are in Section 8.

2. RELATED WORK

Sampling has an extensive history in statistics and relational databases. We focus on stream sampling. As mentioned earlier, a number of specific sampling algorithms have been designed for quantiles [18], heavy hitters [3], distinct counts [19], subset-sums [2], set resemblance and rarity [10], geometric sampling for range counting [7] and adaptive sampling for convex hulls [8], etc. Many of these have been implemented and tested on reasonable streams, but few, to the best of our knowledge, on IP network line speeds at which packets are forwarded. In [14], the authors implemented the heavy hitters' algorithm [3] as a UDAF in line speed. In [2] the subset-sum sampling method is implemented at IP flow speeds and not at packet speeds; flows are several orders of magnitude aggregated from packet streams.

There are a number of DSMSs being developed: Aurora [13], STREAM [19], Gigascope [21], TelegraphCQ [17], NiagaraCQ [16], etc. Many of them support random sampling, including the

DROP operator of Aurora, the SAMPLE keyword in STREAM, and sampling functions in Gigascope. Still, these are uniform sampling operators. We do not know of prior work on these systems that systematically implemented a variety of sophisticated stream sampling methods.

3. GIGASCOPE

In this section, we briefly review some relevant aspects of the Gigascope DSMS and its architecture. The interested reader is referred to [21][14] for a more complete description.

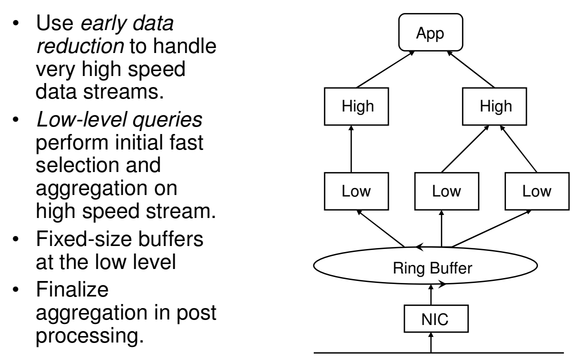
A primary requirement of a DSMS is to provide a way to unblock otherwise blocking operators such as aggregation and join. Different DSMSs take different approaches, but in general they provide a way to define a *window* on the data stream on which the query evaluation will occur at any moment in time. In Gigascope, one or more attributes of a data stream are marked as being *ordered*. Query evaluation windows are determined by analyzing how a query references the ordered attributes. For example, consider the following schema.

```
        PKT(time increasing, srcIP, destIP, len)
```

The `time` attribute is marked as being ordered, specifically increasing. Then the following query computes the sum of the length of packets between each source and destination IP address for every minute

```
        Select tb, srcIP, destIP, sum(len)
        From PKT
        Group by time/60 as tb, srcIP, destIP
```

In order to obtain high performance in data reduction, Gigascope has a two level architecture (see Figure 1). Query nodes which are fed by source data streams (e.g., packets sniffed from a network interface) are called *low level queries*, while all other query nodes are called *high level queries*. Data from a source stream is fed to the low level queries from a ring buffer without copying. Previous work [14][21] has shown that early data reduction by low level queries is critical for high performance.



- Use *early data reduction* to handle very high speed data streams.
- *Low-level queries* perform initial fast selection and aggregation on high speed stream.
- Fixed-size buffers at the low level
- Finalize aggregation in post processing.

Figure 1. Gigascope architecture.

4. STREAM SAMPLING ALGORITHMS

Recall that while approximate stream algorithms can be implemented as UDAFs, they return *point* values rather than set values. That is, to return samples $s_1, s_s, \ldots, s_k$ associated with group G, they return data in a schema such as (G, S1, S2, …, Sn) rather than as (G, S). When we considered the problem of incorporating stream sampling algorithms which return set results into a DSMS, we observed that a large class of these algorithms has a similar control structure. In this section, we survey a representative selection of stream algorithms to illustrate their common structure.

4.1 Reservoir Sampling

The reservoir sampling algorithm [1] solves the problem of selecting a random sample of size n from a pool of N records, where the value of N is unknown. Let T be a tolerance parameter, where $10 < T < 40$; t denote the number of data records processed so far. The current set of candidates for the final sample is stored in the array C. The basic idea of reservoir sampling algorithm can be described as follows:

Within each time window:

- Make first n data records candidates for the sample by saving them into reservoir of size Tn.
- Process the rest of the record within the time window in the following manner:
    - At each iteration generate an independent random variable $\zeta(n,t)$.
    - Skip over the next ζ data records.
    - Make the next data record a candidate by replacing one at random. The index of the record being replaced is `(`*n*`*random())`, where `random()` is random number generator that returns a real number in the unit interval.
- If the current number of candidates exceeds n records, randomly choose n samples out of the reservoir of candidates.

An independent random variable ζ can be generated in several ways. The fastest version of the algorithm generates ζ in constant time, on the average, by a modification of von Neumann's rejection-acceptance method and runs in average time $O(n(1+\log(N/n)))$, which is optimal, up to a constant factor.

4.2 Heavy Hitters

The *heavy hitters* problem is to find the elements in a data stream which account for at least ε fraction of the all tuples. A fast and simple heavy hitters algorithm was proposed by Manku and Motwani [3]. Let f_e be the true frequency of element e in the stream. The incoming stream is conceptually divided into buckets of width $w = \lceil 1/\varepsilon \rceil$ transactions each, where ε is an error bound. Buckets are labeled with bucket id starting from 1. The current bucket id is calculated as $b_{current} = \lceil N / w \rceil$, where N is current length of the stream. The algorithm also uses a parameter s (support): for all collections of transactions, an itemset $X \subseteq I$, where I is universe of all items, is said to have support s if X occurs as a subset in at least a fraction s of all transactions. The data structure D is a set of entries of the form (e, f, Δ) where e is an element in the stream, f is an integer representing its estimated frequency, and Δ is the maximum possible error in f. Initially D is empty. The algorithm works as follows:

- For every new element e check whether it exists in D. If so, increment its frequency f by 1. Otherwise create a new entry in D of the form $(e, 1, b_{current} - 1)$.
- At the boundary of every bucket iterate over all elements of D. An element (e, f, Δ) is deleted if $f + \Delta \le b_{current}$.
- When a user requests a list of items with threshold s, we output those entries where $f \ge (s - \varepsilon)N$

The algorithm is simple and uses at most $\frac{1}{\varepsilon}\log(\varepsilon N)$ space. Although the output is approximate, the error is guaranteed not to exceed ε, in the sense that if $f_e \ge s$ the algorithm will return element e, and if $f_e < s - \varepsilon$, the algorithm will not return e.

4.3 Min-Hash Computation

The *resemblance*, ρ, of two sets A and B is the size of their intersection divided by the size of their union:

$$\rho(A, B) = | A \cap B | / | A \cup B |$$

A *min-hash signature* [24] is a compressed representation of a set from which one can approximate the resemblance of two sets. Let $h_i(a)$ be a hash function. The signature of set A, $S(A)$, is:

$$s_i(A) = \min(h_i(a) \mid a \in A)$$
$$S(A) = (s_1(A), \ldots s_n(A))$$

If $S(A)$ and $S(B)$ are two min-hash signatures, then

$$\hat{\rho}(A, B) = \sum_{i=1}^{n} I(s_i(A), s_i(B))$$

Where $I(x,y)$ is the indicator function, returning 1 if x=y and 0 otherwise. While any given element $s_i(A)$ can be easily computed in an SQL query, a signature typically contains 100 or more elements, making its expression in SQL quite cumbersome. However, a substitute for the minimum of N hash functions is the N minimum values of a single hash function [24].

In [10], the authors use min-hash to sample uniformly from the set of *distinct* elements in the stream and use it to estimate rarity (the ratio of the number of items that appear once in the stream to the number of distinct items) as well as set similarity between two windowed streams.

4.4 Subset-Sum Sampling

Estimation of sums of sizes of objects sharing a common set of properties is of a particular interest for the network management community. In this context the *subset-sum sampling* algorithm provides a better estimate than random sampling. Like Reservoir sampling, the subset-sum sampling can produce fixed size results. Unlike reservoir sampling, subset-sum sampling provides guarantees on sums of a measure attribute.

The subset-sum sampling algorithm [2] collects a sample S of tuples from R in such a way that we can accurately estimate sums from the sample. We phrase the algorithm in database language by assuming that the schema of R is `(C,x)`, where C is an attribute we use for subset selection (the "color" of a tuple) and x is the measure attribute. Then

$$E\left[\sum (t.x \mid t \in S \wedge t.C = c)\right] = \sum (t.x \mid t \in R \wedge t.C = c)$$

Furthermore, the variance of the subset sum over S is within a factor z (defined below) of the subset sum over R.

In the basic subset-sum sampling algorithm, the user sets a threshold z, which determines the sample size. Each tuple t is sampled with probability $p(x) = \min\{1, t.x / z\}$. In particular, the algorithm uses a counter, initialized to zero, and works in the following manner:

- For every new tuple t, check whether t.x > z. If yes, sample the tuple. Otherwise, add value of the t.x to the small flow counter.
- If tuple was not sampled, check whether counter > z. If yes, subtract z from counter and sample the tuple, setting t.x to z. Otherwise discard the tuple.

The idea behind this algorithm is that tuples with large values of t.x contribute the largest amount to a sum. Therefore all large tuples are sampled; however small tuples cannot be discarded without biasing some subset-sum. The algorithm samples one small tuple every time the combined weight of the small tuples exceeds z. To estimate the sum, the measure t.x of the sampled small tuple is adjusted to z, since it represents a weight of threshold z: $t.x = \max\{t.x, z\}$.

The result of the algorithm described above is a sample of arbitrary size, which introduces an element of unpredictability. In many cases we would like a sample of a particular size, say 1000 samples regardless of the distribution of t.x or the size of R. The second version of the algorithm (*dynamic subset-sum* sampling) will produce a consistent number of sampled tuples. The user specifies the desired sample size N and an initial value of the threshold z. In addition to small tuples count (`count`), the algorithm tracks the number of tuples sampled so far (`sample_count`). The algorithm works in the following manner:

- Collect samples according to the basic subset-sum sampling algorithm, keeping a count of the number of sampled tuples in `sample_count`.
- If `sample_count` > γ*N (e.g., γ=2), estimate a new value of z which will result in N tuples. Subsample S using basic subset-sum sampling and the new value of z, and continue with basic subset-sum sampling.

- When all tuples from R have been processed, if `sample_count` $> N$ then adjust z and subsample S using basic subset-sum sampling.

When applied to a data stream, subset-sum sampling occurs in successive time windows. In this case, an initial threshold can be estimated for the new time window using the threshold from the old time window, adjusting its value to obtain an estimated N samples during the new time window.

The authors of [2] suggest a variety of strategies for adjusting z. In our implementation, we used the *aggressive* version of the z threshold adjustment (z-threshold, $|S|$-currently maintained number of samples, M-desired number of samples, B-number of samples for which sample size > threshold):

If $0 \leq |S| < M$, then $z_{new} = z_{old}(|S|/M)$

If $|S| \geq M$, then $z_{new} = z_{old}((\max(|S|-B,1))/(M-B))$

4.5 Summary

We observe that these stream sampling algorithms are quite sophisticated, and far from "pick each item with some probability" that one expects from uniform sampling. They also solve very different problems and each has found many applications. Still they follow a common pattern. First a number of items are collected from the original data stream according to a certain criteria, and perhaps with aggregation in the case of duplicates. If a condition on the sample is triggered (e.g., the sample is too large), a cleaning phase is triggered and the size of the sample is reduced according to another criteria. This sequence can be repeated several times until the border of the time window is reached and the sample is output. This framework fits each of the summarized algorithms as follows:

- **Subset-Sum sampling**: Sample records according to the basic subset-sum sampling algorithm. Trigger the cleaning phase when `count_sample` > $\gamma*N$. In the cleaning phase, adjust z and subsample.
- **Heavy hitters**: Count the frequency of occurrence for every distinct sample. Trigger the cleaning phase every w input tuples. In the cleaning phase, delete samples according to the defined rules.
- **Min-hash**: Sample a hash value whenever it is within the smallest N of hash values seen thus far. Trigger the cleaning phase when the number of samples exceeds $\gamma*N$. In the cleaning phase, remove the hash values larger than the Nth smallest value seen thus far.
- **Reservoir sampling**: repeatedly generate ζ,, skip that number of records, and select the next record for the reservoir. Trigger the cleaning phase when the sample size exceeds Tn. In the cleaning phase, randomly choose n records from the reservoir to keep and delete the rest.

Our operator in the next section is inspired by the common framework above.

5. THE SAMPLING OPERATOR

From the discussion above, we derive a number of common characteristics for the sampling algorithms in question:

- A "global" state structure.
- A loose predicate for admitting a tuple to the sample.
- A predicate which triggers a sample cleaning phase.
- A predicate for removing samples during the cleaning phase.
- A finishing-off predicate.

The process of sampling is in some ways similar to that of aggregation, as they both collect and output sets of tuples which are representative of the input. Accordingly, our textual representation of the sampling operator is based on the textual representation of aggregation:

```
SELECT <select expression list>
FROM <stream>
WHERE <predicate>
GROUP BY <group-by variables definition list>
[SUPERGROUP <group-by variable list>]
[HAVING <predicate>]
CLEANING WHEN <predicate>
CLEANING BY <predicate>
```

The "global" state structure stores the control variables of the sampling algorithm. For example, in the Manku-Motwani algorithm [3] the state stores variables such as the count of tuples processed since the last cleaning phase and the number of cleaning phases which have been triggered. Since we might wish to obtain a sample on a group-wise basis (e.g., for each source IP address, report the destination IP addresses accounting for at least 10% of the total packets sent from the source IP), we associate the sampling state with *supergroups*, and samples with the groups in a supergroup. The variables in the SUPERGROUP clause must be a subset of group-by variables defined in the GROUP BY clause (thus, supergroups are a specialization of grouping sets [12]). By default, the supergroup is ALL. Along with sampling state variables, the supergroup can compute superaggregates (aggregates of the supergroup rather than the group). One example of a useful superaggregate is `count_distinct$()`, which returns the number of distinct groups in a supergroup (we use the `$` to denote that an aggregate is associated with the supergroup rather than the group).

More concretely, the semantics of a sampling query is as follows:

- When a tuple is received, evaluate the WHERE clause. If the WHERE clause evaluates to false, discard the tuple.
- Else if the condition of the WHERE clause evaluates to TRUE then
    - Create and initialize a new supergroup and a new superaggregate structure if needed, otherwise update the existing superaggregates (if any).
    - Create and initialize a new group and a new aggregate structure if needed, otherwise update the existing aggregates (if any).
    - Evaluate the CLEANING_WHEN clause.
    - If the CLEANING_WHEN predicate is TRUE

        - Apply CLEANING_BY clause to every group.
        - If the condition of CLEANING_BY clause evaluates to FALSE
            - Remove group from the group table, and update any superaggregate
- When the sampling window is finished,
    - Evaluate the HAVING clause on every group.
    - If the condition in the HAVING clause is satisfied, then the group is sampled, else discard the group.

That completes the description of the operator. The discussion thus far is independent of any specific DSMS.

6. THE OPERATOR IN GIGASCOPE

In this section, we discuss how sampling operator interacts with a specific DSMS, namely Gigascope, and is realized in it.

6.1 Sampling operator in Gigascope

The sampling operator in previous section brings up certain details within Gigascope. For example, in the Gigascope DSMS, the sampling window ends whenever any ordered group-by variable changes value, so the sampling operator will produce output once every time window. As a corollary, all ordered group-by variables are part of the supergroup. Also, in some algorithms, e.g., dynamic subset-sum sampling, initial values of a state in a new time window are derived from the state of the old time window. Our implementation of the sampling operator supports this at superaggregate structure initialization time by checking if a supergroup with the same non-ordered group-by variables existed in the previous time window. If so, all states in the new superaggregate are initialized by a function which accepts the equivalent state from the old time window.

For an example, the following Gigascope query expresses the dynamic subset-sum sampling algorithm which collects 100 samples:

```
SELECT uts, srcIP, destIP,
     UMAX(sum(len),ssthreshold())
FROM PKTS
WHERE ssample(len,100) = TRUE
GROUP BY time/20 as tb, srcIP, destIP, uts
HAVING ssfinal_clean( sum(len),
     count_distinct$(*) ) = TRUE
CLEANING WHEN
      ssdo_clean(count_distinct$(*)) = TRUE
CLEANING BY ssclean_with(sum(len)) = TRUE
```

where `UMAX(val1, val2)` is a function which returns the maximum of the two values, and `uts` is a nanosecond granularity timestamp (with its timestamp-ness cast away) used to make each tuple its own group.

The `sshthreshold()`, `ssample()`, `ssfinal_clean()`, `ssdo_clean()` and `ssclean_with()` functions are *stateful functions*, which we discuss in the next section.

To complete the description of the sample operator, we need to discuss some working details, which we do in the context of our implementation in Gigascope.

6.2 Stateful Functions

To implement some of the algorithms, a number of functions need to access the same global state throughout the execution. For this reason, we call those functions *stateful*. Typically, a collection of functions will share the same *state* structure. Stateful functions are very similar to UDAFs, but with the following differences:

- They can produce output a number of times during the execution.
- The state can be modified only when the functions which share the state are referenced.

A state is declared as follows:

```
STATE <type> <name>;
```

The declaration of stateful functions ties the function to the state it shares:

```
SFUN    <type>    [modifiers]    <state_name>
<function_name> (<param_list>)
```

In case of subset-sum sampling algorithm:

```
STATE char[50] subsetsum_sampling_state;
SFUN int subsetsum_sampling_state ssample(int,
 CONST int);
SFUN int subsetsum_sampling_state
 ssfinal_clean(int, int);
SFUN int subsetsum_sampling_state
 ssdo_clean(int);
SFUN int subsetsum_sampling_state
 ssclean_with(int);
SFUN int subsetsum_sampling_state
 ssthreshold();
```

When the query references a new supergroup, the space for the SFUN state is allocated in the superaggregate structure. The state is initialized with its associated initialization function. For example, the prototype of the state initialization function in our implementation of the sampling operator is:

```
void _sfun_state_init_<state name>(<pointer to
memory for the state>, <pointer to old state,
or NULL>);
```

Stateful functions are implicitly passed a pointer to their associated state. In our implementation, the prototype of the stateful functions has the following form:

```
<return type> <name>(void *s, <param_list>);
```

where `s` is the pointer to the state.

In the case of our subset-sum sampling implementation, some of the functions that we added to the Gigascope runtime library are:

```
void
_sfun_state_init_subsetsum_sampling_state(
void* n, void* o);
int ssample(void*s, int len, int sample_size);
```

6.3 Groups and Supergroups

As discussed earlier, very often there is a need to reference global aggregates, or supergroups. For instance, in subset-sum sampling the cleaning phase is triggered when the number of groups exceeds the threshold (it's important to notice that in the subset-sum sampling implementation every packet needs to be distinctly unique, thus every group consists of a single packet). Another example of the query that uses supergroups is the min-hash problem, when we would like to compute *k* min-hash destination IP addresses per source IP address; and hence we need a superaggregare which returns the *k*th smallest value.

There is a difference between regular aggregation and global (super) aggregation. To be able to maintain superaggregate, we need to maintain group aggregate of the same type. When a new group is added or deleted (as a result of the cleaning phase), we need to update the supergroup aggregate by adding or subtracting the group aggregate value. One of the useful superaggregates is `count_distinct$()` which reports the number of groups in the supergroup.

6.4 Sampling Operator Implementation

Our implementation of the sampling operator maintains three types of hash tables: one for the groups, one for the supergroups and an additional table that keeps track of all groups for every supergroup:

Group table:

key – set of group-by variables

value – structure that maintains group aggregates

Supergroup table:

key - set of supergroup variables not including ordered variables (when no supergroup is specified, the key is associated with a single time window).

value – structure that maintains state(s) associated with the supergroup, and any superaggregates.

Supergroup-Group table:

key - set of supergroup variables (when no supergroup is specified, the key is associated with a single time window).

value – list of all groups in this supergroup

Note that the key of the supergroup table is always a subset of elements that represent the key of the group table.

We actually maintain two supergroup hash tables – "old" and "new". The "old" supergroup hash table maintains all the supergroups that were sampled in the previous window.

The evaluation process can be summarized as follows:

- When a tuple is received, compute the key for the supergroup table using group-by variables.
- If at the border of the window, call `final_init()` function for the states in the new supergroup table (to signal to the state that the time window is finished) and apply HAVING clause to every group of the new group hash-table. Clear the group table, the old supergroup table, and the supergroup-group table, and move the new supergroup table to the old supergroup table.
- If the supergroup of the newly arrived tuple exists in the new supergroup table, then apply WHERE condition to the tuple. If the condition evaluates to TRUE, update superaggregates of the supergroup, else start processing next tuple.
- If the supergroup doesn't exist in the new supergroup table, check whether the supergroup with the same key exists in the old supergroup table. If so, initialize the state of the new supergroup by using `state_init()` function, passing a pointer to the old state as the second argument. If the supergroup is entirely new, pass a NULL as the second argument. Create a new supergroup in new hash table. Apply WHERE condition to the tuple. If the condition evaluates to TRUE, update superaggregates.
- Compute key for the group table using group-by variables.
- If the group with this key exists in the new group hash-table, update group aggregates.
- If the group doesn't exist, create a new group and new aggregates of the group. Add the key of the group to the supergroups' entry in the supergroup-group table.
- Apply the CLEANING WHEN condition to the supergroup state. If the condition evaluates to TRUE, trigger the cleaning phase by applying CLEANING BY clause on every group that belong to the current supergroup (i.e., using the supergroup-group hash-table). If the condition evaluates to FALSE, then delete the group from the group hash-table and remove its key from the supergroup's supergroup-group table.
- Stateful functions that appear in SELECT clause will be evaluated last, when the output tuple is created.

6.5 Evaluation Example

Let us consider an example of the subset-sum sampling algorithm. The global structure of the algorithm uses a number of parameters, such as the value of the threshold z, the counter of small packets `count`, the counter of large packets `bcount`, value of the cleaning threshold γ, etc. The evaluation process of the query that expresses the algorithm is as follows:

- When the tuple is received, call `ssample()` function:

  The loose predicate for admitting a tuple to the sample is the basic subset-sum sampling predicate using the current value of z. If the function returns false, then the predicate condition had failed and we start processing next tuple. If the function returns true, process the tuple by creating (or updating) appropriate entries for supergroup, group and supergroup-group hash tables.
- Call `ssdo_clean()` function:

  The cleaning phase is triggered when the current sample size exceeds the threshold of the number of samples that can be maintained by currently processed supergroup. If the function returns false, the condition is not met and we start processing next tuple. Otherwise, z is adjusted and the cleaning phase is triggered.
- Call `ssclean_with()` function on every group of currently processed supergroup. The current sample is cleaned by applying the new value of threshold for the size of the data record and deleting those records which don't meet the

cleaning condition. The cleaning condition states that if the size of the data record < value of the threshold before the most recent adjustment (*z_prev*), then *z_prev* will replace size of the record during the cleaning phase.

- Call `ssfinal_clean()` at the border of every window. If the number of samples still exceeds the desired size of the final sample, do the final subsampling. This function implements the final cleaning condition which is identical to the cleaning condition implemented in ssclean_with() function. If the function call returns false, the group is evicted from the hash table. Otherwise, the group is sampled and the output tuple is created.

6.6 Query Examples

Although we have focused on the dynamic subset-sum sampling implementation, in this section we show how the other three algorithms from our representative four can be implemented using the generic sampling operator.

Query for Heavy Hitters Algorithm: This query will report the 100 most common source addresses within a time window of 1 minute. The function `current_bucket()` returns id of current bucket. The aggregate `first()` returns the first value that was returned by `current_bucket()` function within current time window. The function `local_count(N)` increments `current_bucket` and returns true once every N calls.

```
SELECT tb, srcIP, sum(len), count(*)
FROM TCP
GROUP BY time/60 as tb, srcIP
CLEANING WHEN local_count(100) = TRUE
CLEANING BY count(*) < current_bucket()-
       first(current_bucket())
```

Query for Min-Hash Computation: This query will report 100 min-hash values of destination IP addresses per source IP address. This query does not make use of stateful functions but instead relies on the `count_distinct$(*)` and the `Kth_smallest_value$(HX,100)` superaggregates `(Kth-smallest_value(x,n)` returns the nth smallest value of x).

```
SELECT tb, srcIP, HX
FROM TCP
WHERE HX <= Kth_smallest_value$(HX, 100)
GROUP_BY time/60 as tb, srcIP, H(destIP) as HX
SUPERGROUP BY tb, srcIP
HAVING HX <= Kth_smallest_value$(HX, 100)
CLEANING WHEN count_distinct$(*) >= 100
CLEANING BY HX <= Kth_smallest_value$(HX, 100)
```

Query for Reservoir Sampling Algorithm: This query will return 100 random samples per time window of 1 minute. The function `rsample(100)` implements the sampling condition by returning true for those tuples that should be saved in the reservoir of candidate tuples, and returning false for those that are skipped over. The function `rsdo_clean()` returns true when the number of candidates (`count_distinct$()`) exceeds the threshold value of *Tn*, and returns false otherwise. The functions `rsclean_with()` and `rsfinal_clean()` randomly subsample *n* final samples the reservoir of candidates:

```
SELECT tb, srcIP, destIP
FROM TCP
WHERE rsample(100) = TRUE
GROUP_BY time/60 as tb, srcIP, destIP
HAVING rsfinal_clean() = TRUE
CLEANING WHEN
    rsdo_clean(count_distinct$()) =  TRUE
CLEANING BY rsclean_with() = TRUE
```

7. EXPERIMENTS

We implemented the sampling operator in the Gigascope DSMS in order to experiment with the feasibility and performance of the operator. The Gigascope implementers also provided us with access to several network data streams. We implemented not only the operator, but also amended the parser and query analyzer to instantiate the sampling operator from a query with the textual representation described in Section 5.

In our experiments, we focus on the dynamic subset-sum sampling algorithm. The dynamic subset-sum sampling algorithm is used extensively in the AT&T network performance monitoring infrastructure [2], and consequently this algorithm is well understood by the Gigascope developers. In addition, the Gigascope developers indicted that dynamic subset-sum sampling is a good first algorithm because of the demand for its use. Our implementation of dynamic subset-sum sampling follows the description given in Section 5.

We had two network feeds available for experiments. The first is the network connection to our research center. This data stream produces a moderate 5,000 to 15,000 packets per second, with a rate that is highly variable. The second network feed is a data center tap, producing moderately high speed 100,000 packets per second (about 400 Mbits/sec). This data feed is highly aggregated, and hence has a much lower variability in its data rate than the first. When testing accuracy, we generally use the first data feed because its high variability will tend to emphasize estimation problems. When testing performance, we generally use the second data feed because its low variability and high data rate make measurements much more consistent. For all experiments, we used an inexpensive dual 2.8 GHz processor server.

7.1 Accuracy

We measured the accuracy of the dynamic subset-sum sampling algorithm by running two query sets simultaneously. One computed the sum of packet lengths during successive 20 second intervals, and the other applied dynamic subset-sum sampling to collect 1000 samples of packets, then computed the sum of (subset-sum sample adjusted) packet lengths for each time interval. We found that on many of the time intervals, the dynamic subset-sum sampling algorithm is inaccurate. This property is illustrated in Figure 2, where the aggregate result is labeled "actual" and the dynamic subset-sum sampling result is labeled "estimated (non-relaxed)".

The problem lies in the threshold update procedure discussed in 4.4. The load during the next interval is estimated to be the load during this interval; if the load drops sharply, dynamic subset-sum sampling collects too few samples and underestimates the sum.

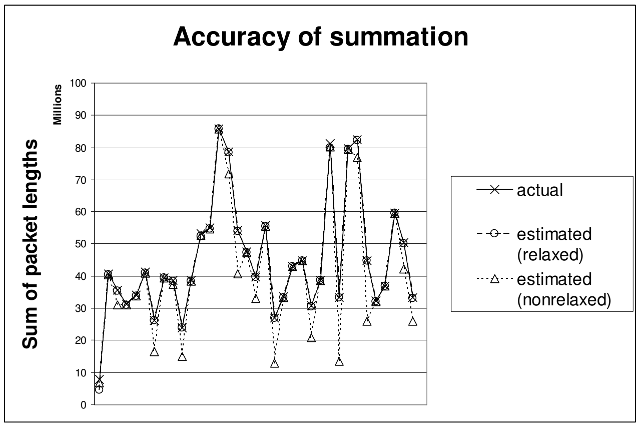


Figure 2. 1000 samples per period.

To correct this problem, we made a minor adjustment to the dynamic subset-sum sampling so that it will estimate that the load in the next time period is a fraction *1/f* of the load during this interval. We call this the *relaxed* version. In Figure 2 we use *f=10* and the relaxed estimates match the actual sum very closely for all time periods. The relaxed algorithm works well because the cleaning phases readily adapt the threshold upward to the appropriate value.

Another illustration of the problem with non-relaxed subset-sum sampling is shown in Figure 3. The relaxed algorithm occasionally over-samples, while the non-relaxed algorithm frequently under-samples causing an underestimation.

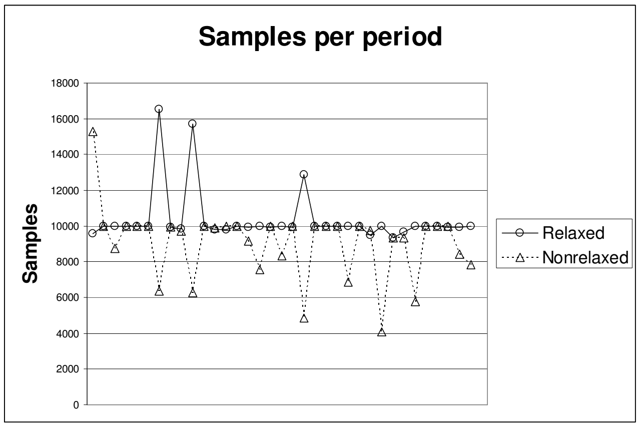


Figure 3. 1000 samples per period.

The cost of the relaxed algorithm is that the cleaning phase is invoked more frequently. Figure 4 shows the number of cleaning phases for the relaxed and non-relaxed dynamic subset-sum sampling algorithms during the experiment. The first interval was very short (as can also be seen from the other charts). In the second interval, both algorithms used a large number of cleaning phases to identify the appropriate threshold; afterwards the number of cleaning phases stabilized at a low level. The relaxed algorithm consistently used about 4 cleaning phases, as compared to 1 for the non-relaxed algorithm. If the cost of the cleaning phase is small (which we explore in the next section), using the relaxed algorithm incurs only a small overhead.

We repeated these experiments to collect 100 and 10,000 samples per period, and obtained nearly identical results (a user will collect a larger or smaller number of samples depending on storage costs and the degree of subsetting during analysis).

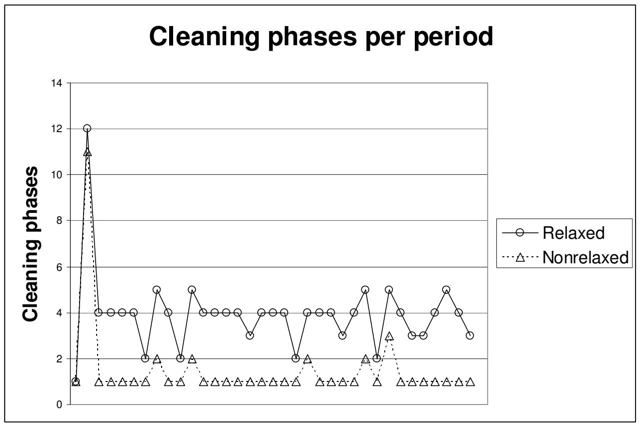


Figure 4. 1000 samples per period.

7.2 Performance

To evaluate the CPU overhead of running adaptive subset-sum sampling using our sampling operator, we ran both the relaxed and the non-relaxed dynamic subset-sum sampling algorithms on the high speed link (100,000 packets/sec), as the CPU utilization of these queries on the moderate speed link is too low to measure accurately. For a comparison, we also ran basic subset-sum sampling using a user-defined function in a selection operator. A comparison of the CPU usage for each of these algorithms is shown in Figure 5. Even when processing 100,000+ packets/sec and producing large outputs, the dynamic subset-sum sampling algorithm implemented using the sampling operator uses only a small fraction of a CPU (two CPUs are available at the server). Compared to the selection query (basic subset-sum sampling), the sampling operator uses only about 3% to 5% additional CPU load. The cost of the additional cleaning phases to support relaxed subset-sum sampling can be seen in this chart. However the overhead is small, at most about 2% of CPU for this experiment.

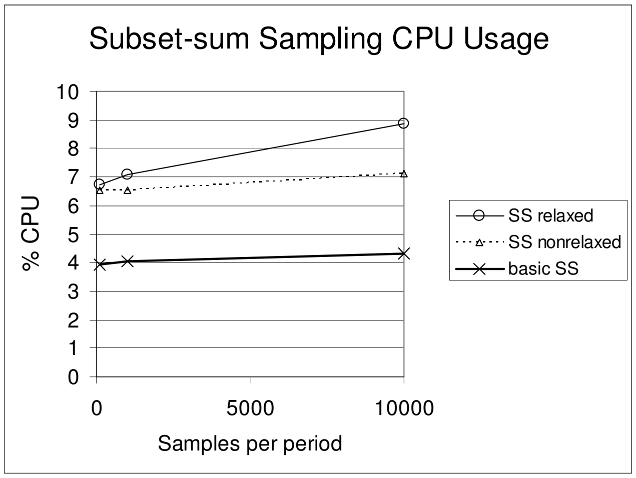


Figure 5. CPU usage for sampling.

However, there is a problem with this implementation of dynamic subset-sum sampling. Recall that there are two types of queries nodes in the Gigascope architecture: low level queries which read from the network interface, and high level queries which read from Gigascope-managed query streams. The low-level queries nodes are simple data reduction operators. Currently only selection and (partial) aggregation are supported. Therefore we need to run a low-level selection query to feed the subset-sum sampling queries. In the run of experiments shown in Figure 5, evaluating the low-level query required about 60% of a CPU, due to the cost of memory copies.

Fortunately, it is possible to evaluate part of a subset-sum sampling query at the low-level query. We modified the low-level selection query to have it perform basic subset-sum sampling with a threshold 1/10th the level used by the dynamic subset-sum sampling algorithm when it returns 10,000 samples per interval. The low-level query load dropped to about 4% of a CPU. In addition, the dynamic subset-sum sampling CPU load dropped significantly, as shown in Figure 6.

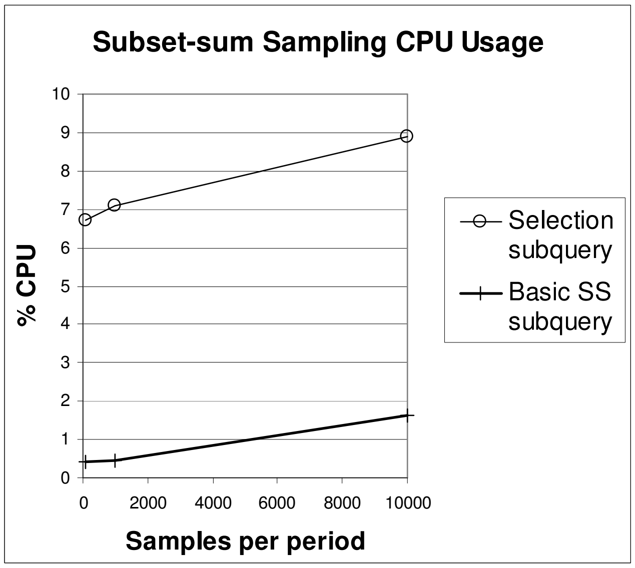


Figure 6. Effect of low-level query type.

We ran additional experiments regarding the setting of γ (the trigger to initiate a cleaning phase). Increasing (decreasing) γ decreases (increases) the number of times cleaning is done, but increases (decreases) its cost. We found little dependence of CPU load on γ.

8. CONCLUSION

Query sets which make use of very high speed data streams must often use approximate data reduction strategies to provide complex statistics while keeping up with the offered data load. A useful approximation technique is *sampling*, which reduces the data set into a much smaller and yet representative result. Typical sampling methods are often quite simple: sample each item with some probability, say *p*. But in streaming context, even uniform sampling from distinct elements on the stream is a challenge. Over the past few years, researchers have proposed very sophisticated sampling algorithms on streams for a variety of problems. Rather than propose new stream sampling methods, we have focused on how to implement the many intricate sampling methods in the literature. Our approach has been to abstract and propose a new stream operator for evaluating sophisticated sampling algorithms, on a data stream. This operator is powerful enough to evaluate many widely different stream sampling algorithms including subset-sum sampling (from networking), reservoir sampling (from databases), min-hash sampling (from theoretical algorithms), etc., as well as sampling-based aggregation algorithms such as the Manku-Motwani heavy hitters' algorithm, and many more. We urge the readers to try modeling other stream sampling algorithms via our stream operator to appreciate its flexibility and generality. Some of our ongoing work consists of cascading one type of stream sampling inside a different type of stream sampling group; we will report on those results in the journal version of this paper.

We implemented the sampling operator in the Gigascope DSMS, and implemented dynamic subset-sum sampling on top of that. We made a performance evaluation of dynamic subset-sum sampling on both highly variable and high speed data streams. We found that

- The accuracy of the dynamic subset-sum sampling algorithm can be greatly improved by *relaxing* the threshold between time windows. This was re-engineering that was a result of experience with the real system.
- The sampling operator imposes only a small CPU overhead, as compared to a simple selection operator. We can readily scale subset-sum sampling to much higher data rates.
- By performing part of the subset-sum sampling at the low level query, we can collect a 1% subset-sum sample on a high speed data stream using less than 6% of a CPU.

Obtaining the best performance from a DSMS such as Gigascope requires a significant amount of early data reduction at the low-level queries. The method for doing this will depend on the approximation algorithm. For example, the Manku-Motwani heavy hitters algorithm would be best supported by aggregation at the low-level queries. We have not explored operator transforms in this paper, but we have gained valuable query optimization tips during our experimental study.

The significance of our results is that we have developed a simple way in which sophisticated streaming algorithms that returns set results can be integrated into a query system. The supporting UDAFs and functions need only follow a simple API. Once written, the user has the power of the query language to explore new combinations. This ease of experimentation allowed us to find the simple upgrade of subset-sum sampling which so improved its accuracy. The relaxed version of subset-sum sampling, along with the sampling operator, has been incorporated into the release version of Gigascope. This implementation is the first one that we know of in an operational DSMS which can handle line speeds.

Our success stems from our observation that a large class of sampling algorithms have an essentially simple communication structure, namely between individual samples and a sample summary only. We have focused on this core aspect of sampling algorithms. We note that it is quite possible to derive sampling-based algorithms which operate on the samples in more complex ways and therefore require a far more complex communication structure. An excellent example is a more-holistic sampling algorithm such as the Greenwald-Khanna quantile algorithm [18]. The *compress* phase of this algorithm merges adjacent samples, and thus requires inter-sample communication. This algorithm (expressed as a UDAF in [14]) and others which may have such computations on samples built into them, are best expressed using a stream UDAF on top of the sampling operator we have developed here. In contrast, all sampling algorithms that work on a per-sample tuple basis can be implemented using our sampling operator.

In addition to capturing capturing a common thread of evaluation of a large variety of sampling algorithms, our sampling operator is able to maintain information about groups and supergroups in terms of aggregates and superaggregates required for implementation and statistical analysis of a sampling algorithm. We believe that this, along with stateful functions, gives the user the level of flexibility required for implementation and customization of various sampling-related algorithms. Our work with subset-sum sampling demonstrated this, but we provide another example.

The following example demonstrates the flexibility of the sampling operator. In network traffic analysis it is often useful to perform network measurements using flow statistics rather than packet statistics, since flows offer a considerable compression of information over packet headers. The straightforward implementation of this approach in terms of the stream sampling operator can be expressed as a set of queries, where flow aggregation is performed leveling a first query, and the result is fed to a higher level sampling query. However, this implementation exhibited difficulties under certain network conditions, in particular when there is a large number of small flows consisting of only a few packets (e.g. during DDOS attacks). Under these conditions, the flow aggregation query requires an enormous number of groups (corresponding to the enormous number of flows), exhausts the available memory, and fails. To overcome this problem we modified the implementation of the subset-sum sampling algorithm by integrating flow aggregation with sampling into a single query processing phase. This implementation of the algorithm allows us to create very informative flow samples on streams of network data with a moderate memory overhead. The key trick is that small flows can be quickly sampled and purged from the group table. The new sampled flows query is a more stable implementation which is resistant to rapid network changes. We will report on the details and our experience elsewhere.

Acknowledgements: We thank Oliver Spatscheck at AT&T Research.

9. REFERENCES

[1] J. S. Vitter. Random sampling with reservoir. ACM Transactions on Mathematical Software, 11(1):37-57, 1985.

[2] N. Duffield, C. Lund, M. Thorup. Learn more, sample less: control of volume and variance in network measurement. SIGCOMM 2001 Measurement workshop.

[3] G. Manku and R. Motwani. Approximate frequency counts over data streams. Proc. VLDB, 2002, 346-357.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom. Models and Issues in Data Stream Systems, Proceedings of 21st ACM, PODS 2002.

[5] S. Muthukrishnan. Data stream algorithms and applications. http://www.cs,rutgers.edu/~stream-1-1.ps.

[6] A. Singh. http://www.cs.ucsb.edu/~ambuj/Courses/multimediaDB/sampling.pdf

[7] A. Bagchi, A. Chaudhary, D. Eppstein, and M. Goodrich. Deterministic sampling and range counting in geometric streams. *ACM Symp Computational Geometry* 2004.

[8] J. Hershberger and S. Suri. Adaptive Sampling for Geometric Problems over Data Streams. *ACM PODS '04* (Symp. on Principles of Database Systems) June 13-18, Paris, France.

[9] P.J. Haas, J.F. Naughton, S. Seshadri, L. Stokes, "*Sampling Based Estimation of the Number of Distinct Values of an Attribute*", Proc. VLDB 1995, Zurich, Switzerland.

[10] M. Datar and S. Muthukrishnan. Estmating rarity and similarity on data stream windows. *Proc. ESA*, 2002. 323-334.

[11] SQL Server 2005. http://www.microsoft.com/technet/prodtechnol/sql/2005/evaluate/dwsqlsy.mspx

[12] P. Gulutzan and T. Pelzer, *SQL-99 Complete, Really*, CMP Books, 1999.

[13] D. Carney, U. Cetinternel, M. Cherniack, C. Coney, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul and S. Zdonik. Monitoring Streams – A New Class of Data Management Applications. *VLDB 2002*.

[14] G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, O. Spatscheck, D. Srivastava. Holistic UDAFs at Streaming Speeds, *SIGMOD 2004*.

[15] M. Gyssens , L.V.S. Lakshmanan and I.N. Subramanian, Tables as a Paradigm for Querying and Restructuring, *Proc. ACM PODS 96*, pg. 93-103.

[16] J. Chen, D.J. DeWitt, F. Tian and Y. Wang, NiagaraCQ: A Scalable Continuous Query System for Internet Databases, *SIGMOD 2000* pg. 379-390.

[17] S. Chandrasekaran *et al.* TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. Proc. CIDR 2003.

[18] M. Greenwald and S. Khanna, Space-Efficient Online Computation of Quantile Summaries, *Proc. SIGMOD 2001.*

[19] P.Gibbons. Distinct Sampling for Highly-Accurate Answers to Distinct Values Queries and Event Reports. *Proc. VLDB* 2001: 541-550.

[20] R. Motwani, J. Widom, A. Arasu. B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, R. Varma. Query Processing, Resource Management, and Approximation in a

Data Stream management System. In *CIDR*, pages 245-256, Jan 2003.

[21] C.Cranor, T. Johnson, O. Spatschnek, V. Shkapenyuk. Gogascope: A Stream Database for Network Applications. In *Proc. ACM SIGMOD*, page 262, 2002.

[22] H. Wang, C. Zaniolo and C. Luo, ATLAS: A Small but Complete SQL Extension for Data Mining and Data Streams, *Proc. VLDB 2003* pg 5-20.

[23] Y.-N. Law, H. Wang and C. Zaniolo, Query Languages and Data Models for Database Sequences and Data Streams, *Proc. VLDB 2004* pg 492-503.

[24] A. Broder. On the Resemblance and Containment of Documents, *IEEE Compression and Complexity of Sequences '97* pg. 21-29.