Holistic UDAFs at Streaming Speeds

Graham Cormode*
Rutgers University
graham@dimacs.rutgers.edu

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

Flip Korn
AT&T Labs–Research
flip@research.att.com

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

Oliver Spatscheck
AT&T Labs–Research
spatsch@research.att.com

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

ABSTRACT

Many algorithms have been proposed to approximate holistic aggregates, such as quantiles and heavy hitters, over data streams. However, little work has been done to explore what techniques are required to incorporate these algorithms in a data stream query processor, and to make them useful in practice.

In this paper, we study the performance implications of using user-defined aggregate functions (UDAFs) to incorporate selection-based and sketch-based algorithms for holistic aggregates into a data stream management system's query processing architecture. We identify key performance bottlenecks and tradeoffs, and propose novel techniques to make these holistic UDAFs fast and space-efficient for use in high-speed data stream applications. We evaluate performance using generated and actual IP packet data, focusing on approximating quantiles and heavy hitters. The best of our current implementations can process streaming queries at OC48 speeds (2x 2.4Gbps).

1. INTRODUCTION

The phenomenon of data streams research is evident. This has been led by two research directions in the database community.

First, powerful algorithms have been developed for processing data in a stream. They work in an abstracted model of data streams where items are presented one after another; they use small amount of storage, compute various holistic[1] aggregates on the stream and provide accuracy guarantees. At the high level, these algorithms can be divided into two categories: *data-driven* and *universe-driven*. The data-driven algorithms select one or more data items that appear in the stream and maintain statistics about their distribution in the stream. They typically have space and accuracy bounds a function of n, the number of data items in the stream. Examples of such algorithms are in [32, 24, 33]. We refer to them as *selection-based* algorithms. Typically, the order in which the data is presented influences the performance of these algorithms. The universe-driven algorithms work on a virtual "array" of the attribute values and maintain various inner-products or "sketches" over the data stream. They typically have space and accuracy bounds a function of U, the size of the universe from which the attribute values are drawn. Examples of such algorithms are in [25, 9, 21]. We refer to them as *sketch-based* algorithms. Typically, the order in which the data is presented does not affect the sketch these algorithms maintain, but identical data values over different universe sizes may have different sketches. Selection and sketch based algorithms are known for a range of holistic aggregates including quantiles, heavy hitters, count distinct, rare counts, correlated aggregates, etc.

Second, there is a concerted effort to build data stream management systems (DSMSs) either for general purpose or for a streaming application. Traditional database systems store data, maintain them under transactions and support query processing on a consistent view of the data evolving under transactions. Emerging applications in data streams shift the emphasis of a database system. In data stream applications, data arrives very fast and the rate is so high that one may not wish to (or be able to) store all the data. These DSMSs devise methods to pipeline tuples through query processing mechanisms, and schedule operations to maximize throughput, etc. Many of the DSMSs are motivated by monitoring applications. Example DSMSs are in [6, 35, 36, 7, 12].

The quintessential application seems to be the processing of IP traffic data in the network. Routers forward IP packets at great speed, spending typically a few hundred nanoseconds per packet. Processing the IP packet data for a variety of monitoring tasks — keeping track of statistics, provisioning, billing and detecting network attacks — at the speed at which packets are forwarded is an illustrative example of data stream processing. One can see the need for holistic aggregates in this scenario: quantiles provide simple statistical summary of the traffic carried by a link, heavy hitters nicely describe significant portion of the traffic on a link, and count distinct and count rare are indicators for normal activity vs activity under denial of service attack. Thus, monitoring holistic aggregates on IP traffic data streams is a compelling application.

Besides the straightforward use of holistic aggregates, our interactions with Gigascope users reveals the need for composition and grouping based on holistic aggregates. For example, a common network analysis query is one such as, "for every source IP and 1 hour interval, report the median, 95th percentile, and 99th percentile of the TCP round trip time". Similar grouping queries but with different holistic aggregates (such as `COUNT DISTINCT`) in

*Supported by NSF ITR 0220280 and NSF EIA 02-05116.
†Supported by NSF EIA 0087022, NSF ITR 0220280 and NSF EIA 02-05116.
[1]The term "holistic" was used in [23] to describe functions such as Median() and Mode() for which there is no constant bound on the size of the storage needed to exactly compute them.

SIGMOD 2004 June 13-18, 2004, Paris, France.

military monitoring applications can be found in [38, 4].

Despite the convergence of a compelling application like IP traffic data analysis and development of the principles of DSMSs, there has been little attention paid to the task of computing holistic aggregates in a DSMS for a real application such as network monitoring. Engineering holistic aggregation in a DSMS for real, very high speed data streams is a challenge. Query optimization even for simple holistic aggregate computation in a data stream is not well understood.

In this paper, we address this gap between principles and methods versus real needs in a practical application. We study the performance implications of using user-defined aggregate functions (UDAFs) for incorporating both selection-based and sketch-based algorithms for holistic aggregates into a data stream management system's query processing architecture. We use our streaming database, Gigascope [11, 12, 13], as a testbed and network traffic as our data source. Our contributions are as follows.

1. We identify key performance bottlenecks and tradeoffs, and propose novel techniques to make holistic UDAFs–selection-based as well as sketch-based ones—fast and space-efficient for use in high-speed data stream applications. Our techniques rely on judicious combination of low level aggregation on IP traffic data streams at the network router level with higher level composition, and adapting to the data characteristics.

2. We evaluate performance using generated and actual IP packet data on a "live" DSMS — Gigascope running at an IP router — focusing on approximating quantiles and heavy hitters. We derive more than half a dozen different implementation strategies for these holistic aggregates and test them with real as well as simulated data. The best of our current implementations can process streaming queries at up to OC48 speeds (2x 2.4Gbps), and is practical as an IP network data stream analysis engine in large ISPs.

Our implementation work is by necessity closely tied to the Gigascope architecture. However there are some general lessons to be learned:

1. Early data reduction is critical for complex querying of very high speed data streams. So we believe that a two-level architecture of query processing is highly suitable in the general context of a DSMS.

2. There is often a range of early data reduction strategies to choose from for processing approximate complex aggregates, including use of appropriate *partial* or subaggregation.

3. The most appropriate strategy depends on the streaming rate as well as the available processing resources; choosing the best strategy is a complex query optimization problem.

4. Approximate complex aggregates are quite effective, providing accuracy guarantees while vastly reducing the processing load.

5. Adaptive implementations of "heavy-weight" approximations such as sketches make them practical even in the presence of highly skewed data, so there is a potential in general purpose DSMSs too.

Our work sheds light on the broader intricacies of query optimization in a DSMS. Even for simple holistic UDAFs, the query optimizer has a large number of choices.

2. RELATED WORK

In the area of data stream algorithms, solutions are known for computing specific holistic aggregates. As we described earlier, the known algorithms can be divided into two categories: data-driven and universe-driven, with selection-based and sketch-based algorithms respectively. Many selection-based and sketch-based algorithms are known for holistic aggregates of our interest, i.e., finding quantiles [33, 24, 22] and heavy hitters [28, 32, 8]. These results are the most relevant to our work here. Selection- and sketch-based algorithms are known for other holistic aggregates such as correlated aggregates [18], and distinct counting [16, 19, 9]. But more generally, data stream algorithms are known for a variety of problems including wavelets [21], histograms [20, 25, 37], set expressions [17], complex queries [14], clustering [26], decision trees [15], etc. An overview of this area relevant to our study can be found at [5, 34], and in tutorials [17]. Many of these algorithms have typically been tested on synthetic datasets, or in some cases, on real IP network traces. We are not aware of any of these methods that have been directly incorporated into a "live" DSMS.

The area of DSMSs has seen extensive activity as well. A number of DSMSs have been proposed and built into prototypes including Aurora [6], Telegraph [7], Stream [35], Tribeca [36], and Gigascope [12]. Some of these DSMSs provide methods to do (random) sampling (e.g., [35]), and the collection of continuous queries for the military application that has been accumulated at Stanford presents examples using holistic aggregates. However, we do not know of any detailed study of the performance of holistic aggregates within these DSMSs. There has also been work on DSMSs for specific applications like streaming data from sensors [31] and financial applications [30]; again, no detailed performance study of holistic aggregates is known in these application at streaming data speeds we consider in this paper. The recently proposed ATLaS UDAF specification [38] supports, among other interesting features, streaming UDAFs. See Section 3.2 for more discussion.

3. INTEGRATING UDAFS IN GIGASCOPE

Gigascope [11, 12, 13] has a special architecture for handling very high speed data streams. In this section, we discuss some relevant aspects of the Gigascope architecture and how we integrate UDAFs into Gigascope.

3.1 Gigascope Architecture

Gigascope is designed for monitoring very high speed data streams using inexpensive processors. To accomplish this goal, Gigascope uses an architecture which is optimized for its particular applications.

First, Gigascope is a stream-only database — it does not support stored relations or continuous queries. This restriction greatly simplifies and streamlines the implementation. However, since there are no continuous queries (as implemented in, e.g., [35]) there are no explicit query evaluation windows, which are necessary to unblock operators such as aggregation and join. Instead, attributes in streams can be labeled with a "timestampness", such as monotone increasing. The query planner uses this information to determine how (and whether) a blocking operator can be unblocked. In an aggregation query, at least one of the group-by attributes must have a timestampness, say monotone increasing. When this attribute changes in value, all existing groups and their aggregates are flushed to the operator's output (similar to the *tumble* operator [6]). The values of the group-by attributes with timestampness thus define *epochs* in which aggregation occurs, with a flush at the end of each epoch.

Second, Gigascope has a two-level query architecture, where the low level is used for data reduction and the high level performs more complex processing. This approach is employed for keeping up with high streaming rates in a *controlled* way (i.e., guaranteed accuracy) in contrast to some existing DSMSs which employ load shedding [6, 4]. High speed data streams from a Network Interface Card (NIC), are placed in a large ring buffer. These streams are called *source streams* to distinguish them from data streams created by queries. The data volume of these source streams are far too large to provide a copy to each query on the stream. Instead, the queries are shipped to the streams. If a query Q is to be executed over source stream S, then Gigascope creates a subquery q which directly accesses S, and transforms Q into Q' which is executed over the output from q. In general, one subquery is created for every table variable which aliases a source stream, for every query in the current query set. The subqueries read directly from the ring buffer. Since their output streams are much smaller than the source stream, the two-level architecture greatly reduces the amount of copying (simple queries can be evaluated directly on a source stream).

The subqueries (which are called "LFTAs", or low-level queries, in Gigascope) are intended to be fast, lightweight data reduction queries. By deferring expensive processing (expensive functions and predicates, joins, large scale aggregation), the high volume source stream is quickly processed, minimizing buffer requirements. The expensive processing is performed on the output of the low level queries, but this data volume is smaller and easily buffered. Depending on the capabilities of the NIC, we can push some or all of the subquery processing into the NIC itself. In the testbed described in Section 5, the NIC used for the synthetic data is capable of processing a type of projection operator, while the NIC used for the live data has no special processing capabilities. In general, the most appropriate strategy depends on the streaming rate as well as the available processing resources. Choosing the best strategy is a complex query optimization problem, the goal of which is to maximize the amount of data reduction without overburdening the low level processor and thus causing packet drops.

To ensure that aggregation is fast, the low-level aggregation operator uses a fixed-size hash table for maintaining the different groups of a `GROUP BY`. If a hash table collision occurs, the existing group and its aggregate are ejected (as a tuple), and the new group uses the old group's slot. That is, Gigascope computes a partial aggregate at the low level which is completed at a higher level. The query decomposition of an aggregate query Q is similar to that of subaggregates and superaggregates in data cube computations [23].

Third, Gigascope creates queries by generating C and C++ code, which is compiled and linked into executable queries. To integrate a UDAF into Gigascope, we add the UDAF functions to the Gigascope library and augment Gigascope query generation to properly handle references to UDAFs (Section 3.2).

3.2 UDAF specification

As discussed in Section 3.1, incorporating a UDAF into Gigascope is a matter of incorporating the UDAF calls into the Gigascope library, and providing the query planner with the specification of the UDAF. We modified Gigascope so that it can understand UDAF specifications and make calls to the UDAF functions at the appropriate places.

A UDAF is commonly composed of three functions [27]: an INITIALIZE function, which initializes the state of a *scratchpad* space, an ITERATE function, which adds a value to the state of the UDAF, and a TERMINATE function, which releases UDAF resources and returns a value. In order to support multiple return values from the same UDAF computation (discussed below), we split the TERMINATE function into an OUTPUT function and a DESTROY function. Low-level Gigascope queries require an additional function, FLUSHME. Recall that low-level queries are simple, fast queries for data reduction. Holistic aggregate data structures might be large and/or require occasional expensive restructuring. The FLUSHME call is used by a low-level UDAF to indicate that it is "full" and should be flushed to a high-level query to complete its processing. Therefore we can use a small and fast data structure at the low-level; partial processing will be completed at a higher level.

We inform Gigascope of the UDAFs that might occur in a query by providing UDAF declarations. A UDAF declaration must include the UDAF name, its return type, the types of its parameters, and its scratchpad type. For example,

```
int UDAF char(36) approx_median(int)
```

declares that approx_median is a UDAF which accepts an integer, uses 36 bytes of scratchpad storage, and returns an integer.

In order to apply approx_median to values derived from a source data stream, we must also specify the subaggregate (used in the low-level query) and the superaggregate (used in the high-level query). For example,

```
int UDAF char(36) [low_approx_med, high_approx_med]
                  approx_median(int)
vstring UDAF char(2800) low_approx_med(int)
int UDAF char(36) high_approx_med(vstring)
```

declares that low_approx_med is the subaggregate and that high_approx_med is the superaggregate of approx_median. If a query Q references approx_median on a value derived from a source stream, it is transformed into q_1 which references low_approx_med and Q' which references high_approx_med on the low_approx_med of q_1. Note that the return value of low_approx_med, a variable-length string which represents the UDAFs state, is the value accepted by high_approx_med.

In many queries, we want the UDAF to return many values. For example, a query might ask for the median, the 95th percentile, and the 99th percentile values of packet round trip times. Computing the UDAF three times is inefficient, instead we use *extractor functions* to declare that the UDAF needs to be computed only once. An extractor function is just a macro for specifying that a function is to be called on a UDAF. For example,

```
int EXTR percentile_fcn approx_quantile percentile(int; int)
```

is a macro which transforms `percentile(len,95)` into `percentile_fcn(approx_quantile(len),95)`. The duplicate references to approx_quantile are now easily recognized. The approx_quantile aggregate returns a searchable data structure when the OUTPUT function is called. The percentile_fcn function performs a search on this structure. The aggregate's DESTOY call will release the resources used by the return value, if necessary.

We designed our UDAF specification to be an easily implemented extension to the conventional UDAF specification, but which would support aggregate query decomposition and multiple return values. A more sophisticated specification, ATLaS has been recently proposed [38]. One of the benefits of the ATLaS specification is that the UDAF functions are defined in SQL within the declaration. While this property is highly desirable for general purpose DBMS extensibility, for several reasons we felt that the traditional method of C-language UDAF function definitions was more appropriate for our purposes. For one, it is easier to implement and makes fewer demands on the query optimizer. For another, Gigascope is a specialized system. The UDAFs are likely to be written by experts

for whom the highest possible performance is the critical issue and who are more familiar with C than with SQL. The easy extensibility described in [38] is a lesser concern.

In the following query, which is similar to those used for the experiments, the median packet length is computed for every source IP address and every one minute interval:

```
SELECT tb, sourceIP, median(length) FROM UDP
GROUP BY time/60 as tb, sourceIP
```

Since UDP is a source data stream, this query will be broken into a subaggregate and a superaggregate query by the query planner.

4. STREAMING ALGORITHMS

4.1 Selection-based Quantiles

Greenwald and Khanna [24] proposed a novel data structure, the *quantile summary*, that effectively maintains lower- and upper-bounds on the ranks ($r_{min}(v)$ and $r_{max}(v)$, respectively) for each value v from the input stream. After n input values, the data structure $S(n)$ consists of an ordered sequence of tuples which correspond to a subset of the observations from the input stream; initially, $S(n)$ is empty. Each tuple $t_i = (v_i, g_i, \Delta_i)$ consists of three components: (i) a value v_i that corresponds to an element in the data stream; (ii) the value g_i equals $r_{min}(v_i) - r_{min}(v_{i-1})$; and (iii) Δ_i equals $r_{max}(v_i) - r_{min}(v_i)$. By ensuring that the summary structure $S(n)$ satisfies the property $\max_i(g_i + \Delta_i) \leq \lfloor 2\epsilon n \rfloor$, any ϕ-quantile query can be answered to within ϵn precision in rank. To achieve this, the input stream is conceptually divided into buckets of width $w = \lceil \frac{1}{\epsilon} \rceil$. Each value v from the current bucket is inserted into S between tuples t_{i-1} and t_i, where $v_{i-1} < v \leq v_i$, with values $g = 1$ and $\Delta = g_i + \Delta_i - 1$. Periodically, the space is compressed by merging adjacent pairs of tuples t_i and t_{i+1} whenever $(g_i + g_{i+1} + \Delta_{i+1}) \leq \lfloor 2\epsilon n \rfloor$. Their analysis of this algorithm showed a space bound of $O(\frac{1}{\epsilon} \log(\epsilon n))$.

Implementation in Gigascope. We implemented three quantile UDAF variants based on quantile summaries, which make use of the Gigascope processing hierarchy. These algorithms span a range from simple to complex preprocessing at the low level query, and divide up the work between the low level query (LLQ) and high level query (HLQ) in different amounts as follows.

Algorithm 1 (LLQ-lite). This algorithm does minimal work at the LLQ, which is used for buffering the incoming tuples in an array (ordered by arrival time) before being sent to the HLQ for batch processing.

Algorithm 2 (LLQ-heavy). This algorithm maintains a linked list of samples ordered by item values. After insertion, it reduces the size by sweeping through it in a full compress phase. If the storage space at the LLQ becomes full, it outputs the data structure (to be sent to the HLQ using a FLUSHME call), discards the space, and starts over with a new data structure.

Algorithm 3 (LLQ-medium). This algorithm attempts to find a "happy medium" between Algorithms 1 and 2. It trades off processing at the LLQ for more at the HLQ but, whereas the processing time for Algorithm 2 is linear in the number of tuples, this strategy does updates in logarithmic expected time by maintaining a skiplist directory of the tuples ordered by item values. Also, it *partially* compresses after each insertion by performing a local probe at where the new element is inserted as well as a probe at a random location in the list; this is done in constant time.

```
Merge(S_ℓ, S_h)
/* S_ℓ(m) = ⟨(v_i, g_i, Δ_i)⟩ is summary at LLQ of size M */
/* S_h(n) = ⟨(v_j, g_j, Δ_j)⟩ is summary at HLQ of size N */
/* The result is S(m+n) of size M+N */
01   i := j := 1; S := ∅;
02   S_ℓ[M+1] := S_h[N+1] = (∞, 1, 0);
03   for k := 1 to (M+N) do
04      if (v_i < v_j)
05         S[k] := (v_i, g_i, Δ_i + g_j + Δ_j − 1);
06         i+ = 1;
07      else if (v_j < v_i)
08         S[k] := (v_j, g_j, Δ_j + g_i + Δ_i − 1);
09         j+ = 1;
10      else if (v_i = v_j)
11         S[k] := (v_i, g_i + g_j, Δ_i + Δ_j);
12         i+ = 1; j+ = 1;
13   S_h := S;
```

Figure 1: Merge Algorithm

We now describe how the output $S_\ell(n)$ from the LLQ is processed at the HLQ, which maintains a quantile summary $S_h = S_h(n)$ of N tuples. For Algorithm 1, the output is a simple array of values, each of which is inserted into S_h. For Algorithms 2 and 3, the LLQ output is a quantile summary $S_\ell = S_\ell(m)$, of M tuples, and the update procedure is more involved. First, the algorithm compresses tuples in S_ℓ that could have been compressed at the LLQ, if any. Then S_ℓ and S_h are merged into a single summary $S = S(m+n)$, of size $M + N$, during which their Δ-values (and thus maximum ranks) are adjusted; the pseudocode for this is given in Figure 1. After the merge, adjacent pairs of tuples t_k and t_{k+1} in S for which $(g_k + g_{k+1} + \Delta_{k+1}) \leq \lfloor 2\epsilon(m+n) \rfloor$ are compressed to reduce space. The resulting quantile summary S is guaranteed to report quantiles with at most $\epsilon(m+n)$ rank error. Although it cannot guarantee a space bound of $O(\frac{1}{\epsilon} \log \epsilon(m+n))$, due to worst-case scenarios when merging at the HLQ, our experiments indicate that this bound tends to hold in practice.

4.2 Sketch-based Heavy Hitters

Many different sketch methods have been proposed, for computing frequency moments [3], count distinct queries [16], join size estimation [2, 14] and heavy hitters [8, 10]. We will focus on computing the heavy hitters on streams of values and updated counts, e.g., finding large flows grouped by source IP address with counts coming from packet sizes. The "count-min sketch" method described in [10] gives a probabilistic approach to approximating the count of any item, with an error proportional to the sum of the counts of items. This can easily be incorporated into a scheme to find the heavy hitters (all items whose count exceed a threshold fraction of the total count) with a simple top-down search procedure. For implementation in Gigascope, we will fix the parameters of the sketch, which determine the size of the data structure, and test how to divide the processing work between the low-level and the high-level.

Implementation in Gigascope. Each sketch is implemented as an array of counts, $sk[1 \ldots d, 1 \ldots w]$. There are d different hash functions $h_1 \ldots h_d$ which map item ids onto $\{1 \ldots w\}$. Each new packet will be interpreted as an update to the sketch, in a way determined by the user query: for example, a query on Heavy Hitters for source IP addresses based on packet size means that each packet will be interpreted as an update with id =Source IP, and val =packet size (in bytes). With each update, (id, val), the sketch is updated by $sk[i, h_i(id)]$+=val for $i = 1 \ldots d$. To estimate the sum of all values for one id, we take $\min_i sk[i, h_i(id)]$. The error in the estimate is proportional to $\sum val/w$, and the probability of

higher error is proportional to 2^{-d}. Two sketches with the same h_i, w, and d can be summed, entry-wise, to make the sketch of the sum of the streams. This property is needed by some of our implementations.

To find items with the highest counts, we keep g sketches $sk(1)$, $\ldots, sk(g)$ of items at g different levels of granularity (e.g., a sketch for ids, sketch for $\lfloor id/2 \rfloor$, for $\lfloor id/4 \rfloor$ etc). Then the search for values greater than $\sum val/100$ proceeds in a similar way to a binary search. The space cost is proportional to $g * w * d$ and the update cost scales with $g * d$. There are tradeoffs between settings of the sketch parameters: increasing w gives better accuracy, but uses more space; increasing d gives fewer errors, but at the cost of both update time and space; smaller g gives faster updates but can cause more errors and take more time to extract the results. We set $w = 256$ giving an expected error factor of less than 1% in estimating counts (choosing a power of two also makes the hash functions more efficient to compute), and set $d = 2$. We fix $g = 3$ by keeping sketches of the full 32-bit items as well as 24- and 16-bit prefixes of items. We also kept exact counts for the 256 8-bit prefixes, giving a total sketch size of 7KB. These parameter settings were made on the basis of experimentation. In future work we plan to investigate the impact of other parameter settings, but such a comparison is beyond the scope of this paper.

Low-level Query Strategies

At the low level, we considered four methods, in order of increasing complexity.

1. Buffer. The Buffer strategy simply keeps a 1KB array a of (id, val) pairs as they arrive and flushes the buffer after 128 values. If the buffer is flushed when it is partially full, then only the occupied prefix of the array is passed up to the high-level.

2. Hash. The Hash strategy also uses an array a of (id, val) pairs, but uses it as a hash table. When an update (id, val) arrives, we test if $a[hash(id)] = id$, and if so add val to the current count. If the slot is empty, then we put id in the slot, else we search $a[hash(id) + 1], a[hash(id) + 2] \ldots$ for id or an empty slot.[2] This can aggregate counts, and is expected to show improvement over buffering for skewed data sources: by aggregating all insertions of a particular item, a single update is required at the high level, instead of many.

3. Compress. A disadvantage of the Hash strategy is that if it is flushed while being sparsely populated, then the whole table is copied up to the high-level, whereas only a small amount needs to be sent. Compress augments the hashing approach: when the structure is flushed, if the count of non-zero entries is low then the table is compacted by moving each entry to the first available free slot, and only the populated prefix of a is copied up to the high level. This does not affect the high level processing, and reduces the memory transfer.

4. Low Sketch. The most intensive low-level strategy computes a sketch sk' at the low level. This requires the 7KB of space being allocated for each group, which may be a high overhead in terms of space for large numbers of groups, and a bottleneck for transfers, especially on epoch boundaries.

A design feature for these experiments is that the output of the first three strategies above are interchangeable, so we can compare the effect of different choices at the low level with different choices at the high level.

[2]If the search exceeds 8 consecutive locations, then the routine requests a flush.

High Level Strategies

At the high level, we considered three different ways to use the sketch routines.

A. Low Sketch. This strategy is the partner routine for the Low Sketch at the low-level, and keeps a sketch, sk at the high-level. When the low-level is flushed and a sketch, sk' is received from the low-level, we set $sk(i)[j,k]$+=$sk'(i)[j,k]$ for all i, j, k: this uses the property of summability of sketches.

B. Direct Sketch. The Direct Sketch strategy accepts an array $a[1 \ldots p]$ from the low-level, and updates each sketch with each (id, val) pair as described above.

C. Adaptive Sketch. For skewed or sparse data, the previous methods automatically allocate the space for sketching and return approximate results when it would be more space efficient to keep the exact data and return exact answers for some groups. The adaptive approach is designed to smoothly adapt to the input distribution. Initially, it keeps exact results as a list l of (id, val) pairs, and when new updates (id', val') are received from the low level, the list is searched for (id', val). If found, we update val+=val'; if not found, we append (id', val') to the list. If the length of the list exceeds a set length, then a sketch is allocated, and the list is used to populate the sketch. The default for this threshold was set to 64 distinct values, and in experiments we compared to values of 128 and 256. For the skewed distributions seen in real data streams, we may observe many thousands or even millions of observed packets before this many *distinct* values are seen (see next section for quantitative results).

5. EXPERIMENTAL ENVIRONMENT

To evaluate the performance of the approximate quantile and heavy hitter algorithms, we modified Gigascope to accept UDAFs, as described in Section 3.2, and incorporated the algorithms for the streaming algorithms into the Gigascope library. For performance testing, we used two data sources.

The first data source is an Agilent Technologies RouterTester 5.0 Gigeth traffic generator [1]. Using it, we can generate about 1Gbit/sec of traffic on each of two Gigeth links. The traffic generator is not a sophisticated source of randomness. We could only vary the source IP address of the packet and the packet length, both independently and uniformly random. The average packet length is always 782 bytes, and when both Gigeth channels are driven at the maximum rate they produce 310,000 packets/second.

We monitored the generated stream using a modern but inexpensive server comprised of two 2.8 Ghz pentium processors and 4 Gbytes of memory. In this system, we configured Gigascope to report the number of packets dropped before they could be presented to the low-level queries. In addition, we are alerted when packets sent from the low-level queries to the high-level queries are dropped, but we could not collect precise statistics. For these experiments, aggregates are collected over 10 second intervals. When measuring the CPU load, we collect the CPU time used by the processes over a 100 second interval.

The traffic generator provides a controlled environment for measuring CPU overhead, but it does not represent a realistic data source. For an alternative data source, we monitor the *span port* of the router which connects our institution to the internet, via a 100 Mbit/sec link (a span port mirrors all traffic for monitoring purposes). We monitored this stream using an older single-cpu 733 Mhz pentium with 128 Mbytes of RAM, which had been previously set up by the network administrators. The experiments used the query in Section 3.2 (varying the UDAF and using 5 minute intervals for the

accuracy experiments). Each experiment ran for 1 hour.

Even though all experiments ran during normal business hours, the traffic on the link varied considerably from experiment to experiment, and even during experiments. Figure 2 shows the per-minute traffic volumes and number of groups for a high-volume run. A typical low-volume run averaged about 400,000 packets per minute (about 7,000 per second), while the high-volume run averaged about 1,110,000 packets per minute (about 18,000 per second). However the average number of groups per minute was 640 and 1260, respectively, exhibiting considerably less variation than the number of packets. As is evident from these statistics and from the chart, the nature of the traffic changes very rapidly.

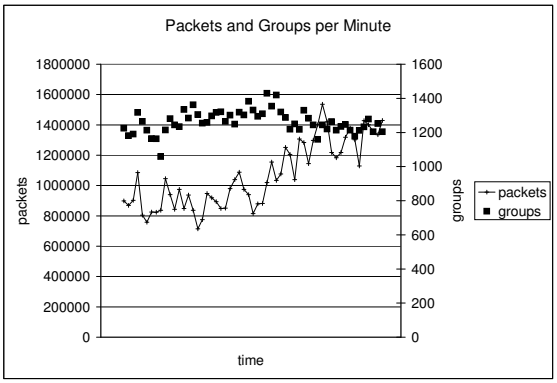


Figure 2: High traffic volume.

The distribution of tuples to groups is extremely skewed. A sample distribution of packets per group plotted on a log-log scale shows a straight line, indicating a power law distribution. This property is present in all of our data samples. These skewed distributions have important implications for implementing fast UDAFs. Most of the groups have only a few tuples, and can be represented exactly by small, simple, and fast data structures. However, most of the packets are processed in groups with a very large number of packets. This property can be clearly seen in Figure 3, which shows the cumulative distribution of the number of groups and the number of packets against the packets per group. Using 1000 packets per group as the boundary between "small" and "large", only 3% of the groups are large, but they process 92% of the packets. Other boundaries produce similar results.

6. LOW-LEVEL QUERY PERFORMANCE

A critical optimization in Gigascope is the splitting of aggregation queries into low-level subaggregation and high-level superaggregation queries. Since the operation of the subaggregate query has a very large impact on performance, in this section we examine the subaggregation algorithms and their performance in detail.

Subaggregation queries are evaluated using a fixed-size buffer to store groups and aggregate data (UDAFs for subaggregation use fixed size scratchpad space in the group tuple, as malloc is deprecated for low-level queries). Because there are a limited number of groups which can be in memory, the subaggregate query acts as an "aggregation cache".

One mechanism that we examined is the aggregate cache replacement policy. Early versions of Gigascope had used an LRU

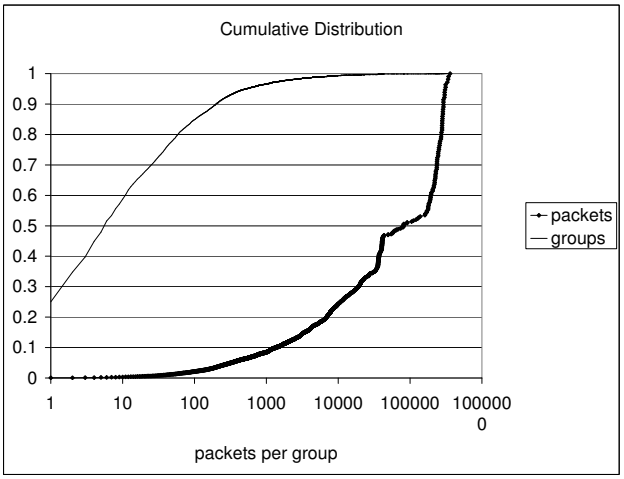


Figure 3: Cumulative distribution of packets and groups.

replacement policy. We found that using LRU caused too much overhead, so we changed the replacement policy to direct mapped. In this policy, groups are mapped to a hash table without chaining. If there is a collision in the hash table, the old group is flushed to make space for the new group.

We were not able to recover the LRU replacement policy to empirically determine the actual overhead of LRU. However we did notice that the direct mapped policy is likely to cause an excessive number of cache invalidations due to hash table collisions. We implemented a simple modification to the direct mapped policy, which we term "second chance". If on the first try there is a hash collision, the replacement policy will rehash. If the second try also results in a collision, the group in the first hash position is ejected.

We ran the following experiment using the traffic generator. We created a simple aggregation query (collecting a COUNT, MIN and MAX). We varied the number of groups (source IP addresses) in the packet stream and decreased the number of cache slots in the low-level query until we found the minimum cache size before packet loss occurred. The results are in Table 1.

# active groups	direct mapped	second chance
2000	1900	1300
5000	4300	3300
10000	7300	7300
20000	64000	19000
40000	410000	44000

Table 1: Minimum cache size before packet loss.

The second chance policy is much more effective than the direct mapped policy, especially when the working set of the source stream becomes large. As the policy is simple, fast, and effective, it is a good substitute for LRU. All of the remaining experiments in this paper were run using the second chance replacement policy, and second chance replaced direct mapped in the production version of Gigascope.

As discussed in Section 3.1, aggregation in Gigascope divides time into a sequence of epochs. At the end of each epoch, all groups are closed, converted to tuples (subject to a HAVING clause) and flushed to the output. This point tends to be a performance bottleneck, as a potentially large number of tuples are created and flushed

to an output stream (involving a lot of memory copies). For holistic UDAFs, the bottleneck becomes worse because of the processing to finalize the aggregate (in the OUTPUT function) can be expensive.

We had noticed this earlier, and implemented a lazy cache flush policy for low-level queries (a similar policy was implemented for high level queries, but an examination of it is beyond our scope). When the epoch changes, all full hash table entries are labeled old. When a tuple arrives in the new epoch, an old full hash table entry is flushed (if any). If the new group collides with an old hash table entry, the old group is flushed. If the new group collides with a new hash table entry, all of the old entries must be flushed immediately, to preserve attribute timestampness in the output stream.

We were able to evaluate the effectiveness of the lazy flush policy because it is a simple matter to disable it (enforce eager flush). We reused the aggregation query from the replacement policy experiment. We then subscribed to an increasing number of these queries (ensuring that the common subqueries were not shared) and measured the packet loss rate at the low-level queries. We varied the number of groups, measured the loss rate and CPU utilization, and report these values for the smallest number of groups for which packets were dropped. The results are in Table 2.

# groups	# queries	loss (eager)	loss (lazy)	cpu util
50000	2	1.8	.1	37
25000	3	.4	0	51
12000	5	.5	.3	80

Table 2: Packet loss rate for eager and lazy flush

The lazy flush policy provides a small but significant reduction in the packet loss rate, especially when the number of groups is large. The improvement can make the difference between an unacceptable and an acceptable packet loss rate. Since holistic UDAFs tend to have large state and expensive OUTPUT functions, these effects are likely to occur with a smaller number of groups.

We note that the CPU utilization might be low (as low as 37%) when packet drops occur. This fact indicates that the group flush at the end of the epoch is a bottleneck. The solution lies more in scheduling and buffering than in reducing CPU costs. The schedulability of the queries improves significantly as the number of groups decreases. If we wish to perform controlled load shedding to respond to handle overload conditions, it is better to sample groups rather than packets.

7. HOLISTIC UDAF PERFORMANCE

We evaluated different UDAF implementations of quantiles and heavy hitters with respect to performance, space usage and accuracy. The synthetic traffic generator was used for many experiments to allow better control over the data characteristics. This generator created uniformly distributed values for the grouping attribute (into either 100, 1K, or 10K groups) and, within each group, the number of distinct values for the attribute to be aggregated was varied (10, 120 or 1436). For the remainder of the experiments, we used the live TCP traffic data.

The goal is to run as many queries as possible on as high a data rate as possible without dropping packets. Our evaluation of the algorithms is based on their schedulability with varying data characteristics and server architectures.

7.1 Selection-based Quantiles

Here we report experimental results using the three algorithms described in Section 4.1. We considered several parameters such as the scratchpad size at the low-level query (LLQ) (small, medium or large).

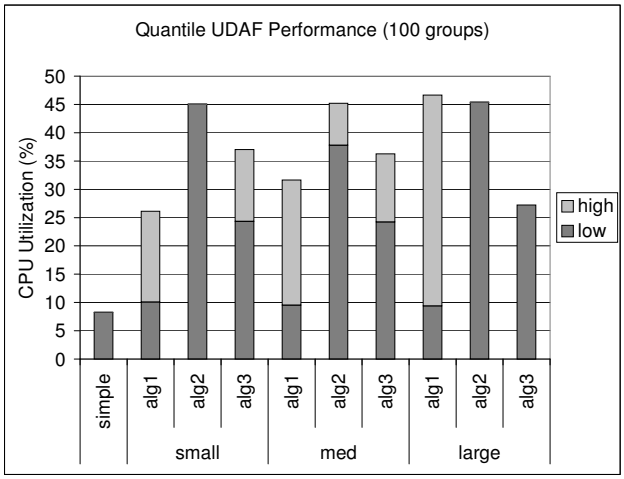
Performance Results. Using data from the traffic generator, the performance differences between the methods are most pronounced in the case of 100 groups (which is when the number of tuples per group is largest). Here the default high-level query algorithm (HLQ-only) drops too many packets to yield meaningful numbers. This was true in all our experiments. Hence, we do not report their results in this section. The performance of the remaining algorithms, including the "null" UDAF as a baseline, are summarized in Figure 4(a).

At the LLQ, Algorithm 1 was clearly the fastest in all cases, regardless of the scratchpad size, and was only marginally slower than the "null" UDAF. Algorithm 2 was the slowest, almost twice as slow as Algorithm 3, for all scratchpad sizes. Recall from Section 4.1 that Algorithm 1 is the most lightweight at the low level, followed by Algorithm 3, then Algorithm 2. Hence, this ranking in performance is to be expected. This ordering was reversed at the HLQ in all cases, with Algorithm 1 giving the slowest performance followed by Algorithm 3, then Algorithm 2. Hence, we observe that the choice of strategy at the low level (inversely) impacts the performance at the high level. This negative correlation is due to the data reduction from the quantile summaries employed at the LLQ by Algorithms 2 and 3: the more reduction, the fewer (and smaller) the transfers from the low to the high level.

However, this trade-off may not be desirable when a large increase in processing cost at the LLQ buys only a modest decrease at the HLQ. In a shared-processor system (which was the environment for this experiment), the low-level queries are executed by the same processor(s) as the high-level queries, and are therefore a bottleneck. Hence, Figure 4(a) shows that the additional cost at the LLQ could only be justified for Algorithm 3 with the large scratchpad size; in all the other cases, the savings at the HLQ was more than offset by the extra work at the LLQ.

Figure 4(b) summarizes the performance of the three algorithms using the traffic generator with 10K groups. Once again, we observe the same ordering among the algorithms with respect to performance at the LLQ, to varying degrees. In this instance, we see the payoffs of extra processing at the LLQ for the medium scratchpad size. Due to the larger number of groups (10K here versus 100 in the previous experiment), there are more transfers, so data reduction can have a more significant impact. However, with a large scratchpad size, the extra work at the LLQ for Algorithms 2 and 3 caused the processor to drop packets. In fact, for Algorithm 2 there was so much loss that measurements could not be computed. While Algorithm 2 can be the best choice under certain conditions, it is also the riskiest. Algorithm 3 can provide benefits, with large scratchpad size and relatively small number of groups. Algorithm 1 was the safest choice overall with respect to packet loss at the low level, but achieves the least amount of data reduction and is thus the least scalable at the high level.

The traffic characteristics using the live TCP data varied so much between experiments that comparisons are difficult. The algorithms exhibited similar behavior to that with the synthetic data. Algorithm 1 had the best performance at the LLQ (followed by Alg 3, then Alg 2) and the worst performance at the HLQ; the per-tuple processing time is summarized in Table 3. Interestingly, the trade-off for extra LLQ processing was beneficial here for Algorithm 2 as it had the lowest total per-packet cost, making it the most scalable in a single-CPU system. This data is highly skewed, even more



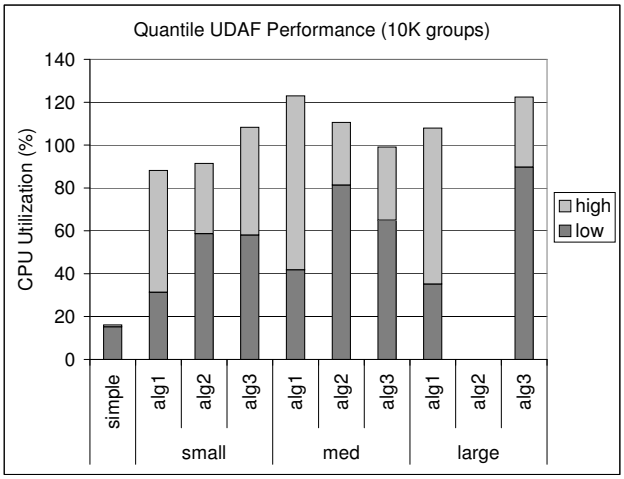


Figure 4: Performance of quantile UDAF algorithms on data from traffic generator.

	LLQ	*HLQ*	*LLQ+HLQ*
Alg1	1.70	13.9	15.6
Alg2	11.8	.129	11.9
Alg3	6.67	9.87	16.5

Table 3: Average processing time per tuple (μsec) using TCP data. Four UDAFs in query.

so than the data from the traffic generator. Therefore, a very small fraction of the groups dominate the transfer costs. For these groups, it pays to expend the effort to aggregate using quantile summaries so as to delay transfers because these groups are active and unlikely to be flushed before becoming full.

Space Usage. As we mentioned in Section 4.2, Algorithms 2 and 3 do not provide as tight a worst-case space bound for the HLQ quantile summary as Algorithm 1 does, due to merging. However, in all our experiments the space usage was comparable for the three algorithms and far less pessimistic than the worst case. In fact, when we looked at this ratio on a per-group basis, the size of the quantile summary appeared to be independent of the number of stream tuples in the group. This is consistent with the observation in [24] that, for values arriving in a random order, the space depends only on ϵ and not on the number of stream tuples, despite the logarithmic dependence in the worst-case bound.

7.2 Sketch-based Heavy Hitters

We ran a series of experiments to determine the cost of using sketches, under the same experimental setups as for the selection based methods. We looked at the effect of combinations of the different strategies proposed above, and the effect of higher system load, modeled by varying the number of active groups. We also looked at the cost in terms of space, and the accuracy attainable using these approaches. These experiments also show that sketches can be practical on network streams at network line speeds.

Performance Results. The first set of results are shown in Figure 5, which shows an experiment on traffic generated by the traffic generator described above. We looked at all the combinations of low-level and high-level strategies possible on the dual processor

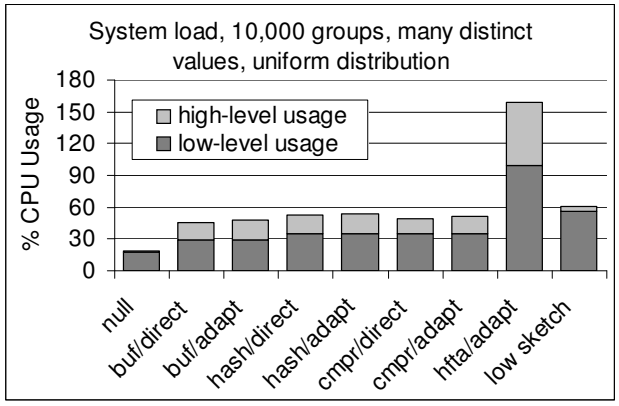


Figure 5: Sketch performance on uniform generated traffic

testbed. The experiment shown in Figure 5 show the effect of a large number of groups, and a large number of values. We compare to the cost of the "null" UDAF, which merely computes the sum of values seen, to show that most strategies at the low-level do not add much to the cost of using Gigascope.

The first conclusion of these experiments is that the two extremes both give bad results: the default approach of running the query solely at the high level or solely at the low level is a bad solution. We observed that trying to run the query completely in the high level, or using the low sketch strategy both caused a significant number of packet drops, whereas the intermediate approaches have no drops at all. Because the high-level approach is so costly at both the high and low level (causing 100% CPU usage for the processor at the low-level) and causes so many packet drops, we do not explore it further. The low sketch strategy also caused many packet drops and put the heaviest strain on the low level system, in this and all other experiments. Since the low-level query system tends to be a bottleneck, this behavior is undesirable.

This leads us to conclude that the best performance will come from picking a combination of buffer, hash, or compress at the low level, and direct or adaptive sketching on the high level. We concentrate on these methods for the remainder of the analysis. The

data generated for the plot in Figure 5 had a large number of unique values, meaning that it would likely cause the adaptive sketching approach to create a sketch very quickly, and indeed it can be seen that the high-level costs of all strategies are similar, with a slight disadvantage to the adaptive approach, since it must create and populate a sketch part-way through the test. There is also little difference between the low level strategies, since all keep a 1KB buffer, which on this traffic distribution will all fill at approximately the same rate: the buffer approach has a slight advantage here since it has lower processing costs to get the same effect as the compress and hash methods.

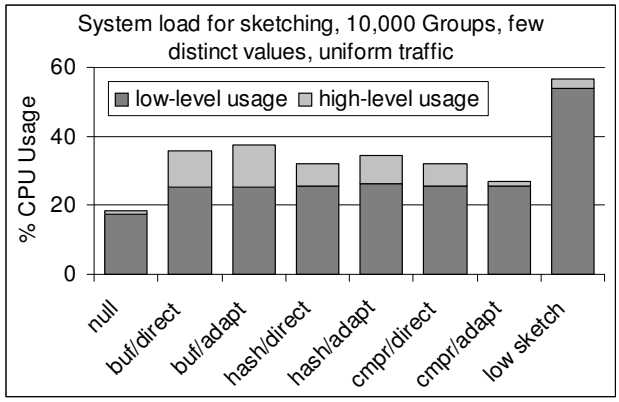


Figure 6: Sketch performance on skewed generated traffic

In practice, we have already noted that network data displays a strongly skewed distribution, and it is on such data (see Figure 6), that one can distinguish between the high level methods. Although there is little difference at the low level for the different strategies, the choice of strategy at the low level does have an impact on the cost at the high level: buffering causes more work than hashing, which is more expensive than compressing the hash table. Interestingly, the best performance is achieved by the adaptive approach combined with compressing the hash table, where the cost at the high level is negligible. For other strategies, the adaptive approach is a little more expensive than the direct approach, but this is to be weighed against the potential space savings and accuracy improvements from the adaptive approach.

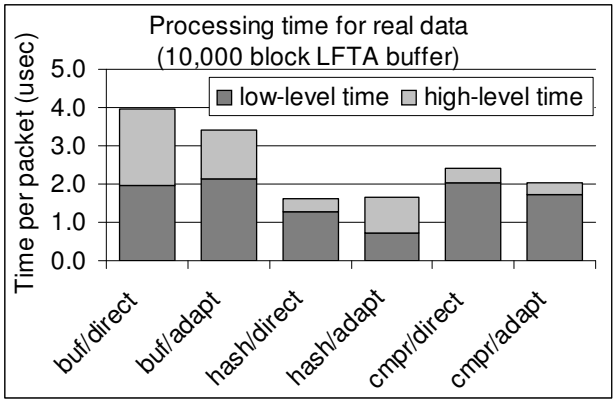


Figure 7: Average processing time on real traffic stream (four UDAFs)

This experience leads us to focus in on these six strategies (pairing the three low-level strategies with the two high-level strategies), and experiment on real data. The timing results shown in Figure 7 show again that the hash and compress strategies are to be preferred. Note that it is harder to cross-compare between results in these experiments, since we cannot "record and replay" real traffic streams. On the same data, the cost of hash/direct and hash/adaptive should be the same at the low level, but fluctuations in the stream we were monitoring affected the distributions. Nevertheless, we can conclude that there are apparent benefits to the adaptive strategy, where the total cost is less than the corresponding direct version, and that trying to do aggregation work at the low level can bring significant reductions in cost at the high level. Overall, the combination of strategies that works best at both levels is when the right balance is struck between amount of work done, and the cost of doing that work. We recorded no packet drops with these methods, even though they were running on a less powerful system. On a more modern processor, the times scale to a cost of around a hundred nanoseconds per packet.

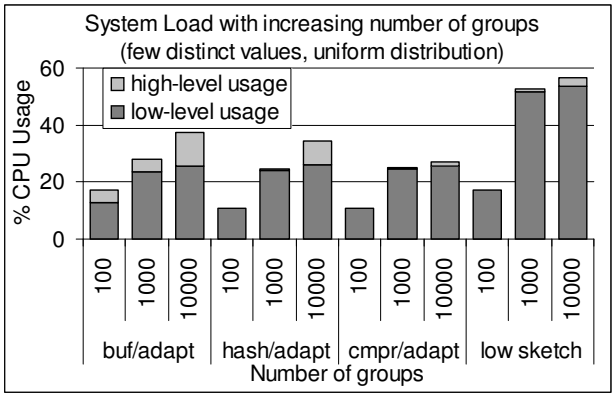


Figure 8: CPU usage with number of groups

Effect of Number of Groups. Figure 8 shows how the cost of the methods scale as the number of groups increases. We show only the low sketch and adaptive strategy: on this data set, we found that for a given low-level strategy, the same cost was virtually identical for the direct and adaptive strategies, so we plot only the latter for clarity. We see that the buffering and hashing approach seem to scale logarithmically with the number of groups. The compress strategy is almost the same as the similar hash strategy until the number of groups becomes large, at which point there are clear advantages to compressing. Once again, attempting to push too much computation down to the low level, in the form of the low sketch approach, is uniformly the most costly, by a significant margin. But doing some processing at the low level, by hashing, or hashing and compressing, shows moderate but noticeable improvements.

Parameters of the Adaptive Sketch Method. We experimented with when to switch from keeping exact counts of items to making a sketch in the adaptive strategy. In Figure 9, we again see a clear correlation with log of the number of groups for sketching after 64 (the default) and 128 distinct values. For 256 values, the cost seems to grow more quickly, reflecting the additional cost of converting the list of exact values to a sketch once the threshold of 256 values is exceeded. On the other hand, there are definite space advantages of choosing a larger value, since the size of a sketch (7KB) is much higher than the cost of each item (8 bytes for each (item, count)

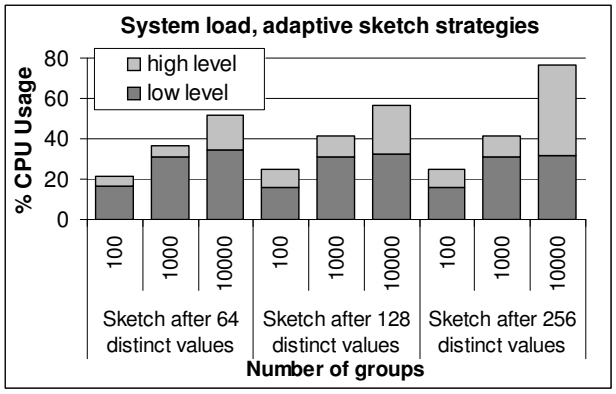


Figure 9: Varying when to sketch for adaptive strategy

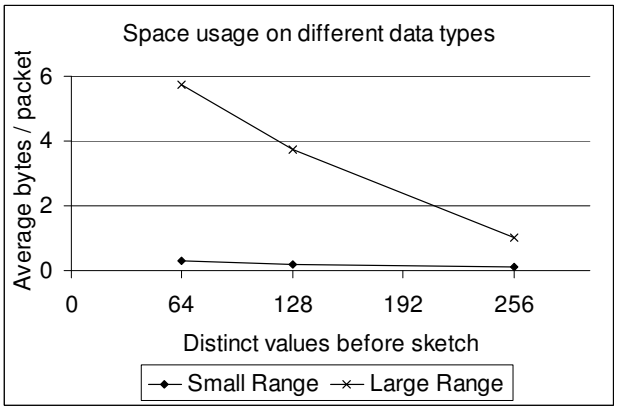


Figure 10: Space cost for adaptive strategy

pair). This is shown in Figure 10, where increasing the threshold before sketching reduces the average number of bytes per packet processed. This is especially clear when the data being analyzed is drawn from a larger range (say, IP addresses rather than port numbers): here, there is more chance of creating a sketch given a lower threshold, and hence the average cost is higher the more quickly a sketch is made.

Accuracy Results. We must also ensure that the results of keeping sketches actually answer the queries with reasonable accuracy. For our heavy hitters queries, we were able to compute the exact answers to the queries when using the traffic generator, and compared them to the output of the sketching strategies. We queried the system to find the top five heavy hitters in various distributions, and compared the exact top five with the top five found using sketching. Then we computed the proportion of approximate answers that were (a) correct, e.g., the top approximate heavy hitter was the true top heavy hitter; (b) off by one, e.g., the third heavy hitter was ranked second or fourth by the sketch method; (c) off by more than one, so a heavy hitter was returned by the sketch method but its rank was further off; or (d) missed, e.g., the fifth heavy hitter was not returned by the sketch method. The results are shown for the adaptive strategy and direct strategy in Figure 11 (note that the low-level strategy is irrelevant here, since the sketch computed is the same whichever low-level strategy is employed). Here we see another advantage of the adaptive strategy: in cases where the

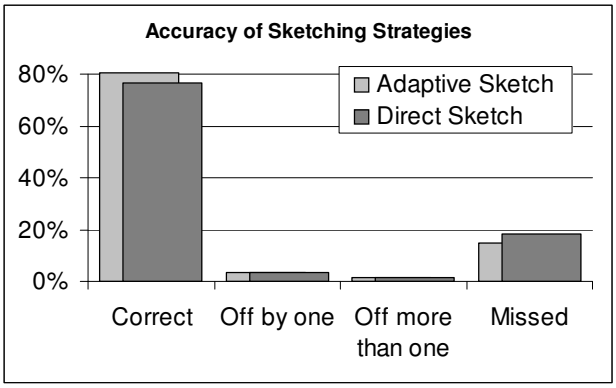


Figure 11: Accuracy for sketches on uniform traffic

threshold is not exceeded and so no sketch is made then exact results are kept and so the results there will be correct. This reduces the fraction of misses from 18% to 14%. Further improvements to the accuracy can be made by increasing the size of the sketch. We would also expect the results to be much more accurate on real data: the synthetic data here is generated uniformly, which is the most challenging case for approximate methods, since all values have counts which are similar, making finding the heavy hitters harder than on a more realistic skewed distribution.

8. CONCLUSIONS

In this paper, we examined the problem of integrating streaming algorithms for holistic aggregates into a DSMS. We modified the Gigascope system to recognize UDAFs, and incorporated algorithms for computing approximate quantiles and heavy hitters. When we tested these algorithms, the best versions ran comfortably on a 2 Gbit/sec input stream under challenging conditions, leading us to conclude that the algorithms can be used to monitor OC48 links. Our UDAFs for quantiles and heavy hitters have been incorporated into the Gigascope library and are being used in production systems.

Achieving high performance required a significant amount of design, testing, and research. The default implementation, in which the UDAF is fed tuples derived from the source stream, had unacceptably bad performance. Another complicating factor is the extreme skew in network data. Most groups are small, containing only a few tuples, but most tuples are contained in large groups. Large-space UDAFs (such as sketches) will have unacceptably large space overheads due to the small groups, while exact algorithms will have unacceptably large space overheads due to the large groups.

We applied a set of techniques to achieve practical implementations, which may be summarized as:

1. Break the UDAF processing into a simple, fast low-level subaggregate which runs close to the data stream, and a high-level superaggregate which computes the desired result. The subaggregation/superaggregation approach allows us to write fast code that does not deal with the complexities of handling large data (integer arithmetic, compact and therefore cacheable data structures, no mallocs, etc.). We also found that running a data reduction query close to the data source to be critical for performance. In many situations, such as code running in routers or in NICs, the low-level query system might not have the resources to do more than compute simple subaggregates.

2. The subaggregate/superaggregate split gives us considerable flexibility in devising UDAF algorithms. We found that we need to experiment to find the best combination of algorithms for a particular architecture. Factors include the streaming rate, data skew, and characteristics of the architecture (e.g., whether the sub and superaggregates run on shared or separate processors).

3. The properties of the data stream might dictate the choice of best algorithm, requiring adaptivity for good performance. Our live IP data stream showed large variations in its behavior, particularly in the extremely skewed distribution of tuples to groups. By writing an adaptive sketch algorithm for the heavy hitters UDAF, we overcame space use problems associated with sketch-based algorithms. When properly used, sketches are a very effective technique in practice.

4. The performance-limiting bottleneck for aggregation in Gigascope occurs when aggregates are flushed at epoch boundaries. This problem is significant even with simple aggregates, and holistic UDAFs can require expensive OUTPUT functions and large data copies to transfer state. While Gigascope ameliorates this problem to some degree by using a lazy flush at epoch boundaries, it is still the point at which a lot of work is required in a short amount of time. We can improve the schedulability of UDAFs by keeping the state small, especially at the low-level queries, and by taking steps, possibly proactive ones, to ensure that the OUTPUT function is fast.

We observe that it is crucial for UDAFs to adapt their resource consumption to the stream they encounter. Selection-based methods have this behavior inherent in their operation, but for sketch-based methods, we had to engineer this property. It was also necessary to find the right division of work between the low level and the high level. For the quantile UDAF, the best approach was usually, but not always, to do a small amount of aggregation at the low level. For the heavy hitters UDAF, trying to do all the work at either one of the high level or low level alone caused unacceptable packet drops. The skewed nature of real data meant that doing some simple hash-table aggregation of data was sufficiently powerful to reduce the cost of computing the sketch at the high level, but sufficiently inexpensive not to strain the low level system. Combining these techniques has given the first successful application of sketching methods to real-time high speed network data.

Computing approximations to holistic aggregates by incorporating UDAFs into a data stream management system provides us with great amount of flexibility in writing and expressing queries. We can write ad-hoc queries, and the output streams can be used for many purposes. For example, it is a simple matter to write a query which computes the heavy hitters among the median packet lengths in packets from all source IP addresses, by chaining together two queries. The UDAF specification language that we developed allows us additional flexibility, because the return value of an aggregate can be the UDAF state itself. For example, we can compute sketches on two data streams, join them, and compute a change detection function on the joined stream.

Future Directions. A curious aspect of our approach is that we break holistic aggregates, which are supposed to be indecomposable, into sub and superaggregates. This approach is effective because we are computing *approximations* to the holistic aggregates — and these approximations are "algebraic".

The decomposition of approximate holistic aggregates opens a new direction in query optimization. One direction is that of choosing the evaluation plan. For example, computing heavy hitters is much faster on a pre-aggregated data stream. Our best heavy hitters UDAF in fact does limited pre-aggregation. Depending on the data characteristics, there might be a collection of possible algorithms. Choosing the evaluation algorithm then becomes part of query optimization.

We note that its also possible to decompose holistic aggregates into more than two levels. For example, we might want to compute heavy hitters from the traffic flowing through a collection of routers. We need to combine the aggregate data from each router, which itself is computed using a subaggregate/superaggregate UDAF. The query planner needs to set up and optimize this query in a way that is transparent to the user.

9. REFERENCES

[1] Agilent Technologies. RouterTester. http://advanced.comms.agilent.com/RouterTester/.

[2] N. Alon, P. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. In *Proc. ACM PODS Conf.*, pages 10–20, 1999.

[3] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proc. ACM STOC*, pages 20–29, 1996. Journal version in *Journal of Computer and System Sciences*, 58:137–147, 1999.

[4] A. Arasu and et al. STREAM: The Stanford stream data manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, 2003.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. ACM PODS*, pages 1–16, 2002.

[6] D. Carney and et al. Monitoring streams - a new class of data management applications. In *Proc VLDB*, pages 215–226, 2002.

[7] S. Chandrasekaran and et al. TelegraphCQ: Continuous dataflow procesing for an uncertain world. In *Proc. CIDR*, 2003.

[8] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proc. ICALP*, pages 693–703, 2002.

[9] G. Cormode, M. Datar, P. Indyk, and S. Muthukrishnan. Comparing data streams using Hamming norms. In *Proc. Intl. Conf. VLDB*, pages 335–345, 2002.

[10] G. Cormode and S. Muthukrishnan. Improved data stream summary: The count-min sketch and its applications. In *Proc. Latin American Informatics (LATIN)*, 2003. Journal version to appear in Journal of Algorithms.

[11] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: high performance network monitoring with an SQL interface. In *Proc. ACM SIGMOD*, page 262, 2002.

[12] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. ACM SIGMOD*, pages 647–651, 2003.

[13] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. The Gigascope stream database. *IEEE Data Engineering Bulletin*, 26(1): pages 27–32, 2003.

[14] A. Dobra, M. Garofalakis, J. E. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *Proc. ACM SIGMOD*, pages 61–72, 2002.

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

[16] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for database applications. *JCSS*, 31:182–209,

1985.

[17] M. Garofalakis, J. Gehrke, and R. Rastogi. Querying and mining data streams: You only get one look. In *Proc. ACM SIGMOD*, 2002.

[18] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *Proc. ACM SIGMOD Conf.*, pages 13–24, 2001.

[19] P. Gibbons. Distinct sampling for highly-accurate answers to distinct value queries and event reports. In *Proc. VLDB*, pages 541–550, 2001.

[20] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *Proc. Intl. Conf. VLDB*, pages 466–475, 1998.

[21] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proc. Intl. Conf. VLDB*, pages 79–88, 2001.

[22] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *Proc. Intl. Conf. VLDB*, pages 454–465, 2002.

[23] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proc. of the 12th Intl. Conf. on Data Engineering*, pages 152–159, 1996.

[24] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. *ACM SIGMOD Record*, 30(2):58–66, 2001.

[25] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proc. ACM Symp. on Theory of Computing*, pages 471–475, 2001.

[26] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *Proc. FOCS*, pages 359–366, 2000.

[27] ISO DBL LHR-004 and ANSI X3H2-95-364. (ISO/ANSI Working Draft) Database Language SQL3.

[28] R. Karp, C. Papadimitriou, and S. Shenker. A simple algorithm for finding frequent elements in sets and bags. *ACM TODS*, 2003.

[29] N. Koudas and D. Srivastava. Data stream query processing: A tutorial. In *Proc. VLDB*, page 1149, 2003.

[30] A. Lerner and D. Shasha. The virtues and challenges of ad hoc + streams querying in finance. *Data Engineering Bulletin*, 26(1):49–56, 2003.

[31] S. Madden and M. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. IEEE ICDE Conf.*, 2002.

[32] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proc. VLDB*, pages 346–357, 2002.

[33] G. Manku, S. Rajagopalan, and B. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proceedings ACM SIGMOD*, pages 426–435, 1998.

[34] S. Muthukrishnan. Data streams: Algorithms and applications. In *ACM-SIAM Symp. Discrete Algorithms*, `http://athos.rutgers.edu/~muthu/stream-1-1.ps`, 2003.

[35] Stanford stream data manager. `http://www-db.stanford.edu/stream/sqr`, 2003. J. Widom and *et al*.

[36] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *Proc. USENIX Technical Conf.*, 1998.

[37] N. Thaper, P. Indyk, S. Guha, and N. Koudas. Dynamic multidimensional histograms. In *Proc. ACM SIGMOD*, pages 359–366, 2002.

[38] H. Wang and C. Zaniolo. ATLaS: A native extension of SQL for data mining. In *SIAM Intl. Conf. Data Mining*, 2003.