Multiple Aggregations Over Data Streams

Rui Zhang[†] Nick Koudas[‡] Beng Chin Ooi[†] Divesh Srivastava[§]

[†] National Univ. of Singapore
{zhangru1,ooibc}@comp.nus.edu.sg

[‡] Univ. of Toronto
koudas@cs.toronto.edu

[§] AT&T Labs–Research
divesh@research.att.com

ABSTRACT

Monitoring aggregates on IP traffic data streams is a compelling application for data stream management systems. The need for exploratory IP traffic data analysis naturally leads to posing related aggregation queries on data streams, that differ only in the choice of grouping attributes. In this paper, we address this problem of efficiently computing multiple aggregations over high speed data streams, based on a two-level LFTA/HFTA DSMS architecture, inspired by Gigascope.

Our first contribution is the insight that in such a scenario, additionally computing and maintaining fine-granularity aggregation queries (phantoms) at the LFTA has the benefit of supporting shared computation. Our second contribution is an investigation into the problem of identifying beneficial LFTA configurations of phantoms and user-queries. We formulate this problem as a cost optimization problem, which consists of two sub-optimization problems: how to choose phantoms and how to allocate space for them in the LFTA. We formally show the hardness of determining the optimal configuration, and propose cost greedy heuristics for these independent sub-problems based on detailed analyses. Our final contribution is a thorough experimental study, based on real IP traffic data, as well as synthetic data, to demonstrate the effectiveness of our techniques for identifying beneficial configurations.

1. INTRODUCTION

The phenomenon of data streams is real. 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; yet, the need exists to query and analyze this data.

The quintessential application seems to be the processing of IP traffic data in the network (see, e.g., [3, 15]). 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, e.g., keeping track of statistics, 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 aggregation queries in this scenario: to provide simple statistical summaries of the traffic carried by a link, to identify normal activity vs activity under denial of service attack, etc. For example, a common IP network analysis query is "for every source IP and 5 minute interval, report the total number of packets, provided this number of packets is more than 100". Thus, *monitoring aggregates on IP traffic data streams* is a compelling application.

There has been a concerted effort in recent years to build data stream management systems (DSMSs), either for general purpose or for a specific streaming application. Many of the DSMSs are motivated by monitoring applications. Example DSMSs are in [1, 4, 5, 8, 19]. Of these DSMSs, Gigascope [8] appears to have been tailored for processing high speed IP traffic data. This is, in large measure, due to Gigascope's two layer architecture for query processing. The low level query nodes (or LFTAs[1]) perform simple operations such as selection, projection and aggregation on a high speed stream, greatly reducing the volume of the data that is fed to the high level query nodes (or HFTAs). The HFTAs can then perform more complex processing on the reduced volume (and speed) of data obtained from the LFTA.

The need for exploratory IP traffic data analysis naturally leads to posing related aggregation queries on data streams, that differ only in the choice of grouping attributes. For example, "for every destination IP, destination port and 5 minute interval, report the average packet length", and "for every source IP, destination IP and 5 minute interval, report the average packet length". An extreme case is that of the data cube, i.e., computing aggregates for every subset of a given set of grouping attributes; more realistic is the case where specified subsets of the grouping attributes (such as "source IP, source port", "destination IP, destination port" and "source IP, destination IP") are of interest. In this paper, we address this problem of efficiently computing *multiple aggregations over high speed data streams*, based on the LFTA/HFTA architecture of Gigascope, and make the following contributions:

- Our first contribution is the insight that when computing multiple aggregation queries that differ only in their grouping attributes, it is often beneficial to additionally compute and maintain *phantoms* at the LFTA.

  These are fine-granularity aggregation queries that, while not of interest to the user, allow for shared computation between multiple aggregation queries over a high speed data stream.

- Our second contribution is an investigation into the problem of identifying beneficial configurations of phantoms and user-queries in Gigascope's LFTA.

  We formulate this problem as a cost optimization problem, which consists of two sub-optimization problems: how to choose phantoms and how to allocate space for hash tables

[1]FTA stands for "Filter, Transform, Aggregate".

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA

in the LFTA amongst a set of phantoms and user queries. We formally show the hardness of determining the optimal configuration, and propose cost greedy heuristics for both the sub-optimization problems based on detailed analyses.

- Our final contribution is a thorough experimental study, based on real IP traffic data, as well as synthetic data, to understand the effectiveness of our techniques for identifying beneficial configurations.

  We demonstrate that the heuristics result in near optimal configurations (within 15-20% most of the time) for processing multiple aggregations over high speed streams. Further, choosing a configuration is extremely fast, taking only a few milliseconds; this permits adaptive modification of the configuration to changes in the data stream distributions.

The rest of the paper is organized as follows. We first motivate the problem and our solution techniques in Section 2. We then give a formal definition of the problem, formulate the cost model for it, and present algorithms for choosing phantoms in Section 3. Section 4 presents our collision rate model at the LFTA, which is a key component of the cost model. Section 5 analyzes space allocation schemes. Section 6 presents our experimental study. Related work is summarized in Section 7. We finally conclude in Section 8.

2. MOTIVATION

In this section, we describe the problem of efficiently processing multiple aggregations over high speed data streams, based on the architecture of Gigascope, and motivate our solution techniques, in an example driven fashion.

2.1 Gigascope's Two Level Architecture

Gigascope splits a (potentially complex) query over high speed tuple data streams into two parts, (i) simple low-level queries (at the LFTA) over high speed data streams, which serve to reduce data volumes, and (ii) (potentially complex) high-level queries (at the HFTA) over the low speed data streams seen at the HFTA. LFTAs can be processed on a Network Interface Card (NIC), which has both processing capability and limited memory (a few MBs). HFTAs are typically processed in a host machine's main memory (which can be hundreds of MB to several GB).

2.2 Processing a Single Aggregation

Let us first see how a single aggregation query is processed in Gigascope. Consider a data stream relation R (e.g., `IP packet headers`), with four attributes A, B, C, D (e.g., `source IP`, `source port`, `destination IP`, `destination port`), in addition to a time attribute. Suppose the user defines the following aggregation query:

```
Q0:  select A, tb, count(*) as cnt
     from R
     group by A, time/60 as tb
```

Figure 1 is an abstracted model of Gigascope. M_L corresponds to the LFTA, and M_H corresponds to the HFTA. Q0 is processed in Gigascope as follows. When a data stream record in R arrives, it is observed at M_L. M_L maintains a hash table consisting of a specified number of entries, and each entry is a {*group*, *count*} pair. *group* identifies the most recently observed group that hashes to this entry and *count* keep track of the number of times that *group* has been recently observed without observing other groups that hash to the same entry.

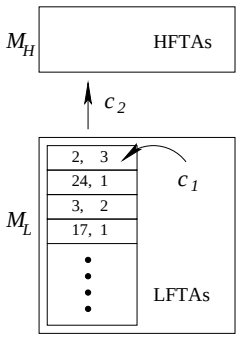


Figure 1: Single aggregation in Gigascope

When a new record r hashes to entry bk, Gigascope checks if r belongs to the same group as the existing group in bk. If yes, the *count* is incremented by 1. Otherwise, a *collision* is said to occur. In this case, first the current entry in bk is evicted to M_H. Then, a new group corresponding to r is created in bk in M_L and the *count* of the group corresponding to r is set to 1.

Query Q0 is processed by Gigascope in an epoch by epoch fashion, for an epoch of length 1 minute (i.e., time/60). This means that at the end of every minute, all the entries in M_L would be evicted to M_H to compute the aggregation results for this epoch at the HFTA. At the HFTA, multiple tuples for the same group in the same epoch may be seen because of evictions, and these are combined to compute the desired query answer.

Consider an example stream with the following prefix: 2, 24, 2, 2, 3, 17, 3, 4. At the LFTA, suppose we use a simple hash function which is the remainder modulo 10. The first item in the stream, 2, hashes to a certain entry. We check the entry in the hash table, which is empty in the beginning, and so we add the entry $(2, 1)$ to the hash table. Then we see 24, which hashes to a different entry of the hash table. Similarly an entry $(24, 1)$ is added. The third item is 2. We check the hash table and find that the existing entry where 2 hashes to contains the same group 2, so we just increment the count of the entry by 1, resulting in $(2, 2)$. The rest of the items are similarly processed one by one. After we processed the 7^{th} item, which is 3, the status of the hash table is shown in Figure 1. The next item 4 hashes to the same entry as 24. When we check the existing entry in the hash table, we find the new group is different from the existing one. In this case, we evict the existing entry $(24, 1)$ to M_H and set the entry to $(4, 1)$.

Gigascope is especially designed for processing network level packet data. Usually this data exhibits a lot of clusteredness, that is, all packets in a flow have the same source/destination IP/port. Therefore, the likelihood of a collision is very low until many packets have been observed. In this fashion, the data volume fed to M_H is greatly reduced.

2.3 Cost of Processing a Single Aggregation

Since M_H has much more space and a much reduced volume of data to process, processing at M_H does not dominate the total cost. The overall bottlenecks are:

- The cost of looking up the hash table in M_L, and possible update in case of a collision. This whole operation, called a *probe*, has a nearly constant cost c_1.
- The cost of transferring an entry from M_L to M_H. This operation, called an *eviction*, has a nearly constant cost c_2.

  Usually, c_2 is much higher than c_1 because the transfer from M_L to M_H is more expensive than a probe in M_L.

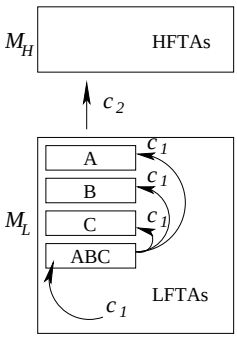


Figure 2: Multiple aggregations using phantoms

The total cost of query processing thus depends on the number of collisions incurred, which is determined by the number of groups of the data and collision rate of the hash table. The number of groups depends on the nature of the data. The collision rate depends on the hash function, size of the hash table, and the data distribution. Therefore, generally, what we can do is to devise a good hash function and allocate more space (within space and peak load constraints, as we will discuss more later) to the hash table in order to minimize the total cost.

2.4 Processing Multiple Aggregations Naively

Given the method to process single aggregation queries, and its cost model, based on the Gigascope architecture, we now examine the problem of evaluating multiple aggregation queries. Suppose the user is interested in the following three aggregation queries:

```
Q1:  select A, count(*)
     from R
     group by A

Q2:  select B, count(*)
     from R
     group by B

Q3:  select C, count(*)
     From R
     group by C
```

A straightforward method is to process each query separately using the above single aggregation query processing algorithm, so we maintain, in M_L, three hash tables for A, B, and C separately. For each incoming record, we need to probe each hash table, and if there is a collision, some entry gets evicted to M_H.

2.5 Processing Multiple Aggregations Using Phantoms

Since we are processing multiple aggregation queries, we may be able to share the computation that is common to each one and thereby reduce the overall processing cost. For example, we can additionally maintain a hash table for the relation ABC in M_L as shown in Figure 2. If we have the counts of each group in ABC, we can derive the counts of each group of A, B and C from it. The intuition is that, when a new record arrives, instead of probing three hash tables A, B and C, we only probe the hash table ABC. We would delay the probes on A, B and C (we omit "hash tables" when the context is clear) until the point when an entry is evicted from ABC.

Since the aggregation queries of A, B and C are derived from ABC, we say that ABC *feeds* A, B and C. Although ABC is not of interest to the user, its maintenance could help reduce the overall cost. We call such a relation as a *phantom*. While for A, B and C, whose aggregate information is of user interest, we call each of them as a *query*. Both *queries* and *phantoms* are called *relations*.

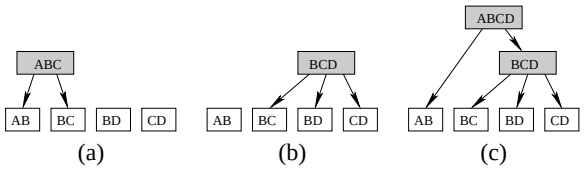


Figure 3: Choices of phantoms

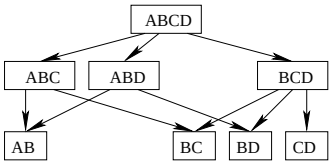


Figure 4: Feeding graph for the relations

Now we examine Figure 2 to illustrate how the instantiation of a phantom can benefit the total evaluation cost. To be fair, the total space used for the hash tables should be the same with or without the phantoms. So when we add the phantom ABC, the size of the hash tables for A, B and C need to be reduced. Suppose the total space allocated for the three queries is M. While we have many choices of space allocation between the hash table instantiations, let us allocate equal space to each instantiation, for simplicity of exposition. Without phantoms, we allocate $M/3$ to each hash table. With the phantom ABC, we allocate $M/4$ to each hash table. Also assume that A, B and C have the same collision rate. Without the phantom, their collision rate is denoted x_1; with the phantom, their collision rate is denoted x_1'. Since the hash table size of A, B and C is smaller in the presence of the phantom, x_1' should be larger than x_1. Let the collision rate of phantom ABC be x_2.

Consider the cost for processing n records. Without the phantom, we need to probe 3 hash tables for each incoming record, and there are $x_1 n$ evictions from each table. Therefore the total cost is:

$$E_1 = 3nc_1 + 3x_1nc_2 \tag{1}$$

With the phantom, we probe only ABC for each incoming record and there would be $x_2 n$ evictions. For each of these evictions, we probe A, B and C, and hence there are $x_1'x_2n$ evictions from each of them. The total cost is:

$$E_2 = nc_1 + 3x_2nc_1 + 3x_1'x_2nc_2 \tag{2}$$

Comparing Equations 1 and 2, we can get the difference of E_1 and E_2 as follows

$$E_1 - E_2 = [(2 - 3x_2)c_1 + 3(x_1 - x_1'x_2)c_2]n \tag{3}$$

If x_2 is small enough so that both $(2 - 3x_2)$ and $(x_1 - x_1'x_2)$ are larger than 0, then E_2 will be smaller than E_1, and therefore instantiation of the phantom benefits the total cost. If x_2 is not small enough so that one of $(2 - 3x_2)$ and $(x_1 - x_1'x_2)$ is larger than 0 but the other is less than 0, then $E_1 - E_2$ depends on the relationship of c_1 and c_2. If x_2 is so large that both $(2 - 3x_2)$ and $(x_1 - x_1'x_2)$ are less than 0, then the instantiation of the phantom increases the cost and therefore we should not instantiate it.

2.6 Choice of Phantoms

In the above example, we have only considered one phantom. In fact, we can have many phantoms and multiple levels of phantoms.

Again, consider stream relation R with four attributes A, B, C, D. Suppose the queries are AB, BC, BD and CD. We could instantiate phantom ABC, which feeds AB and BC as shown in Figure 3(a) (a shaded box is a phantom and a non-shaded box is a query); or we could instantiate phantom BCD, which feeds BC, BD and CD as shown in Figure 3(b); or we could instantiate BCD and ABCD, where ABCD feeds AB and BCD as shown in Figure 3(c). We only list three choices here, although there are many other possibilities.

It is easy to prove that a phantom that feeds less than two relations is never beneficial. So by combining two or more queries, we can obtain all possible phantoms and plot them in a *relation feeding graph* as in Figure 4. Each node in the graph is a relation and each directed edge shows a *feed* relationship between two nodes, that is, the parent feeds the child. Note that this feed relationship can be "short circuited", that is, a node can be directly fed by any of its ancestors in the graph. For example, AB could be fed directly by ABCD without having ABC or ABD instantiated.

Given the relation feeding graph, and a set of user queries that are instantiated in the LFTA, one optimization problem is to identify the phantoms that we should instantiate to minimize the cost. The exhaustive method is obvious, that is, we try all possible combinations of the phantoms and calculate the cost of each combination as in Section 2.5. Then we choose the one with the minimum cost. However, the exhaustive method is too expensive, especially for data stream systems where a fast response is essential.

In our example in Section 2.5, we assumed that each hash table has the same size for simplicity of exposition. However, given a set of phantoms and queries to instantiate in M_L, how does the allocation of space to each hash table affect the collision rate and hence the cost? Therefore, another optimization problem is that, given a set of relations to instantiate, how to allocate space to them so that the cost is minimized.

In summary, our cost optimization problem consists of two sub-optimization problems: how to choose phantoms and how to allocate space. We formulate the cost model for the multiple aggregation problem and propose a cost greedy algorithm to choose phantoms. In addition, for a given set of relations to instantiate, we analyze which space allocation gives the minimum cost; in case the optimal space allocation cannot be calculated, we propose heuristics which can approximate the optimal solution very well.

3. PROBLEM FORMULATION

In this section, we formulate our cost model, and give a formal definition of our optimization problem. We present hardness results, motivating the greedy heuristic algorithms for identifying optimal configurations.

3.1 Notation

When we have chosen a set of phantoms to instantiate in the LFTA, we call the set of instantiated relations (i.e., the chosen phantoms and user queries) as a *configuration*. For example, Figure 3 shows three configurations for an example query. While the feeding graph is a DAG, a configuration is always a tree, consistent with the path structure of the feeding graph. If a relation in a configuration is directly fed by the stream, we call it a *raw relation*. For example, ABC, BD, CD are raw relations in Figure 3(a); and ABCD is the only raw relation in Figure 3(c). If a relation in a configuration has no child, then it is called a *leaf relation* or just *leaf*. User queries are always instantiated in the LFTA, therefore only queries are leaves. For all the configurations in Figure 3, queries AB, BC, BD and CD are the leaves. Note that raw relations and leaf relations need not be mutually exclusive. For example, BD and CD are both raw and leaf relations in Figure 3(a).

We next develop our cost model, which determines the total cost incurred during data stream processing of a configuration. We then formalize the optimization problem studied in this paper.

3.2 Cost Model

Recall that aggregation queries usually include a specification of temporal epochs of interest. For example, in the query "for every destination IP, destination port and 5 minute interval, report the average packet length", the "5 minute interval" is the epoch of interest. During stream processing within an epoch (e.g., a specific 5 minute interval), the aggregation query hash tables need to be maintained, for each record in the stream. At the end of an epoch, all the hash tables of the user queries at the LFTA need to be evicted to the HFTA to complete the user query computations. Thus, there are two components to the cost: intra-epoch cost, and end-of-epoch cost. We discuss each of these next.

3.2.1 Intra-Epoch Cost

Let E_m be the maintenance cost of all the hash tables during an epoch T (the *maintenance cost* for short). It includes updating all hash tables for the raw relations when a new record in the stream is processed. If (and only if) there is collision in hash tables for the raw relations, the hash tables of the relations they feed are updated. This process recurses until the hash tables for the leaf level. Each of these updates has a cost of c_1.

If there are collisions in the hash tables for the leaf (user) queries, evictions to the HFTAs are incurred, each with the cost of c_2. Therefore, the total maintenance cost is

$$E_m = \sum_{R \in I} \mathcal{F}_R c_1 + \sum_{R \in L} \mathcal{F}_R x_R c_2 \tag{4}$$

where I is a configuration, L is the set of all leaves in I, $\mathcal{F}_R$ is the number of tuples fed to relation R during epoch T, and x_R is the collision rate of the hash table for R. $\mathcal{F}_R$ is derived as follows.

$$\mathcal{F}_R = \begin{cases} n_T & \text{if } R \in W \\ \mathcal{F}_a x_a & \text{else} \end{cases} \tag{5}$$

where W is the set of all raw relations, n_T is the number of tuples observed in T, $\mathcal{F}_a$ is the number of tuples fed to the parent of R in I, and x_a is the collision rate of the hash table for the parent of R in I. If we further define $\mathcal{F}_a = n_T$ and $x_a = 1$, when R is a raw relation, Equation 4 can be rewritten as follows.

$$E_m = \left[\sum_{R \in I} \Big(\prod_{R' \in A_R} x_{R'} \Big) c_1 + \sum_{R \in L} \Big(\prod_{R' \in A_R} x_{R'} \Big) x_R c_2 \right] n_T \tag{6}$$

where A_R is the set of all ancestors of R in I.

n_T is determined by the data stream and is not affected by the configuration. Hence, the per record cost is:

$$e_m = \sum_{R \in I} \Big(\prod_{R' \in A_R} x_{R'} \Big) c_1 + \sum_{R \in L} \Big(\prod_{R' \in A_R} x_{R'} \Big) x_R c_2 \tag{7}$$

where c_1 and c_2 are constants determined by the LFTA/HFTA architecture of the DSMS. Therefore, the cost is only affected by the feeding relationship and collision rates of the hash tables.

3.2.2 End-of-Epoch Cost

Denote the update cost at the end of epoch T as E_u (the *update cost* for short). It includes the cost of the following operations. From the raw level to the leaf level of the feeding graph of the configuration, we scan each hash table and propagate each item in the hash table to hash tables of the lower level relations they feed. Finally, we scan the leaf level hash table and evict each item in it

to the HFTA, M_H. Using an analysis similar to the one for intra-epoch costs, taking the possibilities of collisions during this phase into account, the update cost E_u can be expressed as follows.

$$\sum_{R\in I, R\notin W}[\sum_{R'\in A_R}(M_{R'} * \prod_{R''\in A_{R'}\cup R', R''\notin W} x_{R''})]c_1 + \sum_{R\in L}[M_R + \sum_{R'\in A_R}(M_{R'} * \prod_{R''\in A_{R'}\cup R', R''\notin W} x_{R''})]c_2 \quad (8)$$

where M_R is the size of the hash table of relation R, and W is the set of all raw relations.

3.3 Our Problem

Intuitively, the lower the average per-record intra-epoch cost, the lower is the load at the LFTA, increasing the likelihood that records in the stream are not dropped during query processing. We also want to ensure that the total cost of the end-of-epoch processing is manageable. This leads to the *multiple aggregation* (MA) optimization problem studied for the two-level LFTA/HFTA architecture in this paper.

Consider a set of aggregation queries over a data stream that differ only in their grouping attributes, $S_Q = \{Q_1, Q_2, ..., Q_{n_Q}\}$, and memory limit M in M_L. Determine the configuration I, of relations in the feeding graph of S_Q to instantiate in (the LFTA) M_L and also the allocation of the available memory M to the hash tables or the relations so that the per-record intra-epoch cost (Equation 7) for answering all the queries is minimized, subject to the end-of-epoch cost being less than peak load cost E_p.

For the MA optimization problem we can show the following:

THEOREM 1. *Let n be the number of possible group-by queries in the feeding graph of S_Q. If $P \neq NP$, for every $\epsilon > 0$ every polynomial time approximation algorithm for the MA problem will have a performance ratio of at least $n^{1-\epsilon}$.*

Moreover, the same is true for the corresponding maximization problem. Define the "benefit" of an aggregate in the feeding graph of S_Q, as the cost improvement with respect to the solution that computes all aggregates in the LFTA. The maximization problem aims to identify the solution with the maximum benefit. It is possible to show that there does not exist a polynomial time algorithm for this problem with performance ratio better than $n^{1-\epsilon}$, if $P \neq NP$.

Our cost optimization problem consists of two sub-problems: how to choose phantoms and how to allocate space. Given the hardness result above, we next describe cost greedy algorithms to choose phantoms, based on the cost model presented earlier. The cost model critically depends on the collision rate model, which we discuss in detail in Section 4. For a given set of relations to instantiate, we analyze which space allocation gives the minimum cost, in Section 5.

3.4 Algorithmic Strategies

The multi-aggregation (MA) problem has similarities to the view materialization (VM) problem [14]. They both have a feeding graph consisting of nodes some of which can feed some others, and we need to choose some of them to instantiate. So one possibility is to adapt the greedy algorithm developed for VM to MA. However, there are two differences between these two problems. First, instantiation of any of the views in VM will add to the benefit; while in MA, instantiation of a phantom is not always beneficial. Second, the space needed for instantiation of a view is fixed but the hash table size is flexible. Therefore, in order to adapt the VM greedy algorithm, we need to have a space allocation scheme so that the hash tables must have low collision rate and therefore each instantiated phantom is beneficial. We discuss this next.

3.4.1 Greedy by Increasing Space

The more space that is allocated to a hash table, the lower is the collision rate. On the other hand, the more groups a relation has for a fixed sized hash table, the higher is the collision rate. Let g be the number of groups in a relation. One way of allocating hash table space to a relation is proportional to the number of groups in the table. Thus, we can allocate space ϕg for a relation with g groups. ϕ is a constant and we set it large so that the hash table is guaranteed to have a low collision rate. We will develop a model to estimate collision rate in Section 4. We can then have a better sense of what value of ϕ might be good according to the analysis there.

The greedy algorithm goes as follows. We calculate the benefit of each phantom according to the cost model, i.e., the difference between the maintenance costs without and with this phantom. Let us denote this benefit by b_R. Then we calculate the benefit per unit space for each phantom R, $b_R/(\phi n_{g_R})$. We choose the phantom with the largest benefit per unit space as the first phantom to instantiate. For the other phantoms, this process is iterated. The process ends when the benefit per unit space becomes negative, the space is exhausted, or all phantoms are instantiated.

This approach has two drawbacks: (1) ϕ needs to be tuned to find the best performance. A bad choice can result in suboptimal performance. (2) By allocating space to a relation proportional to the number of its groups, we make the collision rates of all the relations the same. As we shall show later, this is not a good strategy.

3.4.2 Greedy by Increasing Collision Rates

Here, we present a different greedy algorithm for allocating space to hash tables of the relations in the LFTA. Instead of allocating a fixed amount of space to each phantom progressively, we always allocate *all available space* to the current configuration (how to allocate space among relations in a configuration is analyzed in Section 5). So as each new phantom is added to a configuration, what changes is not the total space used, but the collision rate of each table. Since the more the number of groups mapped to a fixed space, the higher the collision rate, the collision rate would increase as new phantoms are instantiated.

The greedy algorithm is as follows. At first, the configuration I only includes all the queries. We calculate the maintenance cost if a phantom R is added to I. By comparing with the maintenance cost when R is not in I, we can get the benefit. After we add this phantom to I, we iterate with the other phantoms.

As more phantoms are added into I, the overall collision rate goes up and benefit decreases. We stop when the benefit becomes negative. This algorithm depends on estimating the collision rates. We derive a model to estimate the collision rate, in Section 4.

4. THE COLLISION RATE MODEL

In this section, we develop a model to estimate the collision rate. We assume that the hash function randomly hashes the data, so each hash value is equally possible for every record. We first consider uniformly distributed data, and subsequently consider when the data exhibits clusteredness.

4.1 Randomly Distributed Data

Let g be the number of groups of a relation and b the number of buckets in the hash table. If k groups hash to a bucket, we say that this bucket has k groups. Let B_k be the number of buckets having k groups. If the records in the stream are uniformly distributed, each group has the same expected number of records, denoted by n_{rg}. So $n_{rg}k$ records will go to a bucket having k groups. Under the random hash assumption, the collision rate in this bucket is $(1 - 1/k)$. Therefore $n_{rg}k(1 - 1/k)$ collisions happen in this bucket.

The overall collision rate is obtained by summing all the collisions and then dividing by the total number of records. Therefore, we have collision rate

$$x = \frac{\sum_{k=2}^{g} B_k n_{rg} k(1-1/k)}{g n_{rg}} = \frac{\sum_{k=2}^{g} B_k (k-1)}{g} \quad (9)$$

k begins from 2 because when 0 or 1 group hashes to a bucket, no collision happens. In order to calculate it, we still need to know B_k. This problem belongs to a class of problems called the *occupancy problem*.

As we know, the expectation of k for each bucket is g/b [10]. A rough estimation of B_k based on expectation would be

$$B_k = \begin{cases} b & \text{k=g/b} \\ 0 & \text{k}\neq \text{g/b} \end{cases}$$

Substituting this for B_k in Equation 9, we get

$$x = 1 - b/g \quad (10)$$

However, in a real random process, the probability of each bucket having the same number of groups is small. In [12] (Chapter II.5), an example when $g = b = 7$ is given to calculate the probability of different distributions of groups. It is shown that probability of each of the 7 buckets having exactly 1 group is 0.006120, which makes it extremely unlikely. Therefore, we need to calculate B_k based on probability. To the best of our knowledge, no study exists on estimating B_k as we defined here.[2] Our derivation of B_k is as follows.

The probability of k groups out of g hashed to a given bucket is

$$\binom{g}{k}(1/b)^k(1-1/b)^{g-k} \quad (11)$$

Note this holds for any bucket, which means each bucket has the chance of Equation 11 to have k groups. If we assume that all b buckets are independent of each other, then statistically there are

$$b\binom{g}{k}(1/b)^k(1-1/b)^{g-k} \quad (12)$$

buckets each of which has k groups. Substitute Equation 12 for B_k in Equation 9 we have

$$x = \frac{b\sum_{k=2}^{g}\binom{g}{k}(1/b)^k(1-1/b)^{g-k}(k-1)}{g} \quad (13)$$

Our experiments on both synthetic and real data show that the actual distribution of B_k matches Equation 13 well, even though the buckets are not completely independent (they satisfy the equation $\sum_{k=1}^{b} B_k = b$).

4.2 Validation of Collision Rate Model

We have measured experimentally the collision rates on both synthetic random datasets and real datasets. The results for the real datasets are shown in Figure 5; the results for the synthetic datasets are very similar and omitted.

[2]If we use $\mathcal{B}_i$ to denote the number of balls in the i-th bucket, $\mathcal{B}_i$ are called *occupancy numbers*. This problem has been studied before and the $\mathcal{B}_i$'s follow the multinomial distribution [12] (Chapter VI.9). However, our definition of B_k is different from $\mathcal{B}_i$. Instead of the probability of a certain arrangement of the "balls" in the buckets, what we want is the distribution of the "balls".

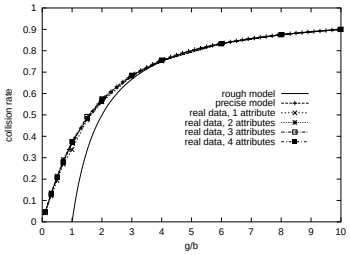


Figure 5: Collision rates of real data

The real datasets are extracted from the netflow dataset as described in Section 6.1. We have assumed random data distribution for the above analysis, while the netflow dataset has a lot clusteredness of multiple packets in a flow. In order to validate our analysis using the real data, we grouped all packets of a flow into a single record, eliminating the effect of clusteredness. (We consider clusterness in a later subsection.) After eliminating clusteredness of the data, we extracted 4 datasets which have 1, 2, 3 and 4 attributes respectively. The number of groups in these datasets are 552, 1846, 2117, 2837 respectively.

The "rough model" curve is plotted according to Equation 10 and the "precise model" curve is plotted according to Equation 13. Collision rates of the real data match the precise model very well. In all the observed collision rates, more than 95% of the experimental results have less than 5% difference from the precise model. The rough model differs greatly from the precise model when g/b is small but becomes similar as g/b gets large. The reason is that the rough model only captures the expected case, which occurs with low probability. When g becomes larger, the behavior gets closer to the average case, therefore the rough model gets close to the precise model.

For the rough model, the collision rate is only dependent on the ratio of g to b as we can see from Equation 10. We will show in Section 4.4 that the precise model is also dependent on g/b, though the function is different.

4.3 Clustered Data

The above analysis was for randomly distributed data. However, real data streams, especially the packets in netflow data (which have exactly the same values for attributes such as source/destination IP/port), are clustered. Although packets from different flows are interleaved with each other in the stream, the likelihood of these interleaved flows hashing to the same bucket is very small. Therefore we can think of the packets in a flow going through a bucket without any collision until the end of the flow. To analyze collision rate for such clustered distributions, we should consider what happens at the per flow level. If we think of each flow as one record, then we can use the same formula as in the random distribution (Equation 9) to calculate the total number of collisions as follows.

$$n_c = \sum_{k=2}^{g} B_k n_{fg} k(1-1/k) \quad (14)$$

where n_{fg} is the number of flows in each group; B_k is still calculated by Equation 12. To obtain the collision rate, we divide n_c by the total number of records, $g n_{fg} l_a$, where l_a is the average length of all the flows. Then we have the collision rate for the data with a clustered distribution as follows.

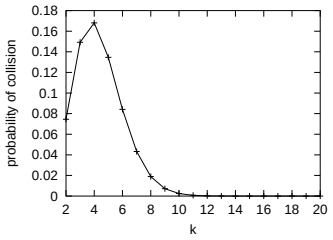


Figure 6: Probability of collision vs. k

$$x = \frac{b\sum_{k=2}^{g}\binom{g}{k}(1/b)^k(1-1/b)^{g-k}(k-1)}{gl_a} \quad (15)$$

We can see that the difference of the collision rate on data with clusteredness from that of the random data is a linear relationship over average flow length l_a. We can view the collision rate of the random data as a special case of clustered data with $l_a = 1$. The average flow length can be computed by maintaining the number of times hash table bucket entries are updated before being evicted.

4.4 On Computing Collision Rates

To calculate the collision according to Equations 13 and 15, we need to compute a sum of about g items, which can be hundreds to thousands of items. This computation is expensive. In this section, we show that actually we only need to sum up to a much smaller number of items. Further, the collision rate is almost solely dependent on g/b, therefore we can pre-compute the collision rates and store them as a function of g/b. In this way, the computational effort of collision rates is greatly reduced.

We computed the probability of collision of different k's according to Equation 13 to see how much each component contributes to the overall collision rate. Figure 6 shows the probability of collision as a function of k when $g = 3000$ and $b = 1000$. We can see that the contributions become almost 0 when k becomes larger than 11 and the shape of the curve looks like a bell. It looks like the PDF of the Gaussian distribution. Examining Equation 13, we find that except the $(k-1)$ part and b/g which is a constant, the rest are the same as the PDF of the binomial distribution. And the binomial distribution can be approximated by the Gaussian distribution. So the plots of the collision probability of different k components can be viewed as a Gaussian distribution with an amplitude of $k-1$. That's why the plot mimics the Gaussian distribution. Given its similarity, we can understand many features of the plots according to characteristics of the Gaussian distribution. The peak of the plot appears at approximately the mean, which is $\mu = g/b = 3$ (the actual maximum in Figure 6 appears at $k = 4$ due to the effect of the amplitude $k-1$). Also we know that the probability in the interval $(-\infty, \mu + 3\sigma]$ is 99.7%. So when we calculate the collision rate, we do not need to sum over all values of k but up to $\mu+3\sigma$ is enough, where $\sigma = \sqrt{g(1-1/b)/b}$. In case of Figure 6, $\mu + 3\sigma = 3 + 3 \cdot \sqrt{2.997} = 8.2$. In Figure 6, the component at $k = 8$ is as small as 0.02 already. It is not 0 yet due to the amplitude $k-1$, so we can calculate up to several more σ say $\mu + 5\sigma$, which is 12 in our case. The components after 12 are almost 0, and 12 is much smaller than 3000 (the number of groups).

The cost of computing collision rates can be further reduced. A Gaussian distribution is determined by μ and σ^2. Here we just want the sum, so we don't care about μ (the mean) but only σ^2, which equals $g(1-1/b)/b$. b in the data stream case is usually several hundred to thousands, therefore $1-1/b$ is almost 1 and the sum is essentially determined by g/b.

g/b	0.25	0.5	1	2	4	8	16	32
variation(%)	1.4	0.43	0.15	0.03	0.004	0	0	0

Table 1: Variations of the collision rate

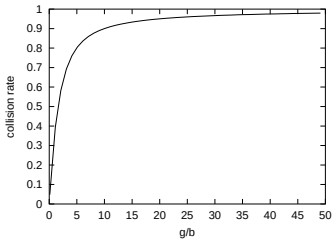


Figure 7: The collision rate curve

As our sum is not exactly a Gaussian distribution, some errors are expected. In the following, we experimentally evaluate how much the errors are. We let g/b be certain constants and compute the collision rates according to Equation 13, for b varied from 300 to 3000. Note that once b is chosen, g is also determined for a given g/b. The resulting collision rate, for g/b varied from 0.25 to 32, is almost constant for the same g/b. The maximum relative variations of collision rates as g/b varies are listed in Table 1. We observe that all the variations are less than 1.5%. Therefore the collision rate only depends on g/b and we can pre-compute collision rate and use regression to model the function.

The collision rate curve as a function of g/b is plotted in Figure 7. We divided the whole curve into 6 intervals and used two-dimensional regression to simulate the curve so as to achieve a maximum relative error of 5% in each interval. The average relative error is actually much lower, which is less than 1%.

According to our previous analysis, the hash table must have a low collision rate if we want to benefit from maintaining phantoms. Therefore, we examine the low collision rate part of this curve closely. A zoom in of the collision rate curve when collision rate is smaller than 0.4 as well as a linear regression of this part is shown in Figure 8. We observe that this part of the curve is almost a straight line and the linear regression achieves an average error of 5%. The linear function for this part is

$$x = 0.0267 + 0.354 \cdot (g/b) \quad (16)$$

Expressing this part of the collision rate linearly is important for the space allocation analysis as we will see in Section 5. In addition, since we now know how the collision rate is determined, we can suggest values of ϕ to use in the adapted greedy algorithm (by increasing space) of Section 3.4. For example, $\phi = 1$ could be a good choice, since it corresponds to a collision rate of about 0.37.

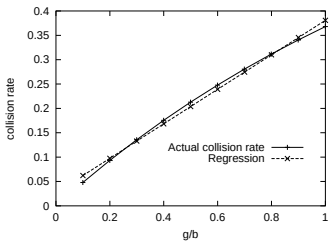


Figure 8: The low collision rate part

5. SPACE ALLOCATION

In this section, we consider the problem of space allocation, that is, given a configuration of certain relations (phantoms and queries) to be instantiated, how to allocate the available space M to their hash tables so that the overall cost is minimized. We start with a simple two-level configuration in Section 5.1, and identify the difficulties in analyzing more complex configurations. Heuristics for space allocation are discussed in Section 5.2.

5.1 A Case of Two Levels

We first study the case when there is only one phantom R_0 and it feeds all f queries, $R_1, R_2, \ldots, R_f$. Let x_0 be the collision rate of the phantom, $x_1, x_2, \ldots, x_f$ be the collision rate of the queries. In order to benefit from maintaining a phantom, its collision rate must be low, therefore we only care about the low collision rate part of the collision rate curve. According to Section 4.4, this part of the curve can be expressed as a linear function $x = \alpha + \mu g/b$, where α=0.0267 and μ=0.354.[3] Since α is small, here we make a further approximation to let $x = \mu g/b$. We will discuss later how the results are affected when we consider α. Given the approximation, $x_i = \mu g_i/b_i$, $i = 0, 1, ..., f$. The total size is M, so $M = \sum_{i=0}^{f} b_i$. The cost of this configuration is

$$\begin{aligned} e &= c_1 + f x_0 c_1 + x_0 \sum_{i=1}^{f} x_i c_2 \\ &= f\mu\frac{g_0}{b_0}c_1 + \mu\frac{g_0}{b_0}\sum_{i=1}^{f}\mu\frac{g_i}{b_i}c_2 + c_1 \\ &= \mu\frac{g_0}{b_0}(fc_1 + \mu c_2\sum_{i=1}^{f}\frac{g_i}{b_i}) + c_1 \\ &= \frac{\mu g_0}{M - \sum_{i=1}^{f} b_i}(fc_1 + \mu c_2\sum_{i=1}^{f}\frac{g_i}{b_i}) + c_1 \end{aligned} \tag{17}$$

e is a function of multiple variables, $b_1, b_2, ..., b_f$. To find out the minimum, we equate the partial derivatives of e to 0. In the following, we calculate the partial derivative of e over b_i, $i = 1, 2, ...f$.

$$\frac{\partial e}{\partial b_i} = \frac{\mu g_0}{(M - \sum_{i=1}^{f} b_i)^2}(fc_1 + \mu c_2\sum_{i=1}^{f}\frac{g_i}{b_i}) + \frac{\mu g_0}{M - \sum_{i=1}^{f} b_i}\mu c_2 g_i(-\frac{1}{b_i^2})$$

Let $\frac{\partial e}{\partial b_i} = 0$, then

$$\frac{\mu g_0}{M - \sum_{i=1}^{f} b_i}\left(\frac{fc_1 + \mu c_2\sum_{i=1}^{f}\frac{g_i}{b_i}}{M - \sum_{i=1}^{f} b_i} - \frac{\mu c_2 g_i}{b_i^2}\right) = 0$$

$\frac{\mu g_0}{M - \sum_{i=1}^{f} b_i}$ is non-zero, so

$$\frac{fc_1 + \mu c_2\sum_{i=1}^{f}\frac{g_i}{b_i}}{M - \sum_{i=1}^{f} b_i} = \frac{\mu c_2 g_i}{b_i^2} \tag{18}$$

for $i = 1, 2, ..., f$.

Observe that left hand side of the equation is the same for any i. So we have

$$\frac{g_1}{b_1^2} = \frac{g_2}{b_2^2} = ... = \frac{g_f}{b_f^2}$$

that is, b_i is proportional to $\sqrt{g_i}$.

Let $b_i = \frac{\sqrt{g_i}}{\nu}$, $i = 1, 2, ...f$. Substituting this for b_i in Equation 18, we have

$$\mu c_2 M\nu^2 - 2\mu c_2\sum_{i=1}^{f}\sqrt{g_i}\nu - fc_1 = 0 \tag{19}$$

This is a quadratic equation over ν. Solving it we have

$$\nu = \frac{\mu c_2\sum_{i=1}^{f}\sqrt{g_i} \pm \sqrt{\mu^2 c_2^2(\sum_{i=1}^{f}\sqrt{g_i})^2 + f\mu c_1 c_2 M}}{\mu c_2 M}$$

$\nu > 0$, so only the one with "+" before the square root on the numerator is the solution. So

$$\begin{aligned} b_i = \frac{\sqrt{g_i}}{\nu} &= \frac{\mu c_2 M\sqrt{g_i}}{\mu c_2\sum_{j=1}^{f}\sqrt{g_j} + \sqrt{\mu^2 c_2^2(\sum_{j=1}^{f}\sqrt{g_j})^2 + f\mu c_1 c_2 M}} \\ &= \frac{M}{\frac{\sum_{j=1}^{f}\sqrt{g_j}}{\sqrt{g_i}} + \sqrt{\left(\frac{\sum_{j=1}^{f}\sqrt{g_j}}{\sqrt{g_i}}\right)^2 + \frac{fc_1 M}{\mu c_2 g_i}}} \end{aligned} \tag{20}$$

where $i = 1, 2, ..., f$.

$$\begin{aligned} b_0 &= M - \textstyle\sum_{i=1}^{f} b_i \\ &= M - \frac{M\sum_{j=1}^{f}\sqrt{g_j}}{\sum_{j=1}^{f}\sqrt{g_j} + \sqrt{(\sum_{j=1}^{f}\sqrt{g_j})^2 + \frac{fc_1 M}{\mu c_2}}} \end{aligned} \tag{21}$$

A key consequence of our analysis is that we should allocate space proportional to the *square root* of the number of groups in order to achieve the minimum cost. Another interesting point is that b_0 (the space allocated to the hash table of the phantom) always takes more than half the available space.

While the 2-level case results in a quadratic equation (that is, Equation 19), a similar analysis on the simplest 3-level case results in an equation of order 8. According to Abel's impossibility theorem, equations of order higher than 4 cannot be solved algebraically in terms of a finite number of additions, subtractions, multiplications, divisions, and root extractions (in the sequel, we simply say "unsolvable"). More general multi-level configurations generate equations of even higher order which are unsolvable, therefore we would use heuristics to decide space allocation for the these unsolvable cases based on the analysis available. Experiments show that our proposed heuristics based on the analysis are very close to optimal and better than other heuristics.

[3] Actually, even if the collision rate for the optimal allocation is a little higher than 0.4, we can still use linear regression for that part. The values of α and μ would be a little different, but experiments show that small variation in their values does not affect the result much.

5.2 Heuristics

For unsolvable configurations, we propose heuristics to allocate space based on the analysis of the solvable cases and partial results we can get from the unsolvable cases. We observe that while b_i^2 (b_i is a leaf relation) is proportional to $\mu c_2 g_i$, which is affected by its own number of groups, according to our analysis on the three level case, b_1^2 (b_1 is a non-leaf relation) is proportional to $d\mu g_1 c_1 + \mu g_1 c_2 \sum_{i=1}^{d} x_{1i}$ (note that $\mu g/b = x$), where x_{1i} are the collision rates of tables for the children of b_1. b_1 is affected not only by its own number of groups, but also its children's. The intuition is that we should care more about a relation when it has children in the feeding graph of the configuration. Therefore we will consider the following space allocation schemes which add some weights to a relation when it has children to feed.

Heuristic 1: Supernode with Linear Combination (SL). We start from the leaf level of the configuration. Each phantom at the record level together with all its children are viewed as a supernode. The number of groups of this supernode is the sum of the number of groups of the phantom and all its children. Then we view the supernode as a query and do the above compaction recursively until the configuration become all queries. For the all query configuration, we can allocate space optimally. After this allocation, each query (some may be supernodes) has some space. We decompose a supernode to a two-level structure and allocate space according to the analysis of Section 5.1, that is, allocate space proportional to the square root of the number of groups. If there are still supernodes in the structure, we do the decomposition recursively.

Heuristic 2: Supernode with Square Root Combination (SR). This heuristic is the same as SL except the calculation of the group of the supernode. Since in the two level case we see that the space should be proportional to the square root of the number of groups, we can also let the square root of the number of groups of the supernode be the sum of the square roots of all its relations.

Note that both SL and SR give the optimal result for the case of one phantom feeding all queries. We will also try two other simple heuristics which are not based on our analysis as a comparison to the above two more well-founded heuristics.

Heuristic 3: Linear Proportional Allocation (PL). This heuristic simply allocates space to each relation proportional to the number of groups of that relation.

Heuristic 4: Square Root Proportional Allocation (PR). This heuristic allocates space to each relation proportionally to the square root of the number of groups of that relation.

Although we cannot compute the optimal solution for space allocation of some cases, there does exist a space allocation which gives the minimum cost for each configuration. One way to find this optimal space allocation is to try all possibilities of allocation of space at certain granularity. For example, if the configuration is AB feeds A and B, and total space is 10, we can first allocate 1 to AB, 1 to A, and 8 to B. Then we try 1 to AB, 2 to A, and 7 to B, and so on. By comparing the cost of all these space allocation choices we will find the optimal one. We call this method the **exhaustive space allocation (ES)**. Obviously this strategy is too expensive to be practical, but we use it in our experiments to compare with the four space allocation schemes and see how much the heuristics differ from the optimal choice. The results of ES are affected by the granularity of varying the space allocation. In our experiments, we found that using a granularity of 1% of M is small enough to provide accurate results.

The space allocation schemes are independent of the phantom choosing strategies, that is, given a configuration, a space allocation scheme will produce a space allocation no matter in what order the relations in the configuration are chosen. Therefore we will evaluate space allocation schemes and phantom choosing strategies independently.

5.3 Revisiting Simplifications

From the beginning of the analysis on space allocation, we have made an approximation on the linear expression of the collision rate, that is, we let x equal $\mu g/b$ instead of $\alpha + \mu g/b$. We also did the analysis when we let $x = \alpha + \mu g/b$. The result of the case with no phantom is the same. The case with one phantom feeding all queries results in a quartic equation which can be solved, so we can still get an optimal solution for this case. However, because solving a quartic equation is much more complex than a quadratic equation and it's more involved to decide which solution of the quartic equation is the one we want, we use the approximated linear expression, that is, $x = \mu g/b$ for space allocation in our experiments. The results of the experiments show that they have good accuracy.

We have made another simplification on the size of each hash table bucket entry in the analysis for ease of exposition. By using $M = \sum b_i$, we have assumed that a hash table entry has the same size for all relations in the LFTA. Actually, the size of a hash table entry for different relations can vary a lot. Suppose we use an int (4 byte) to represent each attribute or a counter. Then a bucket for relation A takes 8 bytes and a bucket for ABCD takes 20 bytes. If we denote the bucket entry size of relation i as h_i, then $M = \sum b_i h_i$. In this case, the results of the analysis are similar. Instead of allocating space proportional to $\sqrt{g}$, we should allocate space proportional to $\sqrt{g_i h_i}$. We have used such variable sized buckets in our implementation, and experimental study, discussed next.

For clustered data, collision rates should be divided by the average flow length l_a. To consider this in space allocation, we should allocate space proportional to $\sqrt{g_i h_i / l_i}$, where l_i is the average flow length of relation i.

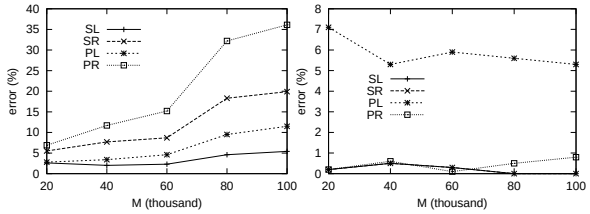
6. EXPERIMENTAL STUDY

6.1 Experimental Setup and Datasets

We prototyped this framework in C in order to evaluate the different techniques we developed. We use 4 bytes as our unit of space allocation. Each attribute value and counter we instantiate has this size. In accordance to operational streaming data managers [8], we consider M between 20,000 and 100,000 units of space (4 bytes each). The ratio of eviction cost to probe cost c_2/c_1 is is modeled as 50 in our experiments, which is also a ratio measured in operational data stream management systems [8].

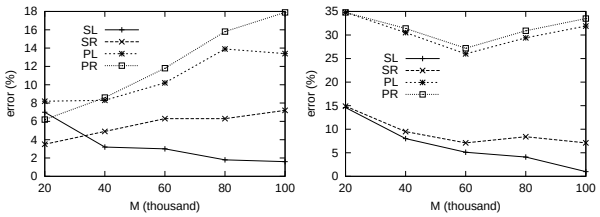
We used both synthetic and real datasets in our evaluation. The real dataset is obtained by *tcpdump* on a network server. We extracted TCP headers obtaining 860,000 records with attributes source IP, destination IP, source port and destination port, each of size 4 bytes. The duration of all these packets is 62 seconds. There are 2837 groups in this 4-attribute relation. For other relations we extracted in this way, the number of groups varies from 552 to 2836. For the synthetic datasets, we generated 1,000,000 3 and 4 dimensional tuples uniformly at random with the same number of groups as those encountered in real data. All the experiments are run on a desktop with Pentium4 2.6GHz CPU and 1GB RAM.

We adopt the following way to specify a configuration. "AB(A B)" is used to denote a phantom AB feeding A and B. We use this notation recursively. For example, the configuration in Figure 3(c) can be expressed as (ABCD(AB BCD(BC BD CD))).



(a) (ABC(AC(A C) B)) (b) AB(A B) CD(C D)

Figure 9: Comparison of space allocation schemes



(a) (ABCD(ABC(A BC(B C)) D)) (b) (ABCD(AB BCD(BC BD CD)))

Figure 10: Comparison of space allocation schemes

6.2 Evaluation of Space Allocation Strategies

Our first experiment aims to evaluate the performance of various space allocation strategies. In these experiments we derive our parameters from the real data set. Our observations were consistent across a large range of real and synthetic datasets. We vary M from 20,000 to 100,000 at steps of 20,000 and the granularity for increasing space while executing ES is set at 1% of M. In all experiments we compute the cost using Equation 7 with a suitable model for collision rate, as described below.

6.2.1 Solvable Configurations

We first experimentally validate the results of our analysis for the case of configurations for which we can analytically reason about the goodness of space allocation strategies.

For the case with no phantoms, (assuming $x = \mu g/b$ as collision rate) we compared the cost of the exhaustive space allocation (ES) with a scheme that allocates space according to our analytical expectations, namely, allocating space proportional to the square root of number of groups. For the case of real data, we tested all possible configurations with no phantoms. The cost obtained by the scheme performing space allocation as dictated by our analytical derivations incurred an error less than 1% compared to ES. The small error comes from our approximation to the collision rate, especially the value of μ, which can be different from the value the optimal solution assumes.

For the case with only one phantom feeding all queries, we use our optimal space allocation scheme derived based on the approximation of collision rate x by $\mu g/b$. We again compare the accuracy of the space allocation scheme allocating space according to our analysis, to that of ES and test all possible configurations for the case of the real data set. The average cost error (compared to ES) of our scheme is usually less than 1% and the maximum observed was 2%. Therefore even with this approximation ($x = \mu g/b$) to the collision rate, the results are still quite accurate.

6.2.2 Unsolvable Configurations

For unsolvable configurations, we evaluated several heuristics. We compared SL, SR, PL, PR as described in Section 5.2 and ES. We evaluated all possible configurations for the case of the real data set (four attributes). The relative errors of the heuristics compared to the cost of ES are shown in Figures 9 and 10 for 4 representative configurations. Related results were obtained for other configurations; all those are summarized in Table 2.

We observe that generally SL and SR are better than PL and PR. Thus, heuristics inspired by our analytical results appear beneficial. Except one case in Figure 10(a) when $M = 20,000$, SL is always the best. PL and PR can have errors as large as 35% and although SR has smaller error, it is always less accurate than SL. In Table 2, we show the average relative error of the four different heuristics compared to ES. SL is the best for all values of M.

In Table 3, we accumulate statistics in order to show in all configurations tested how frequently SL is the heuristic yielding the minimum error. We preset the percentage of configurations tested in which SL yields minimum error, as well as for the cases that SL does not yield the minimum error, how far its error is from the error of the best heuristic, on the average.

M (thousand)	20	40	60	80	100
SL (%)	6.0	3.0	2.2	3.2	2.3
SR (%)	6.2	5.3	5.3	9.0	9.4
PL (%)	15.8	14.2	14.6	21.4	23.4
PR (%)	10.1	11.4	12.4	19.7	22.7

Table 2: Average error for the four heuristics

These results (which are representative of a large set of experiments conducted) attest that SL behaves very well across a wide range of configurations. Even in the cases that it's not the best it remains highly competitive to the best solution. Therefore we would choose SL for space allocation in our algorithms.

6.3 Evaluation of the Greedy Algorithms

We now turn to the evaluation of algorithms to determine beneficial configurations of phantoms. We will evaluate the greedy algorithm GS and our proposed greedy algorithm GC. GC makes use of the SL space allocation strategy; we refer to this combination as GCSL (algorithm GC using SL space allocation). For GS, we would add space of ϕg each time a phantom is added in the current configuration under consideration until there is not enough space for any additional phantom to be considered. At this point we allocate the remaining space to relations already in the configuration proportional to their number of groups. We also consider the following method to obtain the optimal configuration cost. We explore all possible combinations of phantoms and for each configuration we use exhaustive space (ES) allocation to calculate the cost, choosing the configuration with the minimum overall cost. We will refer to this method as EPES in the sequel. Costs are computed using Equation 7 and our approximation to the collision rate.

6.3.1 Phantom Choosing Process

We first look at the query set {A, B, C, D} on a 4-dimensional uniform random dataset with M set as 40,000. Since a good value of ϕ is not known a priori, we vary it and observe the trends. Figure 11 presents the cost of the different algorithms. The costs are normalized by the cost of EPES (the optimal cost). The cost of GS first decreases and then increases, as ϕ increases. If ϕ is too small, each phantom is allocated a small amount of space, at the expense of high collision rate. On the other hand, if ϕ is too large, each

M (thousand)	20	40	60	80	100
SL being best (%)	44	89	89	89	100
Relative error from the best (%)	2.2	0.006	0.15	0.6	0

Table 3: Statistics on SL

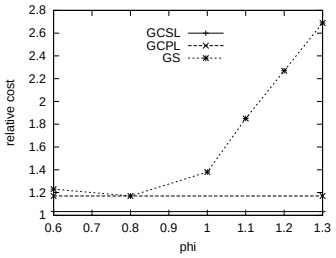


Figure 11: Comparison of phantom choosing algorithms

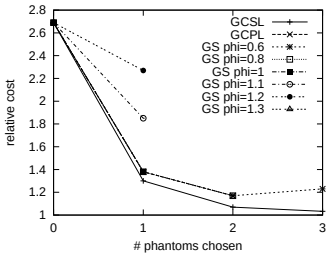
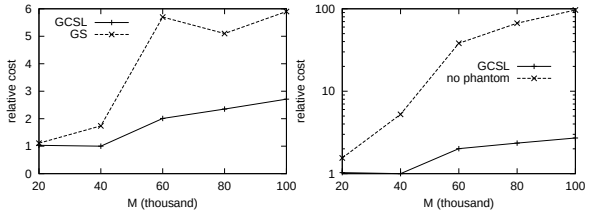
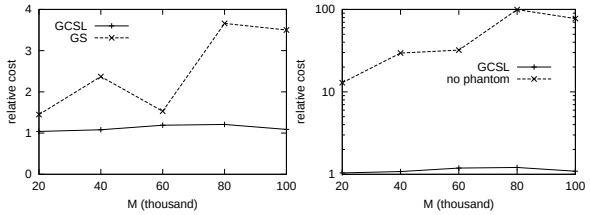


Figure 12: Phantom choosing process



(a) GCSL vs. GS (b) GCSL vs. no phantom

Figure 13: Comparison on synthetic dataset



(a) GCSL vs. GS (b) GCSL vs. no phantom

Figure 14: Comparison on real dataset

phantom has low collision rate, but each phantom takes too much space and prohibits addition of further phantoms, which could be beneficial. This alludes to a knee in the cost curve signifying the existence of an optimum value. For the GCSL algorithm, cost is lower than the cost of GS for any ϕ, because when we adjust the space allocation and calculate the cost each time a phantom is added, we are essentially adapting ϕ to a better value. The gap between the minimum point of the GS curve and GCSL is due to the space allocation scheme. Using GC in conjunction with PL space allocation, yields a curve which precisely lower bounds GS. Thus, GCSL benefits from both the way we choose phantoms and the way space is allocated in these phantoms.

Figure 12 presents the change in the overall cost in the above scenario as each phantom is chosen. We observe that the first phantom introduces the largest decrease in cost. The benefit decreases as more phantoms are added and for GS with $\phi = 0.6$, the cost goes up when adding the third phantom. Note that the third phantom added by GS with $\phi = 0.6$ is different from the third phantom added by GCSL due to the differences in space allocation. For GS with $\phi = 1.2, 1.3$ there is no space to add more than one phantom.

6.3.2 Validating Cost Estimation Framework

With our next experiment we wish to validate our cost estimation framework against the real measured errors. We implemented the hash tables and we let a uniform random dataset pass through the phantoms and queries computing the desired aggregates. The phantoms are chosen and the corresponding space allocation is conducted, using our heuristics. We count the collisions in the hash tables and calculate the true cost of this configuration. We normalize the actual cost of GCSL and GS by the actual cost of the optimal (according to our cost model) configuration obtained by EPES; the relative actual costs are shown in Figure 13(a). For GS, we tried different ϕ values, and only the one with the lowest cost at each value of M is presented in the figure.

We can see that the actual cost of GCSL is always much lower than that of GS, even we could always choose the best ϕ for GS (which is impossible in practice). When M=60,000, the cost of GCSL is as low as 26% of the cost of GS. While GS can have cost as high as 6 times the optimal cost, GCSL is always within 3 times the optimal cost.

We conducted a large set of experiments quantifying the accuracy of our estimation framework against actual measurements. In general, difference between the predictions of the cost model and the actual cost becomes large as M increases. The relative cost difference of GCSL compared to the optimal cost also increases as M increases. This is due to two factors: first when M is very large then collision rates are very small and become increasingly difficult to capture analytically. Second, for large M there are many phantom levels and as a result errors accumulate across multiple levels. However, despite certain inaccuracy, our technique results in a reasonable low cost compared to the optimal cost and outperforms GS considerably, for a variety of data sets, especially for low values of M (which is the common case in practice).

In order to validate the effectiveness of phantoms for computing multiple aggregates, we conducted the following experiment. We run the same queries without maintaining any phantoms and we compare the cost with the cost of GCSL. The results are presented in Figure 13(b). It is evident that maintaining phantoms does reduce the cost greatly (more than an order of magnitude).

6.3.3 Experiments with Real Data

We repeated our validation experiment using real data this time and the query set {AB, BC, BD, CD}. Again we let the real data set stream by the configuration we have obtained using our algorithms and report the resulting actual costs incurred. Once again actual costs are normalized by the actual cost incurred by the EPES strategy. Flow length is derived temporally.

Figure 14(a) presents the results. It is evident that GCSL outperforms GS. Once again we compare the cost of GCSL and the cost incurred without the maintenance of any phantoms. GCSL offers an improvement up to about 100 compared to the cost incurred without the use of phantoms.

6.3.4 Peak Load Constraint

The update cost at the end of epoch as described in Section 3.2.2 can be calculated according to Equation 8. This update cost must be within the peak load constraint E_p. If the update cost E_u exceeds E_p, we can use two methods to resolve it: *shrink* and *shift*. The shrink method shrinks the space of all hash tables proportionally. The shift method shifts some space from queries to phantoms

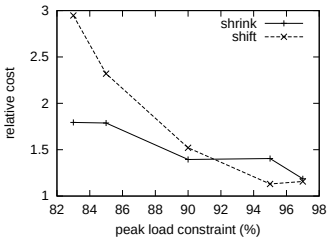


Figure 15: Peak load constraint

since c_2 is much larger than c_1 and a major part of the update cost is incurred by queries. For the real dataset and the query set {AB, BC, BD, CD}, given a space allocation, we calculate its E_u; then we set E_p to a percentage of E_u and use the two methods to reallocate space. After the reallocation, we run the data through the configuration and we compute the cost when $M = 40,000$. The results are in Figure 15. When E_p is not much smaller than E_u, the shift method performs better; while when E_u is much larger than E_p, the shrink method performs better. The reason is that when E_u is close to E_p, a small shift to reduce E_u suffices. When E_u and E_p differ by much, a major shift in space results in non optimal space allocation and thus shrink is better. Similar behavior is observed when M is set as other values.

In terms of the performance of our algorithms, the running time of GCSL in all configurations we tried was sub-millisecond; we don't expand further due to space limitations.

7. RELATED WORK

Our problem is closely related to the problem of multi-query optimization, i.e., optimizing multiple queries for concurrent evaluation. This problem has been around for a long time, and several techniques for this problem have been proposed in the context of a conventional DBMS (see, e.g., [18]). The basic idea of all these techniques is the identification of common query sub-expressions, whose evaluation can be shared among the query execution plans produced. This is also the basis for sharing of filters in pub-sub systems (see, e.g., [11]). Our technique of sharing computation common to multiple aggregation queries is based on the same idea.

Our problem also has similarities to the view maintenance problem, which has been studied extensively in the literature (see, e.g., [13]). In this context, Ross et al. [17] have studied the problem of identifying, in a cost-based manner, what additional views to materialize, in order to reduce the total cost of view maintenance. Our idea of additionally maintaining phantoms, and choosing which phantoms to maintain, to efficiently process multiple aggregations is based on the same conceptual idea.

Many papers (see, e.g., [7, 4, 5]) have highlighted the importance of resource sharing in continuous queries. [7, 16] use variants of predicate indexes for resource sharing in filters in continuous query processing systems. In the context of query processing over data streams, Dobra et al. [9] consider the problem of sharing sketches for approximate join-based processing. [6, 2] consider the problem of resource sharing when processing large numbers of sliding window aggregates over data streams. However, none of these papers proposed the maintenance of additional queries to improve the feasibility of resource sharing.

8. CONCLUSIONS

Monitoring aggregates on IP traffic data streams is a compelling application for data stream management systems. Evaluating multiple aggregations in a two level DSMS architecture is an important practical problem. We introduced the notion of phantoms (fine-granularity aggregation queries) that has the benefit of supporting shared computation. We formulated the MA optimization problem, analyzed its components and proposed greedy heuristics which we subsequently evaluated using real and synthetic data sets to demonstrate the effectiveness of our techniques.

We are currently considering deploying this framework in a real DSMS system. This raises important research questions at the system level, in terms of interaction of such algorithms with the current system, studying issues related to adaptivity and frequency of execution, etc. We hope to report such results in the near future.

9. REFERENCES

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

[2] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, 2004.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[4] D. Carney, et al. Monitoring streams - a new class of data management applications. In *VLDB*, 2002.

[5] S. Chandrasekaran, et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[6] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *VLDB*, 2002.

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, 2000.

[8] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *SIGMOD*, 2003.

[9] A. Dobra, M. N. Garofalakis, J. Gehrke, and R. Rastogi. Sketch-based multi-query processing over data streams. In *EDBT*, 2004.

[10] M. Dwass. *Probability and statistics: an undergraduate course*. W. A. Benjamin, 1970.

[11] F. Fabret, et al. Filtering algorithms and implementation for very fast publish/subscribe. In *SIGMOD*, 2001.

[12] W. Feller. *An introduction to probability theory and its applications*, volume I. John Wiley & Sons, Inc, 1968.

[13] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Data Engineering Bulletin*, 18(2), June 1995. Special Issue on Materialized Views and Data Warehousing.

[14] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, 1996.

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

[16] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[17] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *SIGMOD*, 1996.

[18] T. Sellis. Multiple query optimization. *ACM TODS*, 13(1):23–52, 1988.

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