Noname manuscript No.
(will be inserted by the editor)

Streaming Multiple Aggregations Using Phantoms

Rui Zhang · Nick Koudas · Beng Chin Ooi · Divesh Srivastava · Pu Zhou

the date of receipt and acceptance should be inserted later

Abstract *Data streams* characterize the high speed and large volume input of a new class of applications such as network monitoring, web content analysis and sensor networks. Among these applications, network monitoring may be the most compelling one – the backbone of a large Internet service provider can generate 1 petabyte of data per day. For many network monitoring tasks such as traffic analysis and statistics collection, aggregation is a primitive operation. Various analytical and statistical needs naturally lead to related aggregate queries. In this article, we address the problem of efficiently computing multiple aggregations over high speed data streams based on the two-level query processing architecture of GS, a real data stream management system deployed in AT&T. We discern that additionally computing and maintaining fine-granularity aggregations (called *phantoms*) has the benefit of supporting shared computation. Based on a thorough analysis, we propose algorithms to identify the best set of phantoms to maintain and determine allocation of resources (particularly, space) to compute the aggregations. Experiments show that our algorithm achieves near-optimal computation costs, which outperforms the best adapted algorithm by more than an order of magnitude.

Keywords Data stream · Aggregation · Multiple-query optimization · Phantom · GS

Rui Zhang
University of Melbourne
E-mail: rui@csse.unimelb.edu.au

Nick Koudas
University of Toronto
E-mail: koudas@cs.toronto.edu

Beng Chin Ooi
National University of Singapore
E-mail: ooibc@comp.nus.edu.sg

Divesh Srivastava
AT&T Labs–Research
E-mail: divesh@research.att.com

Pu Zhou
University of Melbourne
E-mail: puz@csse.unimelb.edu.au

1 Introduction

The phenomenon of data streams is real. In data stream applications, data arrives very fast and the volume 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., [4,25]). Routers forward IP packets at high 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 aggregate 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 time interval, report the total number of packets, provided this number of packets is more than 100". Thus, *monitoring aggregations 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 [2,6,8,12,31,22]. Of these DSMSs, GS (originally named Gigascope [12]) is tailored for processing high speed IP traffic data. This is, in large measure, due to GS's two layer architecture for query processing. The low level query nodes (called *LFTA*s[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 (called *HFTA*s). The HFTAs can then perform more complex processing on the reduced volume (and speed) of data obtained from LFTAs.

Various analytical and statistical needs naturally lead to related aggregate queries on data streams. They may 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". They may also differ in the length of the time window. For example, "for every destination IP, destination port and 3 minute interval, report the average packet length". An extreme case is that of the data cube, that is, computing aggregations 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 article, we address this problem of efficiently computing *multiple aggregations over high speed data streams*, based on the two-level (LFTA/HFTA) query processing architecture of GS. Our proposed technique, called **SMAP** (Streaming Multiple Aggregations using Phantoms), is motivated by the idea of maintaining *phantoms*, which are additional queries maintained for sharing computation. SMAP consists of an intelligent way to choose phantoms and an effective resource allocation scheme. Together, they minimize the overall processing cost. Our contributions are summarized below:

- Our first contribution is the insight that when computing multiple aggregate queries, additionally computing and maintaining *phantoms* may reduce the overall query processing cost due to shared computation. Phantoms are fine-granularity aggregate queries that, while not of interest to the user, allow for shared computation between multiple aggregate queries over data streams.
- Our second contribution is an investigation into the problem of identifying beneficial configurations of phantoms and user queries. We formulate this problem as a cost optimization problem, which consists of two sub-problems: how to choose phantoms and how to allocate resource (particularly, space) amongst a set of phantoms and user queries. We formally show the hardness of both sub-problems and propose heuristics for them based on detailed analyses.
- Our final contribution is a comprehensive experimental study, based on real IP traffic data, as well as synthetic data, to understand the effectiveness of our technique.

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

We demonstrate that SMAP results in near optimal configurations (within 15-20% most of the time) for processing multiple aggregations over high speed streams. It reduces processing time over the best adapted technique by an order of magnitude. 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.

This article is an extended version of our earlier paper [33]. There, we presented our technique for cases where the queries differ only in their grouping attributes. Here, we extend the technique to cases where the queries can also differ in the length of the time window. To this end, we present mechanisms to share computation among queries of different time window lengths and develop a more comprehensive cost model to accommodate the new query setting. In addition, we give a more detailed analysis of our algorithms and provide an improved collision rate model used in choosing phantoms. Finally, we have conducted more extensive experiments to demonstrate the effectiveness of SMAP for queries with different query window lengths.

The rest of the article is organized as follows. We first motivate the problem and our solution in Section 2. We then formulate our problem, present its cost model, give hardness result of the problem and summarize our technique, SMAP, in Section 3. Section 4 presents our phantom choosing algorithm. Section 5 derives a model to estimate collision rates of hash tables, which is a key component of our space allocation scheme. Section 6 analyzes space allocation schemes. Section 7 shows how to extend SMAP to handle aggregate queries with different time window lengths. Section 8 reports our experimental study. Related work is discussed in Section 9 and Section 10 concludes the article.

2 Background and Motivation

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

2.1 GS's Two Level Architecture

GS 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 MB). HFTAs are typically processed in a host machine's main memory (which can be hundreds of MB to several GB).

2.2 Single Aggregation in GS

Let us first see how a single aggregate query is processed in GS. Consider the following query Q0.

```
Q0:  select tb, SourceIP, count(*) as cnt
     from IPPackets
     group by time/5 as tb, SourceIP
```

IPPackets is a relation defined by the user to represent IP packets. Its attributes are extracted from raw network data, usually containing *SourceIP*, *SourcePort*, *DestinationIP*, *DestinationPort*, *time*, and maybe other attributes such as *PacketLength*, depending on its exact definition given by the user. The meaning of *SourceIP*, *SourcePort*, *DestinationIP* and *DestinationPort* are self-explanatory. The attribute *time* is the timestamp when the packet arrives. Without loss of generality, we assume the time unit to be a minute in this article. What the above query does is to obtain the number of packets sent from each sender every 5 minutes (that is, show each one's IP address with the number of packets sent from it).

Figure 1 is an abstracted model of processing single aggregate queries in GS. The LFTA and HFTA are denoted by M_L and M_H, respectively. When a network data stream tuple arrives, it is observed at M_L; M_L maintains a hash table consisting of a number of entries, each entry being a $\{$*group*, *count*$\}$ pair. The item *group* always stores the most recently observed values of the grouping attributes of the query. For example, in query Q0, we group the tuples by the attribute *SourceIP* for every 5 minutes, so *group* stores the most recent source IP hashing to this entry. The second item of an entry, *count*, keeps track of the number of times the values stored in *group* have been observed continuously. As a new tuple r hashes to entry *bk*, GS checks if r belongs to the same group as the existing group in *bk* . If yes, *bk*'s *count* is incremented by 1. Otherwise, we say a *collision* occurs. In this case, first the current entry in *bk* is evicted to M_H. Then, the values of r's grouping attributes are stored in *bk*'s *group* and *bk*'s *count* is set to 1. Because M_H is much larger than M_L, for the data in M_H, we can store all the groups in a big table with their counts. When the group in the evicted entry already exists in the table in M_H, we just add the count in the evicted entry to the existing count of the group in the table.

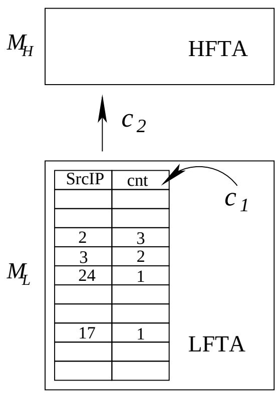


Fig. 1 Single aggregation in GS

Queries in GS are processed in an epoch by epoch fashion (that is, in tumbling time windows[2]). For example, Q0 has a query window length of 5 minutes. This means that at the end of every 5 minute window, all the entries in M_L are evicted to M_H to compute the aggregation results for this time window – this process is called the *end-of-epoch (**EE**) update* and we will discuss more about it in Section 3.2. At M_H, counts of tuples in the same group and in the same epoch are combined to compute the desired query answer.

GS 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

In GS, LFTAs are run on a Network Interface Card (NIC), which has a memory constraint. Therefore, M_L has a size constraint depending on the hardware (typically several hundred KB). M_H has much more space and reduced volume of data to process, so the processing at M_H does not dominate the total cost. The bottleneck of the processing arises from the following two types of costs:

– The cost of looking up a hash table in M_L, and possible update on the hash table in case of a collision. This whole operation, called a *probe*, has a nearly constant cost c_1.
– The cost of evicting 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 data transfer between different levels of the memory hierarchy is much more expensive than simple checks within the cache.

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. Generally speaking, a well designed hash function with a large size (within space and peak load constraints, as we will discuss more later) reduces the total cost.

[2] In the rest of the article, we use "epoch" and "time window" interchangeably.

2.4 Processing Multiple Aggregations Naively

Given the above method of processing a single aggregate query based in GS, we now examine the problem of evaluating multiple aggregate queries. Up to Section 6, we will focus on queries that only differ in their grouping attributes. We discuss how to handle queries that also differ in the length of the time window in Section 7.

Suppose the user issues the following three queries:

```
Q1: select tb, SourceIP, count(*) as cnt
    from IPPackets
    group by time/5 as tb, SourceIP

Q2: select tb, DestinationIP, count(*) as cnt
    from IPPackets
    group by time/5 as tb, DestinationIP

Q3: select tb, DestinationPort, count(*) as cnt
    from IPPackets
    group by time/5 as tb, DestinationPort
```

For simplicity, we will use R to denote the relation of the data stream tuples and A, B, C, D, etc, for R's attributes in the remainder. The above queries are re-written as follow:

```
Q1: select tb, A, count(*) as cnt
    from R
    group by time/5 as tb, A

Q2: select tb, B, count(*) as cnt
    from R
    group by time/5 as tb, B

Q3: select tb, C, count(*) as cnt
    from R
    group by time/5 as tb, C
```

A naive method is to process each query separately using the aforementioned single aggregate query processing algorithm. Specifically, we maintain, in M_L, three hash tables for A, B, and C separately as shown in Figure 2(a). For each incoming tuple, we need to probe each hash table, and if there is a collision, some entry gets evicted to M_H.

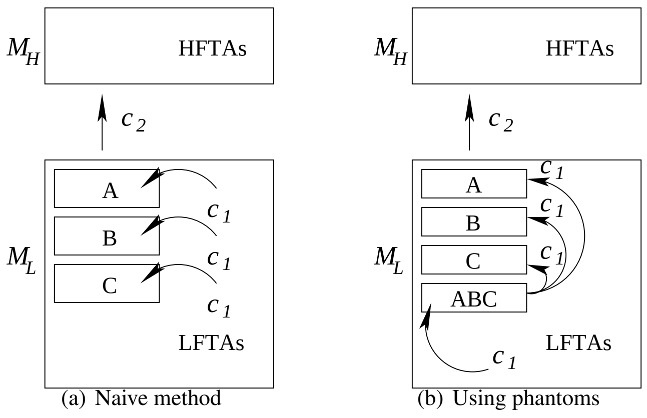


Fig. 2 Processing multiple aggregations

2.5 Processing Multiple Aggregations Using Phantoms

Since we are processing multiple aggregate queries, we may be able to share the computation that is common to some queries 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(b). The queries on A, B and C are processed as follows. When a new tuple arrives, we hash it on the combined attributes, ABC, and maintain the groups and counts of ABC in its hash table. Suppose the new tuple hashes to entry bk_1 in hash table ABC (we will refer to the hash table for a relation R simply as "hash table R" in the remainder and we may omit "hash table" when the context is clear). If the new tuple belongs to the same group as the existing group in bk_1, then we simply increase bk_1's *count* by one; otherwise, we evict the existing entry to the three hash tables A, B and C and put the new group in bk_1 with *count* one. At this moment, the entry evicted from ABC is like a new tuple arriving at A, B and C, individually. Then we process the evicted entry on each of A, B and C as in the single query case.

The intuition of using phantoms is that, when a new tuple arrives, instead of probing three hash tables A, B and C, we only probe one hash table, that is, ABC. The probes on A, B and C are delayed until the point when an entry is evicted from ABC (that is, a collision happens in ABC). Because network data usually exhibits clusteredness, that is, all packets in a flow have the same source/destination IP/port, the likelihood of a collision is low until many packets have been observed. Therefore, maintaining ABC may reduce the overall cost.

Since the aggregate 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, it may help reduce the overall processing cost. We call such a relation a *phantom*. While for A, B and C, whose aggregate information is of users' interest, we call each of them a *query*. Both *queries* and *phantoms* are *relations*.

We use the example in Figure 2 to illustrate how the maintenance of a phantom can benefit the total evaluation cost. For simplicity, suppose A, B and C have the same collision rate. Without the phantom, let their collision rate be x_1; with the phantom, let their collision rate be x'_1. In general, the larger the hash table, the smaller the collision rate. Since the hash table size of A, B and C is smaller in the presence of the phantom, x'_1 is larger than x_1. Let the collision rate of ABC be x_2.

Consider the cost for processing n tuples. Without the phantom, we need to probe three hash tables for each incoming tuple, 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 tuple and there would be x_2n evictions. For each of these evictions, we probe A, B and C, and hence there are x'_1x_2n evictions from each of them. The total cost is:

$$E_2 = nc_1 + 3x_2nc_1 + 3x'_1x_2nc_2 \tag{2}$$

The difference of E_1 and E_2 is:

$$E_1 - E_2 = [(2-3x_2)c_1 + 3(x_1 - x'_1x_2)c_2]n \tag{3}$$

If x_2 is small enough so that both $(2-3x_2)$ and $(x_1 - x'_1x_2)$ are larger than 0, then E_2 will be smaller than E_1, and therefore maintenance of the phantom benefits the total cost. If x_2 is not small enough so that one of $(2-3x_2)$ and $(x_1 - x'_1x_2)$ 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 $(x_1 - x'_1x_2)$ are less than 0, then the maintenance of the phantom increases the cost and therefore we should not maintain it. Here, we omitted the EE update cost. We will give a thorough cost analysis in Section 3.2.

2.6 Choice of Phantoms

In general, we may maintain multiple phantoms and multiple levels of phantoms. Figure 3 shows three possible ways of maintaining phantoms for the set of queries AB, BC, BD

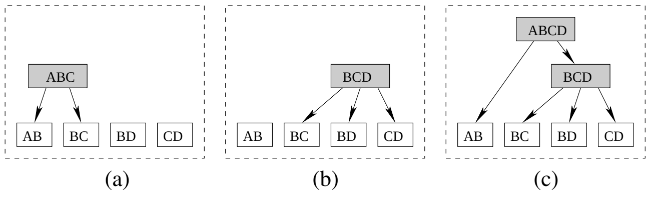


Fig. 3 Choices of phantoms

and CD. 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 shown in Figure 4. Each node in the graph is a relation and each directed edge

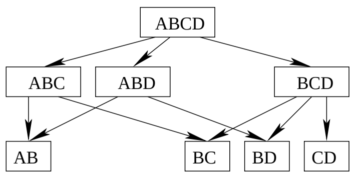


Fig. 4 Relation feeding graph

shows a *feed* relationship between two nodes. The 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 maintained.

Given a set of user queries and the relation feeding graph, an optimization problem is to identify the phantoms that we should maintain to minimize the cost. Another optimization problem is, given a set of phantoms and queries, how to allocate hash table space for them to minimize the overall cost.

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. We end this section with a synopsis of our proposed solution.

3.1 Terminology, Notation and Assumptions

When we have chosen a set of phantoms to maintain in M_L, we call the set of maintained relations (that is, the chosen phantoms and user queries) together with the feeding relationship as a *configuration*. For example, Figure 3 shows three possible configurations for the example query set, $\{$AB, BC, BD, CD$\}$. While a 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 a *leaf*. For all the configurations in Figure 3, queries AB, BC, BD and CD are leaf relations. Leaf relations may also be raw relations. For example, BD and CD are both raw and leaf relations in Figure 3(a). We adopt the following way to specify a configuration. "AB(A B)" is used to denote a phantom AB feeding queries A and B. We use this notation recursively. For example, the configuration in Figure 3(c) is expressed as ABCD(AB BCD(BC BD CD)).

We next develop our cost model, which determines the total cost incurred during data stream processing for a given configuration. We then formalize the optimization problem. Commonly used symbols are summarized in Table 1.

3.2 Cost Model

Recall that aggregate 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, the aggregate query hash tables need to

Table 1 Symbols

Symbol	Meaning
A_R	The set of ancestors of relation R in a configuration
bk	A bucket in a hash table
b	The number of bucket of the hash table
B_k	The number of buckets that k groups hash to
$\mathscr{B}_i$	Occupancy numbers
c	Some constant denoting certain cost
e	Per tuple cost
E	Overall cost
E_m	Intra-epoch cost
E_e	End-of-epoch cost
f	Number of relations a phantom feed (the fanout)
$\mathscr{F}_R$	The number of tuples fed to relation R
g_i	The number of groups of relation i for an epoch
I	A configuration
k	The number of groups hashing to a hash table bucket
l_a	Average length of network flows
L	The set of leaf relations in a configuration
M	Memory constraint in LFTA
M_R	The size of the hash table for relation R
n_e	The number of stream tuples observed in an epoch
n_Q	The number of queries
Q	A query
R	A relation
S	A set
T	The time duration of an epoch
W	The set of all raw relations
x	Collision rate
ϕ	A user defined ratio used in space allocation of the GP algorithm (see Section 4.1)

be maintained, for each tuple in the stream. At the end of an epoch, all the hash tables of the user queries at M_L are evicted to M_H to complete the user query computations. Thus, there are two components to the cost: intra-epoch (IE) cost, and end-of-epoch (EE) cost. We discuss each of these next.

3.2.1 Intra-Epoch (IE) Cost

In an epoch T, maintaining (the hash tables of) the relations in a configuration starts with updating all raw relations when a new tuple in the stream arrives. If and only if there are collisions in the raw relations, the relations they feed take the evicted entries as input and are updated. This process recurses until the leaf relations. If there are collisions in the leaf relations, the corresponding entries are evicted to M_H. We call an eviction between two relations in M_L an *internal* eviction and an eviction from M_L to M_H an *external* eviction. As described in Section 2.3, there are mainly two types of operations that contribute to the cost of the above process: *probe* and *eviction*. In case of a single query, there is no internal eviction. The evictions we talked about in Section 2.3 are actually external evictions, at a cost of c_2 each. An internal eviction incurs probes to descendent relations, which cause c_1 costs in the descendent relations.

The maintenance cost E_m for a configuration during an epoch T can be formulated as follows.

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

where I is a configuration, L is the set of all leaf relations in I, x_R is the collision rate of the hash table R and $\mathscr{F}_R$ is the number of tuples fed to R; $\mathscr{F}_R$ is derived as follows.

$$\mathscr{F}_R = \begin{cases} n_e & \text{if } R \in W \\ \mathscr{F}_p x_p & \text{else} \end{cases} \tag{5}$$

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

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

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

As n_e is determined by the data stream but not affected by the configuration, we only need to focus on the per tuple maintenance cost for a given configuration

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

In this equation, c_1 and c_2 are constants determined by the LFTA/HFTA architecture of the DSMS. Therefore, e_m is determined by the feeding relationship and collision rates of the hash tables.

3.2.2 End-of-Epoch (EE) Cost

At the end of every epoch, we perform the EE update as follows. From the raw level to the leaf level of the configuration, we scan (the hash table of) each relation and evict each entry of the relation to the child relations it feeds. Finally, we scan the leaf relations and evict each item in them to M_H. We refer to the process of evicting all entries in a relation R('s hash table) and performing corresponding updates on R's descendants as a *hash table eviction* on R or *to evict a hash table R*. We may also say *to evict a relation R*, which actually means to evict the hash table of the relation. The EE update consists of hash table evictions of all relations in a configuration in a top-down fashion. Using an analysis similar to the IE cost analysis, taking the possibilities of collisions during this phase into account, the EE cost E_e can be expressed as follows.

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

where M_R is the size of the hash table of relation R.

3.2.3 Overall Cost and Peak Load Constraint

Our final aim is to reduce the overall cost E, which is the sum of the IE cost E_m and the EE cost E_e. E_m is highly dependent on n_e; the larger the n_e the larger the E_m. E_e mainly depends on the total size of the hash tables in M_L, which is determined by the fixed memory constraint. Hence, E_e is relatively stable for an epoch irrespective of the configuration. Typically, T is several minutes and n_e is very large (over 800 thousand per minute in our collected data). E_m is much larger than E_e in our experiments even when E_m is minimized. Therefore, we will focus on minimizing E_m for now. When we consider queries of different time window lengths in Section 7, the phantoms may have much more frequent hash table evictions. We will then take E_e into account in the overall cost.

E_e may not be the major cost, however, it produces a workload peak since all the hash tables in M_L must be evicted in a very short period of time at the end of an epoch. This workload peak must be smaller than a limit, the peak load constraint E_p, to avoid any data loss. Therefore, when we minimize E_m, we should guarantee that $E_e \leq E_p$.

3.3 Problem Definition

Based on the discussion in the previous subsection, we formalize the *multiple aggregation* (MA) problem as follows.

Consider a set of aggregate queries over a data stream, $S_Q = \{Q_1, Q_2, ..., Q_{n_Q}\}$, and a memory constraint M in M_L. Determine the configuration I, of relations in the feeding graph of S_Q to maintain in M_L and also the allocation of the available memory M to the hash tables of the relations so that the overall cost E for answering all the queries is minimized, subject to the peak load constraint $E_e \leq E_p$.

The MA problem consists of two sub-problems: *phantom choosing* and *space allocation*. For any given configuration, there exists a space allocation that has the minimum E. In Section 6, we will show how to allocate the space in order to achieve the minimum E. For now, we assume that we have a function to return the minimum E given a configuration as the input. We can prove that the phantom choosing problem is NP-complete from a reduction of the set cover problem and further, we have the following theorem:

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 $\varepsilon > 0$ every polynomial time ε-approximation algorithm for the phantom choosing problem has a performance ratio of at least $n^{1-\varepsilon}$.*

Given the hardness result, we turn to heuristics (particularly, greedy algorithms) for the phantom choosing problem. A synopsis of our proposed techniques is given next.

GS has queries with tumbling windows of lengths varying from several seconds to several minutes. The MA problem needs to be solved in almost real time at the beginning of each tumbling window to fix the configuration and then GS can start receiving records. Therefore, a highly efficient algorithm is needed for our application. When a new query joins, it waits until the beginning of the next tumbling window to start being processed together with the other queries. The short wait (typically several minutes) is usually not a problem in practice.

3.4 Synopsis of Our Proposal

The MA problem has similarities to the view materialization (VM) problem [23]. 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 maintain. So one possibility is to adapt the greedy algorithm developed for VM to MA. However, there are two differences between these two problems. First, maintenance of any of the views in VM always adds to the benefit, while in MA, maintenance of a phantom is not always beneficial. Second, the space needed for maintenance 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 that fixes the hash table size and at the same time guarantees a low collision rate of the hash table to make each maintained phantom beneficial. This adapted approach, called *greedy by increasing space*, is discussed in Section 4.1. The greedy-by-increasing-space approach has several drawbacks. First, it cannot apply the optimal space allocation scheme as we will described in Section 6. Second, it depends on a good choice of a parameter (ϕ as defined later) to yield good performance, but a good value for the parameter is hard to determine in practice. Therefore we propose a new approach, called *greedy by increasing collision rates*, in Section 4.2. In this approach, we add the phantoms one by one in a cost greedy fashion to the configuration, and we always allocate all the available space to all relations in a configuration during the process of choosing phantoms. Initially, when there is a small number of relations in the configuration, the collision rates are low and hence adding phantoms reduces the cost. As more and more phantoms are added, the hash table sizes become smaller and collision rates become higher. The algorithm stops when no phantom candidate can be added with a reduction to the cost. This strategy does not need any tuning parameter, and is amenable to an optimal space allocation scheme.

In summary, our proposed technique, called SMAP (Streaming Multiple Aggregations using Phantoms), consists of an intelligent phantom choosing algorithm (presented in Section 4) and an effective space allocation scheme (presented in Section 6). Together, they minimize the overall cost. The space allocation scheme requires an accurate model to estimate collision rates, investigated closely in Section 5.

4 Phantom Choosing

In this section, we present two phantom choosing strategies, both being greedy algorithms. The greedy-by-increasing-space approach is adapted from the algorithm for the VM problem [23]. The greedy-by-increasing-collision-rates approach is our proposed algorithm.

4.1 Greedy by Increasing Space

To adapt the greedy algorithm for the VM problem, we need to have a space allocation scheme that fixes the hash table size and at the same time guarantees a low collision rate of the hash table to make each maintained phantom beneficial. Generally, the more space allocated to a hash table, the lower 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 of a relation and b be the number of buckets in a hash table. A straightforward way of allocating hash table space to a relation is in proportion to the number of groups in the table so as to make all the hash tables have similar collision rates. Specifically, we can allocate space ϕg for a relation with g groups, where ϕ is a constant chosen by the user and can be tuned. 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 rates in Section 5. We can then have a better sense of what value of ϕ might be good according to the analysis there.

We need to know the number of groups of the relations for the coming epoch in order to allocate space to them. It is reasonable to assume that the data distribution does not change dramatically in a very short time. Hence, we maintain the number of groups of all the relations in the feeding graph in a tumbling window fashion and the previous window's number of groups of a relation is used to estimate the number of groups of the relation in the coming epoch. The numbers of groups can be maintained efficiently by a combination of sampling and sketching techniques [1].

Figure 5 shows the greedy algorithm by increasing space, named *GP* (not "GS" in order to avoid confusion with the GS data stream system). GP runs as follows. Initially, the configuration I only contains the queries, which must be maintained in M_L. For each candidate phantom in the candidate set S_C, we calculate its benefit (the difference between the costs of the configurations with and without this phantom) according to the cost model. Then we divide the benefit by relation R's number of groups $R.g$, to get the benefit per unit space for R. In the algorithm, $\mathscr{M}$ denotes the size of the remaining memory and β denotes the benefit per unit space.

```
Algorithm **GP**
1   choose a $\phi$ value;
2   $I \leftarrow S_Q$; $S_C \leftarrow \{R \in relations\ in\ the\ feeding\ graph \wedge R \notin S_Q\}$;
3   $\mathscr{M} \leftarrow M - \sum_{R \in S_Q} \phi R.g$
4   $R_m \leftarrow 1$
5   while $\mathscr{M} > 0$ && $S_C \neq \emptyset$ && $R_m \neq NULL$
6       $\beta \leftarrow -\infty$, $R_m \leftarrow NULL$
7       for each $R \in S_C$
8           if $\mathscr{M} > \phi R.g$ && $(cost(I) - cost(R \cup I))/(\phi R.g) > \beta$
9               $\beta \leftarrow (cost(I) - cost(R \cup I))/(\phi R.g)$
10              $R_m \leftarrow R$
11      if $R_m \neq NULL$
12          $I \leftarrow R_m \cup I$
13          $S_C \leftarrow S_C - R_m$
14          $\mathscr{M} \leftarrow \mathscr{M} - \phi R_m.g$
15  return $I$
**End GP**
```

Fig. 5 Algorithm **GP**

We always choose the phantom with the largest benefit per unit space to be added to I. This process continues until either the space is exhausted, or all candidate phantoms are added to I. When the algorithm stops, I contains the relations we should maintain.

This approach has two drawbacks: (i) Parameter ϕ needs to be tuned to find the best performance, but in practice, a good choice is very hard to achieve. (ii) 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.

4.2 Greedy by Increasing Collision Rates

We propose a different greedy algorithm for space allocation. 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 6). As each new phantom is added to a configuration, what changes is not the total space used, but the collision rates of hash tables. The collision rates increase as more phantoms are added.

Figure 6 shows the greedy algorithm by increasing collision rates, named *GC*, which runs as follows. At first, the configuration I only contains all the queries. For each candidate phantom in the candidate set S_C, we calculate its benefit (the difference between the costs of the configurations with and with out this phantom) according to the cost model, but different from algorithm GP which allocates $\phi R.g$ space, here we allocate space according to the scheme devised in Section 6. We always choose the phantom with the largest benefit to be added to I. Note that here we do not use benefit per unit space as the phantom choosing criterion because the effect of the space used by the phantom is already reflected by the cost. If the phantom needs too much space, it

```
Algorithm **GC**
1   $I \leftarrow S_Q; S_C \leftarrow \{R \in relations\ in\ the\ feeding\ graph \wedge R \notin S_Q\}$
2   $\beta \leftarrow 1$
3   while $\beta > 0$ && $S_C \neq \emptyset$
4       $\beta \leftarrow 0, R_m \leftarrow NULL$
5       for each $R \in S_C$
6           if $cost(I) - cost(R \cup I) > \beta$
7               $\beta \leftarrow cost(I) - cost(R \cup I)$
8               $R_m \leftarrow R$
9       if $\beta > 0$
10          $I \leftarrow R_m \cup I$
11          $S_C \leftarrow S_C - R_m$
12  return $I$
End GC
```

Fig. 6 Algorithm **GC**

would reduce the space that other relations have and hence has a negative effect on the overall cost. This process continues until no remaining phantom produces a positive benefit, or all phantoms are maintained. When the algorithm terminates, I contains the relations we should maintain.

A prerequisite of this algorithm is an accurate model to estimate collision rates. We derive such a model next.

5 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 tuple. We first consider uniformly distributed data, and then take clusteredness of data into account.

5.1 Randomly Distributed Data

Let g be the number of groups of a relation and b the number of buckets in the hash table. We say that a bucket has k groups if k groups hash to the bucket. Let B_k be the number of buckets having k groups. If the tuples in the stream are uniformly distributed, each group has the same expected number of tuples, denoted by n_{rg}. Then $n_{rg}k$ tuples will go to a bucket having k groups. Under the random hashing assumption, the collision rate in this bucket is $(1-1/k)$. Therefore $n_{rg}k(1-1/k)$ collisions happen in this bucket. The collision rate is obtained by summing all the collisions and then dividing the sum by the total number of tuples as follows.

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

Here k values start from 2 because when 0 or 1 group hashes to a bucket, no collision happens. In order to compute x, we still need to know B_k. Deriving B_k concerns a class of problems called the *occupancy problem*; B_k can be estimated by the "multinomial allocations" model [5] (Chapter 6.2). We sketch the derivation below. The probability of k groups out of g hashed to a given bucket is

$$\binom{g}{k}(1/b)^k(1-1/b)^{g-k} \tag{10}$$

This holds for any bucket, which means that each bucket has the chance of Equation 10 to have k groups. If we assume that all b buckets are independent of each other, then statistically there are

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

buckets each of which has k groups. Substitute Equation 11 for B_k in Equation 9 we have

$$x = \frac{b\sum_{k=2}^{g}\binom{g}{k}(1/b)^k(1-1/b)^{g-k}(k-1)}{g} \tag{12}$$

Equation 12 can be further simplified to the following equation.

$$x = 1 - \frac{b}{g} + \frac{b}{g}(1-\frac{1}{b})^g \tag{13}$$

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 of each other (they must satisfy the equation $\sum_{k=1}^{b} B_k = b$).

5.2 Validation of Collision Rate Model

We have measured experimentally the collision rates on both synthetic random data sets and real data sets. The results on the synthetic and real data sets are shown in Figure 7 and Figure 8, respectively.

The real data sets are extracted from the TCP data set as described in Section 8.1. We have assumed independent data distribution for the above analysis, while the TCP data set has a lot of clusteredness due to 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 tuple, eliminating the effect of clusteredness (We consider clusteredness in the next subsection). After eliminating clusteredness of the data, we extracted 4 data sets which have 1, 2, 3 and 4 attributes respectively. The number of groups in these data sets are 552, 1846, 2117, 2837 respectively. For the synthetic data sets, we generated data sets which have 500, 1000, and 2000 groups respectively. All multi-attribute random data sets have the same data distribution and hence the collision rates are the same as a one-attribute random data set. Therefore, we do not specify number of attributes in Figure 7. The

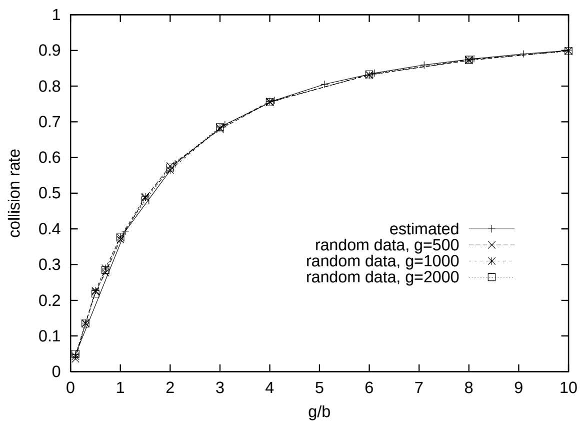


Fig. 7 Collision rates of random data

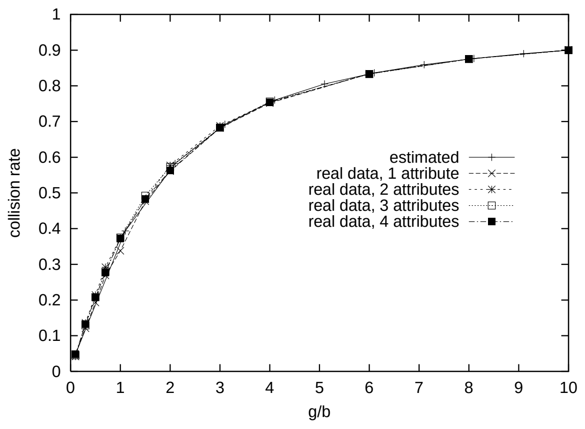


Fig. 8 Collision rates of real data

estimated curve is plotted according to Equation 13. As we can see, the observed collision rates of both the random and the real data match the estimated collision rate very well. In all the observed collision rates, more than 95% of the experimental results have less than 5% difference from the estimated collision rates. These results validate the accuracy of the collision rate model on randomly distributed data.

5.3 Clustered Data

Real data streams, especially the packets in TCP header data, exhibit strong clusteredness. The packets in the same traffic flow have exactly the same values for attributes such as source/destination IP/port. To analyze collision rates for such clustered distributions, we should consider what happens at the per flow level. If we ignore the fact that flows may interleave, and assume that packets in a flow go through a bucket without any collision until the end of the flow, then we can think of each flow as one tuple and use the same formula as in the random distribution (Equation 9) to calculate the total number of collisions as follows:

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

where n_{fg} is the number of flows in each group; B_k is still calculated by Equation 11. To obtain the collision rate, we divide n_c by the total number of tuples, $gn_{fg}l_a$, 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.

$$\begin{aligned} x &= \frac{b\sum_{k=2}^{g}\binom{g}{k}(1/b)^k(1-1/b)^{g-k}(k-1)}{gl_a} \\ &= (1-\frac{b}{g}+\frac{b}{g}(1-\frac{1}{b})^g)/l_a \end{aligned} \tag{15}$$

The difference of the collision rate for clustered data from that for random data is a linear relationship over l_a. The collision rate model for random data is a special case of that for clustered data with $l_a = 1$.

We have assumed that flows do not interleave each other in the above derivation, but in reality they do interleave. Here, we can think of l_a as the “effective” average flow length rather than the “real” average flow length. This parameter captures the effect of both the flow length and the interleaving of flows at the same time, which is just what we actually need to estimate collision rates for clustered data. Therefore, interleaving of flows is not a problem because its effect is captured together with flow lengths by l_a. Next, we provide experimental results to validate the collision rate model in Equation 15.

We have done the following experiments. We let real data sets (with clusteredness) actually run through hash tables of a configuration and record the number of collisions of a relation, so that we can compute the actual collision rate, denoted by x_a, for each relation. Then we divide the collision rate computed using Equation 13 by x_a to obtain the *measured average flow length*, l_m. Figure 9 shows l_m of different relations in the configuration ABCD(AB BCD(BC BD CD)) as a function of g/b. The raw relation ABCD has an l_m in the range of 20 to 80, while the other relations have l_m in the range of 1 to 8. The reason is that the raw relation “absorbs” most of the clusteredness, and there is not much clusteredness remaining to lower levels of relations. If we define raw relation as depth 1 in the configuration tree, then AB and BCD are depth 2 relations, and BC, BD and CD are depth 3 relations. We can also observe that relations of the same depth have very similar l_m. Relations of depth greater

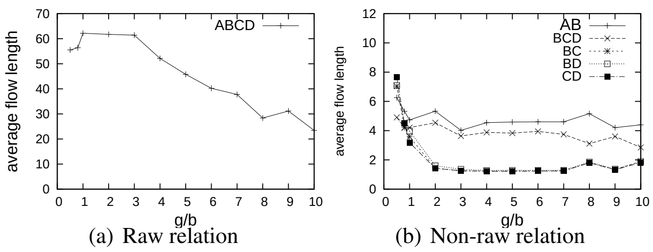
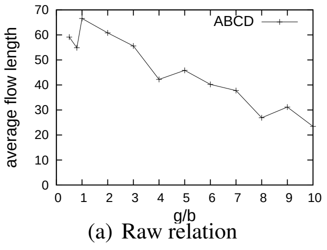
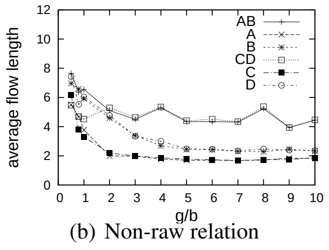


Fig. 9 Measured average flow length, ABCD(AB BCD(BC BD CD))

than or equal to 3 have l_m close to 1. Another observation is that l_m decreases as g/b increases. This is because when b gets smaller (which makes g/b larger), it gets more likely that two flows hash to the same bucket and hence interleave, which makes the effective average flow length smaller. For relations with small l_m, their l_m decreases less because a small l_m means the flows are interleaved heavily already, and interleaving them more will not reduce l_m much. Tests on many other configurations have shown very similar behavior. Figure 10 shows the result of another configuration ABCD(AB(A B) CD(C D)).



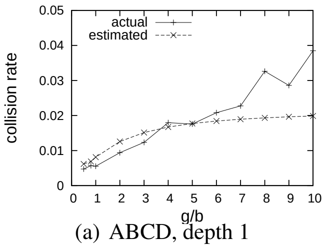
(a) Raw relation



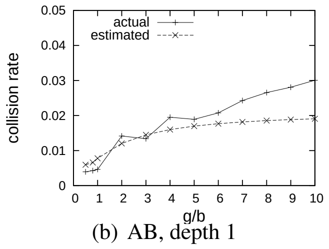
(b) Non-raw relation

Fig. 10 Measured average flow length, ABCD(AB(A B) CD(C D))

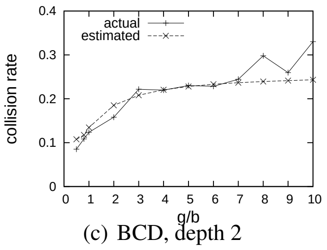
Based on the above observations, we use the following method to obtain l_a. We measure l_m for each relation in each time window. The average l_m of all the relations of depth i ($i = 1,2,3,...$) in the previous time window is used as an estimate of l_a for the relation of depth i in the current time window. Then we use Equation 15 to estimate the collision rates of the relations for the current time window. Figure 11 shows the results of four representative relations with various depths. Overall, our collision rate model provides reasonably accurate results. The accuracy is especially high (relative errors mostly within 10%) in the low collision rate parts, which are the cases occurring in optimal configurations. For the parts with higher collision rates, the accuracy gets worse. However, the errors are still within 80%, which is acceptable given the hard-to-predict nature of data stream distributions. The experimental study presented later shows that our algorithm using the above collision rate model achieves significant performance gain in practical settings despite the estimation errors.



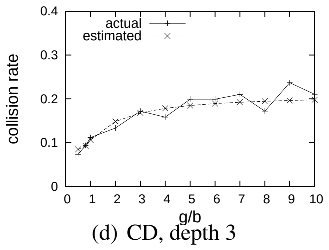
(a) ABCD, depth 1



(b) AB, depth 1



(c) BCD, depth 2

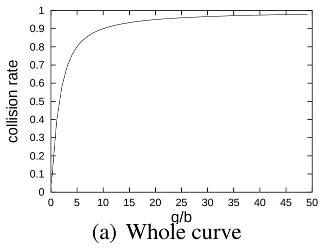


(d) CD, depth 3

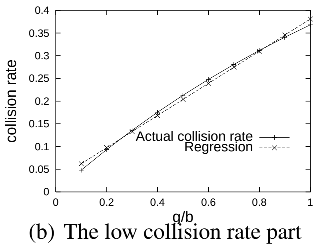
Fig. 11 Collision rates of clustered data

5.4 Approximating the Low Collision Rate Part

We can plot the collision rate model for random data (Equation 13) as a function of g/b, which is shown in in Figure 12(a). According to our previous analysis, the hash table must have a low collision rate if we want to benefit from maintaining phantoms, so 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 12(b). We observe that this part of the curve is almost a straight line and the linear regression achieves an average error less than 5%. The linear function for this part is



(a) Whole curve



(b) The low collision rate part

Fig. 12 The collision rate curve

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

For clustered data,

$$x = 0.0267/l_a + 0.354 \cdot g/(l_a \cdot b) \tag{16}$$

Expressing this part of the collision rate linearly is important for the space allocation analysis as we will see in the next section.

6 Space allocation

In this section, we investigate the problem of space allocation, that is, given a configuration, how to allocate the available space M to the relations in the configuration so that the overall cost is minimized. We start with a simple two-level configuration, and then identify the difficulties in analyzing more complex configurations. Given that, we finally discuss heuristics for space allocation.

6.1 A Case of Two Levels

Consider 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, and x_1, x_2, $\ldots$, x_f be the collision rates 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 5.4, this part of the curve can be expressed as a linear function $x = \alpha + \mu g/(l_a \cdot b)$, where $\alpha = 0.0267/l_a$ and μ=0.354.[3] Since α is small, here we make a further approximation to let $x = \mu g/(l_a \cdot b)$. We will discuss later how the results are affected when we consider α. Both g and l_a are known constants for each relation, therefore we set the value of g to g/l_a to simplify the symbols in our derivation, and we should recall that g is actually g/l_a in the end of the derivation. Given the approximation and simplified notation, $x_i = \mu g_i/b_i$, $i = 0,1,...,f$. The total size is M, so $M = \sum_{i=0}^{f} b_i$. The cost of this configuration is

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

The cost 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. We calculate the partial derivative of e over b_i, $i = 1,2,...f$, as follows.

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

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

$$\frac{g_1}{b_1^2} = \frac{g_2}{b_2^2} = ... = \frac{g_f}{b_f^2} \tag{19}$$

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

$$\mu c_2 M \nu^2 - 2\mu c_2 \sum_{i=1}^{f}\sqrt{g_i}\,\nu - fc_1 = 0 \tag{20}$$

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

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

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

$$b_i = \frac{\sqrt{g_i}}{\nu} = \frac{\mu c_2 M\sqrt{g_i}}{\mu c_2\sum_{j=1}^{f}\sqrt{g_j} + \sqrt{\mu^2 c_2^2(\sum_{j=1}^{f}\sqrt{g_j})^2 + f\mu c_1 c_2 M}} \tag{21}$$

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

$$b_0 = M - \sum_{i=1}^{f} b_i = M - \frac{M\sum_{j=1}^{f}\sqrt{g_j}}{\sum_{j=1}^{f}\sqrt{g_j} + \sqrt{(\sum_{j=1}^{f}\sqrt{g_j})^2 + \frac{fc_1 M}{\mu c_2}}} \tag{22}$$

$\frac{\partial e}{\partial b_i} = 0$ is a necessary condition of e taking the maximum or minimum. From the above analysis, we can see that there is only one solution satisfying the necessary condition. It is easy to test that a different group of b_i values gives larger e than the e given by the b_i values in Equations 21 and 22. Now, to prove that b_i values in Equations 21 and 22 give the minimum of e, it is enough to show that the minimum does not occur at the boundary of the domain of the variables $b_0, b_1, ...b_f$. Variables b_i satisfies that $0 < b_i < M$, $i = 0,1,...,f$ and $M = \sum_{i=0}^{f} b_i$. The boundary of the domain has at least one b_i approaching 0 from the right. When any of

[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 variations in their values do not affect the result much.

the b_i approaches 0 from the right, e approaches positive infinity. Therefore, the minimum does not occur at the boundary of the domain, and hence the b_i values in Equations 21 and 22 give the minimum of e. Here, we view b_i as real numbers, but in reality, they are integers, which may not take the exact values given by Equations 21 and 22. However, as M is a very large integer, typically 10,000 to 100,000, b_i is usually of the magnitude of thousands. The fractional parts of the exact values calculated from Equations 21 and 22 are negligible (less than 1/1000 of the actual value). As a result, we can still safely use Equations 21 and 22 to obtain the b_i values to get the minimum e. As tested in the experimental study, the cost of using this space allocation scheme always has less than 1% difference from the optimal cost for two-level configurations.

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

6.2 Other Cases

We have performed a similar analysis on cases with more than two levels of relations. Even a simple case of three-level configuration results in an equation of order 8. Equations of order higher than 4 cannot be solved by radicals according to Galois' Theory, that is, we do not have a closed form solution for the equation. In addition, because the coefficients are parameters which can take wide range of values that we do not know in advance, we cannot determine whether the equation is solvable in advance. More complex multiple-level configurations result in even higher order equations which cannot be solved algebraically. Therefore, we call configurations with three or more levels ***intractable*** *configurations* and propose heuristics to decide space allocation for them based on the analysis results we have for the two-level configuration. Accordingly, we call ***tractable*** *configurations* those for which we have closed-form solutions. Equations of order higher than 4 can be solved numerically. However, the equation system may have very high orders and also be very complicated. It may have multiple solutions, and it is not guaranteed that the equation system always converges quickly to solutions. Our application requires a near real-time response algorithm. Therefore, we do not use numerical methods in our solution.

We have only considered one two-level case in Section 6.1. The results of other one- or two-level cases are summarized as follows.

1. The configuration has no phantom but only leaf relations, that is, a one-level case. This case has a closed-form solution. To achieve minimum cost, allocate space proportional to the square root of the number of groups.
2. The configuration has one phantom, but it does not feed all queries, that is, some queries are fed from the stream directly. This is a two-level case, and it results in an equation of order 6, which is intractable.
3. The configuration has two phantoms, each of which feed some queries but they do not feed each other. Together they feed all queries. This is also a two-level case and it results in an equation of order 8, which is intractable.

In short, only the case with no phantom or the case with one phantom feeding all queries is tractable. All other cases are intractable configurations.

6.3 Heuristics

For intractable configurations, we propose heuristics to allocate space based on the analysis of the tractable cases and partial results we can get from the intractable cases. Our experimental study shows that our proposed heuristics based on the analysis are very close to optimal and are better than other heuristics.

In the analysis on the two-level case in Section 6.1, we observe from Equation 19 that the square of the number of buckets assigned to a leaf relation, b_i^2, should be proportional to the number of groups of that relation, g_i. From the analysis on the three-level case, we observe similar behavior on relations at the same level, that is, b_j of relations at the same level is proportional to $\sqrt{g}_j$. However, b_1^2 (R_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. From these observations, we gain the intuition that the number of buckets allocated to a relation at the same level should be proportional to the square root of its number of groups; moreover, a relation that feeds other relations should be allocated more space according to the number of groups of the relations it feeds. With this intuition, we propose the following space allocation scheme (heuristic 1).

Heuristic 1: Supernode with Linear Combination (SL). We only have the optimal solution for the two-level case, but not for a general case with more levels of relations in the configuration. We introduce the concept of ***supernode*** so that we can use the two-level space allocation scheme recursively to solve the general case. Starting from the leaf level, each relation right above the leaf level together with all its children is viewed as a supernode. The number of groups of this supernode is the sum of the number of groups of the relations that compose this supernode, that is, the phantom and the queries fed by the phantom. Then we view the supernode as a leaf node and do the above compaction recursively until the configuration contains only two levels. For a two-level configuration, we can allocate space optimally according to the analysis of Section 6.1. After the first space allocation, each node (some may be supernodes) is assigned some space. Then, we decompose each supernode to a two-level configuration and allocate the space of this supernode to all nodes optimally inside the supernode again, that is, allocate space proportional to the square root of the number of groups. We do this decomposition recursively until all supernodes are fully unfolded.

Figure 13 shows an example of how heuristic SL works. Figure 13(a) is a configuration needing space allocation. The configuration has four queries: A, B, C, D, and three phantoms: AB, ABC, ABCD. The number of groups of each relation is written beside the relation. According to heuristic SL, relations A, B and AB are first combined into a supernode AB', whose number of groups is the sum of the number of groups of A, B and AB. The first combination results in Figure 13(b). We still have more than two levels, so we continue to combine AB', C and ABC to a supernode, which results in a the two-level configuration as shown in Figure 13(c). Now we can allocate space according to the analysis on the two-level configuration and unfold the supernodes one level after another until we get back the original configuration.

We also try another heuristic described below (heuristic 2) which is the same as SL except the way we compute the number of groups of the supernode.

Heuristic 2: Supernode with Square Root Combination (SR). Since in the two-level case we see that the space should be proportional to the square root of the number of groups, we may also let the *square root* of the number of groups of the supernode be the sum of the *square roots* of the number of groups of all its relations. The other steps are the same as SL.

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 seemingly straightforward heuristics (heuristics 3 and 4) 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 obtain the optimal solution for space allocation of general cases through analysis, 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, suppose the configuration has three relations, AB, A and B, where AB feeds A and B. The 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 **exhaustive space allocation (ES)**. Obviously this strategy is too expensive to be practical, but we use it in our experiments to comparatively study the four heuristics.

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 in the experimental study.

6.4 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 using $x = \alpha + \mu g/b$. 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 is 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 experimental study shows that even with this approximation, the results are still very accurate.

We have also tried other alternatives of approximating Equation 13 such as: (1) $(1-\frac{1}{b})^g \approx e^{-\frac{g}{b}}$; (2) taking the first few terms of the binomial expansion. However, for alternative (1), the result is an even more complicated equation system, which is of higher degree and has variables in exponents. For alternative (2), if we take the first 3 terms of the binomial expansion, it's nearly a linear approximation, but with much worse accuracy; if we take more terms of the binomial expansion, the result is a much more complicated equation system than using our linear approximation and the accuracy is no better than our linear approximation.

Another simplification we have made is on the size of each hash table bucket in the analysis for ease of exposition. By using $M = \sum b_i$, we have assumed that a hash table bucket has the same size for all relations in the M_L. Actually, the size of a hash table entry for different relations can be different. Suppose we use an `int` (4 bytes) 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 size of relation i as h_i, then $M = \sum b_i h_i$. With this, the results of the analysis are similar, but instead of allocating space proportional to $\sqrt{g}$, we should allocate space pro-

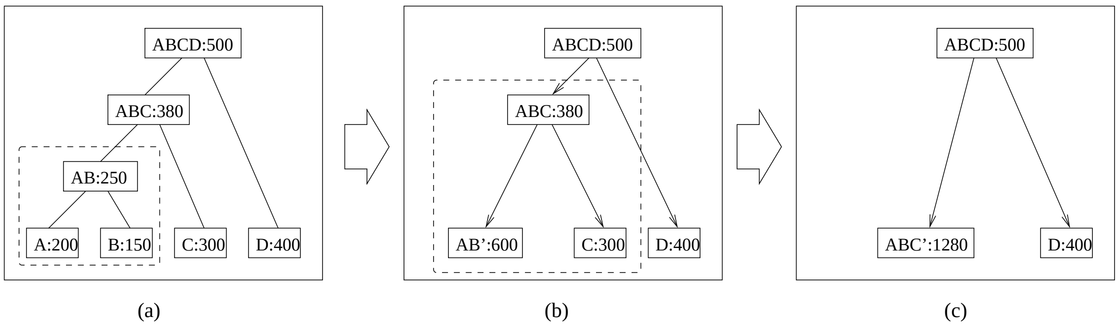


Fig. 13 Heuristic SL

portional to $\sqrt{g_i h_i}$. We have used such variable sized buckets in our implementation for experimental study.

7 Queries of Different Epoch Lengths

In this section, we extend our SMAP technique to multiple aggregate queries with different epoch lengths. Consider the following set of queries:

```
Q1: select tb, A, count(*) as cnt
    from R
    group by time/2 as tb, A

Q2: select tb, B, count(*) as cnt
    from R
    group by time/3 as tb, B

Q3: select tb, C, count(*) as cnt
    from R
    group by time/5 as tb, C
```

The epoch lengths for A, B, and C are 2, 3, and 5 minutes, respectively. To take advantage of phantoms for sharing computation, we have to solve the problem of different epochs. We can still use the phantom choosing algorithms presented in Section 4, but need to adapt the cost model and space allocation scheme to this new problem setting.

Next, we analyze what changes we have to make for the SMAP technique to work. The possible phantoms and the relation feeding graph for the aforementioned set of queries are shown in Figure 14(a). Consider the configuration in Figure 14(b). Hash table A must be evicted every 2 minutes to answer Q1, which requires hash table AB to be evicted at least every 2 minutes. Similarly, hash table B (for Q2) requires hash table AB to be evicted at least every 3 minutes. Therefore, hash table AB must be evicted at the end of timestamps 2, 3, 4, 6, 8, 9, ..., that is, all multiples of 2 and 3. To satisfy the hash table eviction needs of AB and C, hash table ABC must be evicted at all timestamps when hash table AB or C is evicted, that is, all multiples of 2, 3 and 5. In general, it is not hard to prove the following proposition:

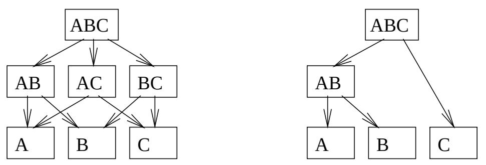


Fig. 14 Example

Proposition 1 *A relation ('s hash table) needs to be evicted at timestamps of all multiples of the epoch lengths of all its descendent leaf relations.*

As a result, hash table evictions tend to be more frequent as more queries with different epoch lengths share phantoms. This causes much higher EE costs which may no longer be negligible compared to the IE cost (recall that we have focused on minimizing only the IE cost so far after the discussion in Section 3.2.3). In addition, the EE costs now can vary a lot depending on the configuration. Therefore, we update the cost model presented in Section 3.2 to take into account the EE cost next.

7.1 Cost Model for Queries of Different Epoch Lengths

Given a configuration, the timestamps for all hash table evictions exhibit a cyclic behavior, the cycle length being the least common multiple (LCM) of the epoch lengths of all the leaf relations. For the example of Figure 14 (b), the LCM of the epoch lengths of all the leaf relations is 30. Within a cycle of 30 minutes, hash tables A, B and C are evicted every 2, 3 and 5 minutes, respectively; hash table AB is evicted at timestamps 2, 3, 4, 6, ..., 28, 30; hash table ABC is evicted at timestamps 2, 3, 4, 5, 6, ..., 28, 30. At the end of 30 minutes, if we start timestamping from 0 again, the same eviction pattern will happen in the new cycle. For a given set of queries, different configurations have the same cycle length, which is determined by the epochs of the queries. Finding the configuration with the best overall cost over all time is equivalent to finding the configuration with the best overall cost in a cycle. In what follows, we derive the per cycle overall cost, which consists of the cycle-intra-epoch cost and cycle-end-of-epoch cost. Let w_i be the length of the epoch for Q_i, where $1 \le i \le n_Q$, and γ be the length of the cycle. Then $\gamma = lcm(\{w_1, w_2, ..., w_{n_Q}\})$, where $lcm()$ is a function that returns the LCM of a set of numbers. We summarize the additional symbols used in this section in Table 2.

Table 2 Symbols

Symbol	Meaning
D_R	The set of all descendent relations of R
E_c	Overall cost in a cycle
E_{cee}	Cycle-end-of-epoch cost
E_{cie}	Cycle-intra-epoch cost
E_R	The cost of a hash table eviction on relation R
$lcm()$	A function that returns the LCM of a set of numbers
L_R	The set of descendent leaf relations of R
n_c	The number of stream tuples observed in a cycle
n_R	The number of hash table evictions of R
n_{R_i}	The number of hash table evictions of R_i
w_i	The length of the epoch of Q_i
γ	The length of a cycle

7.1.1 Cycle-Intra-Epoch (CIE) Cost

The intra-epoch cost depends on the feeding relationship, collision rates and the data stream. It is not affected by the new problem setting (different epoch lengths). Therefore, we can still use Equation 7 to obtain the CIE cost as follows.

$$E_{cie} = n_c e_m \tag{23}$$

where n_c denotes the number of stream tuples observed in a cycle. We can estimate n_c through the statistics maintained in the previous time window as discussed in Section 4.1.

7.1.2 Cycle-End-of-Epoch (CEE) Cost

In a cycle, a relation R may perform multiple EE updates, one at the end of each epoch of R. Every EE update of R entails a hash table eviction on R. The CEE cost is the sum of all the hash table eviction costs of all relations in a cycle.

Let L_R denote the set of descendent leaf relations of R in a configuration. According to Proposition 1, every $Q_i \in L_R$ causes γ/w_i hash table evictions on R. To obtain the number of hash table evictions of R in a cycle, if we just add up γ/w_i for all Q_i's in L_R, we will count the hash table evictions at the timestamps of common multiples of w_i's multiple times. Consider a configuration with only two queries, A with the epoch of 2 and B with the epoch of 3. One phantom AB feeds A and B; hence the cycle is 6. Relation A causes AB to evict three times (at timestamps 2, 4 and 6) while B causes AB to evict twice (at timestamps 3 and 6). Simply adding three by two would count AB's eviction at timestamp 6 twice. The double-counting happens at the timestamp of common multiple of 2 and 3, that is, 6. Therefore, we should subtract the number of common multiples of 2 and 3 in a cycle (that is, 1) from the sum of three and two. Then, we obtain the actual number of hash table evictions of AB, 4. We can summarize the above calculation as $\gamma/w_A + \gamma/w_B - \gamma/lcm(w_A, w_B)$, where w_A and w_B are the epoch lengths of A and B, respectively.

Further consider the example in Figure 14 (b), which has the parameters $\gamma = 30, w_A = 2, w_B = 3, w_C = 5$. If we use the above method to calculate the number of hash table evictions of ABC, we have $\gamma/w_A + \gamma/w_B + \gamma/w_C - \gamma/lcm(w_A, w_B) - \gamma/lcm(w_B, w_C) - \gamma/lcm(w_A, w_C) = 21$. But if we actually list all the timestamps for the hash table evictions of ABC, there are 22 of them. The miscalculation happens at timestamp 30, which is $lcm(w_A, w_B, w_C)$. We counted it three times by adding $\gamma/w_A, \gamma/w_B, \gamma/w_C$ and also discounted it three times by subtracting $\gamma/lcm(w_A, w_B), \gamma/lcm(w_B, w_C), \gamma/lcm(w_A, w_C)$. Therefore, we should add $\gamma/lcm(w_A, w_B, w_C)$ back, which results in 22. This calculation is summarized as $\gamma/w_A + \gamma/w_B + \gamma/w_C - \gamma/lcm(w_A, w_B) - \gamma/lcm(w_B, w_C) - \gamma/lcm(w_A, w_C) + \gamma/lcm(w_A, w_B, w_C)$.

Let n_R denote the number of hash table evictions of R in a cycle. Following the above discussions, it is not hard to prove that in general

$$n_R = \sum_{s \subseteq L_R, s \neq \emptyset} (-1)^{|s|+1} \frac{\gamma}{lcm(\{w_i | Q_i \in s\})} \tag{24}$$

where s denotes a nonempty subset of L_R.

Let E_R denote the cost of a hash table eviction on relation R. Using a very similar analysis to that in Section 3.2.1, E_R can be expressed as follows.

$$E_R = \sum_{r \in D_R} \mathscr{F}_r c_1 + \sum_{r \in L_R} \mathscr{F}_r x_r c_2 \tag{25}$$

where D_R denotes the set of all descendent relations of R, x_r is the collision rate of the hash table r, and $\mathscr{F}_r$ is the number of tuples fed to r when evicting all entries from R; $\mathscr{F}_r$ is derived as follows.

$$\mathscr{F}_r = \begin{cases} b_R & \text{if the parent relation of } r \text{ is } R \\ \mathscr{F}_p x_p & \text{else} \end{cases} \tag{26}$$

where b_R is the number of buckets of the hash table for relation R, x_p is the collision rate of the hash table for the parent of r and $\mathscr{F}_p$ is the number of tuples fed to the parent of r during the update process of R. With Equation 26, E_R can be expressed as

$$E_R = b_R \left[\sum_{r \in D_R} \prod_{r' \in A_r - (A_R \cup R)} x_{r'} c_1 + \sum_{r \in L_R} \prod_{r' \in (A_r \cup r) - (A_R \cup R)} x_{r'} c_2 \right] \tag{27}$$

where A_R is the set of all ancestors of R. It is easy to verify that this equation is also correct when R is a leaf relation.

Now, we can express the CEE cost as follows.

$$\begin{aligned} E_{cee} &= \sum_{R \in I} n_R E_R \\ &= \sum_{R \in I} \Bigg[\Big(\sum_{s \subseteq L_R, s \neq \emptyset} (-1)^{|s|+1} \frac{\gamma}{lcm(s)} \Big) \cdot \\ & b_R \Big(\sum_{r \in D_R} \prod_{r' \in A_r - (A_R \cup R)} x_{r'} c_1 + \sum_{r \in L_R} \prod_{r' \in (A_r \cup r) - (A_R \cup R)} x_{r'} c_2 \Big) \Bigg] \end{aligned} \tag{28}$$

Then the overall cost

$$E_c = E_{cie} + E_{cee} \tag{29}$$

7.2 Space Allocation with the New Cost Model

In this section, we analyze how to allocate space to relations in order to minimize the overall cost under the new cost model described in Section 7.1. We still start with a simple case of two-level configuration.

7.2.1 A Case of Two Levels

Consider the case with only one phantom R_0 and it feeds all f queries, $R_1, R_2, \ldots, R_f$, with epoch lengths of w_1, w_2, ..., w_f, respectively. Let x_0 be the collision rate of the phantom, $x_1, x_2, \ldots, x_f$ be the collision rates of the queries. According to Equation 23, the CIE cost can be expressed as follows.

$$\begin{aligned} E_{cie} &= n_c(c_1 + f x_0 c_1 + x_0 \sum_{i=1}^{f} x_i c_2) \\ &= n_c\left[\frac{\mu g_0}{M - \sum_{i=1}^{f} b_i}(f c_1 + \mu c_2 \sum_{i=1}^{f} \frac{g_i}{b_i}) + c_1\right] \qquad (30) \end{aligned}$$

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

According to Equation 28, the cycle-update-cost is

$$E_{cee} = n_{R_0}(M - \sum_{i=1}^{f} b_i)(f c_1 + \sum_{i=1}^{f} \mu \frac{g_i}{b_i} c_2) + \sum_{i=1}^{f} n_{R_i} b_i c_2 \qquad (31)$$

where n_{R_i} (including n_{R_0}) can be obtained from Equation 24.

The overall cost

$$\begin{aligned} E_c &= E_{cie} + E_{cee} \\ &= n_c\left[\frac{\mu g_0}{M - \sum_{i=1}^{f} b_i}(f c_1 + \mu c_2 \sum_{i=1}^{f} \frac{g_i}{b_i}) + c_1\right] \\ &+ n_{R_0}(M - \sum_{i=1}^{f} b_i)(f c_1 + \sum_{i=1}^{f} \mu \frac{g_i}{b_i} c_2) + \sum_{i=1}^{f} n_{R_i} b_i c_2 \qquad (32) \end{aligned}$$

We calculate the partial derivative of E_c over b_i, $i = 1, 2, \ldots f$,

$$\frac{\partial E_c}{\partial b_i} = n_{R_i} c_2 -$$

$$n_{R_0}\left[(M - \sum_{i=1}^{f} b_i)\frac{\mu c_2 g_i}{b_i^2} + (f c_1 + \mu c_2 \sum_{i=1}^{f} \frac{g_i}{b_i})\right] +$$

$$n_c \frac{\mu g_0}{M - \sum_{i=1}^{f} b_i}\left[\frac{(f c_1 + \mu c_2 \sum_{i=1}^{f} \frac{g_i}{b_i})}{M - \sum_{i=1}^{f} b_i} - \frac{\mu c_2 g_i}{b_i^2}\right]$$

Let $\frac{\partial E_c}{\partial b_i} = 0$ and let $A = f c_1 + \mu c_2 \sum_{i=1}^{f} \frac{g_i}{b_i}$, then

$$\frac{n_c \mu g_0 A + n_{R_i} c_2 b_0^2 - n_{R_0} A b_0^2}{n_c \mu g_0 b_0 + n_{R_0} b_0^3} = \frac{\mu g_i c_2}{b_i^2} \qquad (33)$$

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

Unfortunately, this is a non-linear equation system on which we cannot obtain a closed form solution. Numerical methods are not applicable in our applications as explained earlier. In what follows, we perform a simplification on the equation system to get some guidance for space allocation.

Observe that the left hand side of the equation is the same for any i, except the part $n_{R_i} c_2 b_0^2$. However, we find that this part may be neglected based on the following analysis. First, we rewrite Equation 33 to:

$$\frac{n_c \mu g_0 A - n_{R_0} A b_0^2}{n_c \mu g_0 b_0 + n_{R_0} b_0^3} = \frac{\mu g_i c_2}{b_i^2} - \frac{n_{R_i} c_2 b_0}{n_c \mu g_0 + n_{R_0} b_0^2} \qquad (34)$$

Next, we compare the two parts of the right hand side of the equation. Let $y_i = \frac{n_{R_i} c_2 b_0}{n_c \mu g_0 + n_{R_0} b_0^2} \,/\, \frac{\mu g_i c_2}{b_i^2}$. Then

$$y_i = \frac{n_{R_i} b_0 b_i^2}{(n_c \mu g_0 + n_{R_0} b_0^2)\, \mu g_i} < \frac{n_{R_i} b_0 b_i^2}{n_c \mu^2 g_0 g_i}$$

Usually b_0 (b_i) is not much larger than g_0 (g_i), but much smaller than n_c, the number of tuples observed in one cycle. The number of hash table evictions of R_i, n_{R_i}, is very small compared to n_c, too. Therefore, y_i is much smaller than 1, which means that the part $n_{R_i} c_2 b_0^2$ may be neglected in Equation 33. Then, the left hand side of Equation 33 is the same for all b_i's, and hence b_i is proportional to $\sqrt{g_i}$, for $i = 1, 2, \ldots, f$. This result is the same as that of queries with the same epoch length. Therefore, we will use the same heuristics presented in Section 6.3 for space allocation here.

8 Experiments

In this section, we report the results of our experimental study on the SMAP technique. We first validate our cost model in Section 8.2. Then Sections 8.3 and 8.4 evaluate the space allocation schemes and the algorithms for choosing phantoms, respectively, focusing on queries with the same epoch length. Section 8.5 evaluates our algorithm using real collision rate statistics. Section 8.6 evaluates the performance of SMAP on queries with different epoch lengths.

8.1 Experimental Setup and Data Sets

We have implemented SMAP in C. We use 4 bytes as the unit of space allocation. Each attribute value or counter has this size. As explained in Section 2.3, LFTAs are run on a Network Interface Card with a memory constraint, typically several hundred KB of memory is allowed. In accordance to the operational GS system, we consider M between 20,000 and 100,000 units of space (4 bytes each) for hash tables because there may be multiple independent queries that GS needs to run in the available memory of the NIC.

We used both synthetic and real data sets in our evaluation. The real data set is obtained by *tcpdump* on a network server of AT&T. We extracted TCP headers obtaining 860,000 tuples 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 data sets, 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 were run on a desktop with Pentium IV 2.6GHz CPU and 1GB RAM.

There are three options to measure the cost of a configuration: (1) using the cost model developed in Section 3.2 (or Section 7.1 for queries of different epochs) together with the collision rate model developed in Section 5; (2) using the cost model but with real collision statistics (i.e., the numbers of collisions recorded in an experiment where records really run through hash tables); (3) using costs measured in a real implementation in GS. As of the time of writing, the operational GS system is still using the naive algorithm described in Section 2.4. AT&T plans to implement SMAP in GS next, but implementation of such an algorithm in an operational industrial system takes substantial time. Therefore, we use options 1 and 2 in our experiments. We first use option (1) to evaluate our space allocation algorithms (Section 8.3) and the phantom choosing algorithms (Section 8.4). We can then choose our best combined algorithm and compare it with the GP algorithm and the naive algorithm through option (2), to validate the effectiveness of our algorithm in a more practical setting (Section 8.5). Experiments on queries with different epoch lengths (Section 8.6) were conducted in a similar fashion. We provide validation of the cost model in Section 8.2 to justify that option (2) is an accurate simulation to a real system implementation.

8.2 Validation of the Cost Model

We have obtained raw log records corresponding to LFTA queries from the operational GS system to validate our cost model. The essence of the cost model is that: a probe in a hash table has the cost of c_1, an eviction to HFTA has the cost of c_2, and c_2/c_1 is a constant value.

The log records were generated every 10 seconds from an aggregate query that has a 1 minute grouping epoch, so the log records are at a finer granularity. Each record contains fields such as *query_name*, *timestamp*, *in_tuple_cnt*, *out_tuple_cnt*, *cycle_cnt*, *collision_cnt*, etc. The field *cycle_cnt* records the total CPU cycles used by the processor in the 10 seconds. These CPU cycles are used to process input tuples (causing probing costs) and to process output tuples (causing eviction costs) in the 10 seconds. We can fit a linear model of the form:

$$cpu\_cnt = c_1 \cdot in\_tuple\_cnt + c_2 \cdot out\_tuple\_cnt,$$

where *in_tuple_cnt* denotes the number of records coming into LFTA and *out_tuple_cnt* denotes the number of records going out of LFTA to HFTA. We performed a comprehensive regression analysis on logs over a 6 hour period using the above formula and obtained the ratio c_2/c_1 of about 15 for TCP data, with less than 1% mean square error. This result shows that our cost model accurately captures the real system performance.

The ratio c_2/c_1 measured in our earlier work was 50 [33]. Due to various improvements on GS over the years, the current measured ratio is 15 and we have re-run all the experiments using this ratio in this paper. In any case, the analysis remains the same and our algorithms still achieve considerable performance gain.

8.3 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 data sets. We vary M from 20,000 to 100,000 at steps of 20,000 and the granularity for increasing space while executing exhaustive space allocation (ES) is set at 1% of M. We compute the cost using Equation 7 with the collision rate model for clustered data.

8.3.1 Tractable 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.

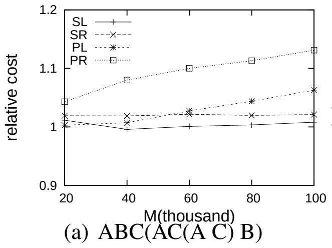
For the case with no phantoms, (assuming $x = \mu g/b$ as collision rate) we compared the cost obtained using the exhaustive space allocation (ES) with the cost obtained using a scheme that allocates space according to our analytical expectations, namely, allocating space proportional to the square root of number of groups. We tested all possible configurations with no phantoms on the real data. The cost obtained using our proposed space allocation scheme has a difference less than 1% compared to the optimal cost (obtained using ES).

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 cost obtained using our space allocation scheme to that obtained using ES using all possible configurations on the real data set. The average relative difference between the cost obtained from our scheme and the optimal cost is usually less than 1% and the maximum observed was 2%.

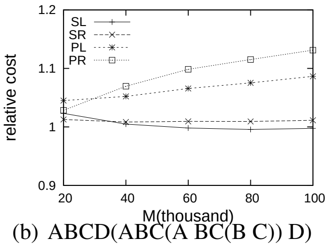
Therefore, the results are accurate despite the approximation ($x = \mu g/b$) to collision rates.

8.3.2 Intractable Configurations

For intractable configurations, we evaluated the heuristics SL, SR, PL, PR as described in Section 6.3. We tested on all possible configurations on the real data set (four attributes). We divide the costs obtained using the heuristics by the optimal cost (obtained using ES) to get relative costs for the heuristics. Figures 15 shows the relative costs obtained using the heuristics for some representative configurations. The average relative costs obtained using the heuristics compared to the optimal cost of all configurations are summarized in Table 3.



(a) ABC(AC(A C) B)



(b) ABCD(ABC(A BC(B C)) D)

Fig. 15 Comparison of space allocation schemes

We observe that generally SL and SR are better than PL and PR. Thus, heuristics inspired by our analytical results appear beneficial. Relative costs obtained using PL and PR can be as large as 30% more than the optimal cost. The cost obtained using SR is smaller than those using PL and PR, but it is in most cases more than the cost of SL. From Table 3, which shows the average relative costs obtained using the four different heuristics, we observe that SL is the best for almost all values of M. Therefore, SL may be a good choice of space allocation. To verify this, we further accumulate statistics in order to see in all configurations tested how frequently SL is the heuristic yielding the minimum cost. In Table 4, we present the percentage of configurations tested in which SL yields minimum cost among the four heuristics, as well as for the cases that SL does not yield the minimum cost, how far the cost obtained using SL is from the minimum cost obtained using the heuristics. These results

M (thousand)	20	40	60	80	100
SL	1.024	1.010	1.004	1.001	1.001
SR	1.021	1.017	1.014	1.011	1.011
PL	1.083	1.084	1.085	1.086	1.092
PR	1.080	1.105	1.115	1.119	1.125

Table 3 Average relative costs of the four heuristics

M (thousand)	20	40	60	80	100
SL being best (%)	31	65	89	92	100
Difference from the best (%)	0.034	0.014	0.005	0	0

Table 4 Statistics on SL

(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 is not the best it remains highly competitive to the best solution. Therefore we would choose SL for space allocation in our algorithms.

8.4 Evaluation of the Phantom Choosing Algorithms

We now turn to the evaluation of algorithms to determine beneficial configurations of phantoms. We will evaluate the adapted algorithm GP and our proposed algorithm GC. Based on previous experimental results, we will consider the two most representative space allocation schemes, SL and PL; we refer to the combination of GC with SL and PL as ***GCSL*** and ***GCPL***, respectively. For GP, 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 use the following method to obtain the optimal configuration cost as a yardstick for measuring other methods. We explore all possible combinations of phantoms and for each configuration we use exhaustive space (ES) allocation to calculate the cost, and then choose the configuration with the minimum overall cost. We will refer to this method as ***EPES*** in the remainder. We still compute the cost using Equation 7 with the collision rate model for clustered data.

We look at the query set $\{A, B, C, D\}$ on a 4-dimensional synthetic data set generated as described in Section 8.1 with M set to 40,000. Figure 16 presents the relative costs obtained using different algorithms normalized by the optimal

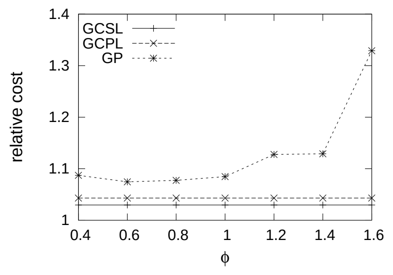


Fig. 16 Phantom choosing algorithms

cost obtained using EPES. Algorithm GP has a parameter ϕ. Since a good value of ϕ is not known a priori, we vary it and observe the trend of the cost resulting from different ϕ values. The cost of GP 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 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. Algorithms GCSL and GCPL do not have the parameter ϕ, so their costs are constant in the Figure. Even though a minimum point can be found for the cost obtained using GP as ϕ varies, GCSL and GCPL are always better than GP due to better phantom choosing strategies (supernode heuristics based on our analysis). GCSL is better than GCPL because GCSL uses a better space allocation scheme. Thus, GCSL benefits from both the way we choose phantoms and the way space is allocated.

Figure 17 presents the change in the overall cost in the

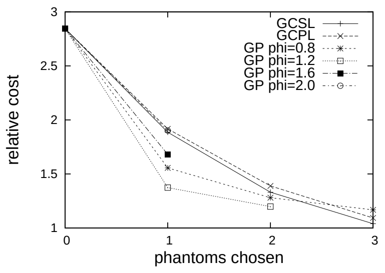


Fig. 17 Phantom choosing process

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. Note that the third phantom added by GP with $\phi = 0.8, 1.2$ is different from the third phantom added by GCSL due to the differences in the space allocation scheme. For GP with $\phi = 1.6, 2.0$, there is no space to add more than one phantom. The optimal configuration for this set of queries is ABCD(AC(A C) BD(B D)) obtained through EPES. GCSL finds the same configuration, while GP finds the configuration of ABCD(ABC (AC(A C) B)).

We conducted an additional experiment which varies M and uses the synthetic data set with four attributes, comparing GCSL and GP. Figure 18 shows the relative cost (compared to the optimal cost) obtained using GCSL and GP with

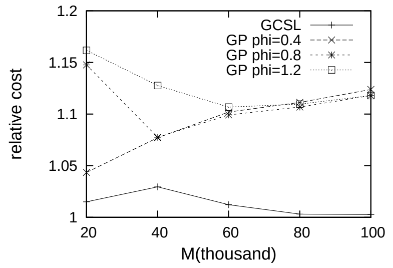


Fig. 18 Cost comparison

various ϕ values. The cost obtained using GCSL is always lower than 1.04 times of the optimal, and is also always lower than the cost obtained using GP.

8.5 Experiments with Real Collision Statistics

Now we use option (2) as described in Section 8.1 to evaluate our algorithm. We implemented the hash tables and then let the data stream actually run through the phantoms and queries. We record the collision counts of the hash tables and compute the costs using the cost model (Equation 7). Again, we normalize the costs obtained using GCSL and GP by the cost of the optimal cost obtained using EPES, to get relative costs.

Synthetic Data. The relative costs obtained using GCSL and GP on the synthetic data set are shown in Figure 19(a). For GP, we tried different ϕ values, and only the one with the lowest cost at each value of M is plotted. The cost of GCSL

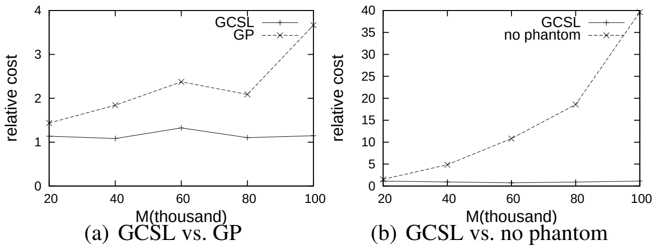


Fig. 19 Comparison on synthetic data set

is always much lower than that of GP, even if we could always choose the best ϕ for GP (which is impossible in practice). This confirms the results of the previous experiments based on the collision rate model. When M=100,000, the cost of GCSL is as low as 26% of the cost of GP. While GP can have cost as high as 3.8 times the optimal cost, GCSL is always within 1.2 times the optimal cost.

In order to validate the effectiveness of phantoms for computing multiple aggregations, we also compared GCSL with the naive method as described in Section 2.4, i.e., to process the queries without any phantom. We run the experiments using the same setting as the previous one and the relative costs obtained using GCSL and using the naive method are shown in Figure 19(b). It is evident that maintaining phantoms reduces the cost significantly (more than an order of magnitude in many cases). The relative cost obtained using the naive method increases as M increases because the optimal cost (obtained using EPES) decreases as M increases while the cost obtained using the naive method is constant.

Real Data. We conducted the same experiment using real data this time and the query set {AB, BC, BD, CD}. Figure 20 shows the results. It is evident that GCSL outperforms GP and the naive method by a large margin. The optimal configuration for this set of queries (e.g., when M=40,000) is ABCD(AB BCD(BC BD CD)) obtained through EPES. GCSL finds the same configuration, while GP finds the configuration of ABCD(ABC(AB BC) BD CD).

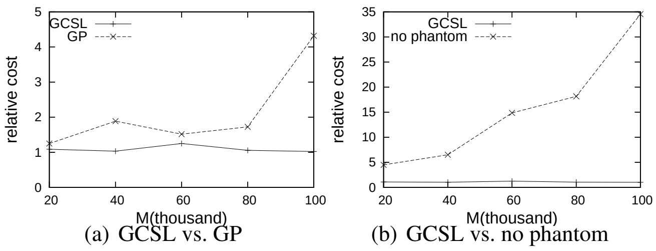


Fig. 20 Comparison on real data set

8.5.1 Effect of Misestimation in Number of Groups

Sampling and sketching techniques are used to estimate the number of groups for each relation. These techniques can only provide approximate numbers. Further, we use the previous time window's numbers of groups to predict the current time window's numbers, which may also introduce some errors. We performed experiments to see how the misestimation affects the performance of our algorithm. We have randomly introduced errors in ranges varying from -50% to 50% to the actual numbers of groups of the relations and still compare the performance of GCSL to the naive method. Representative results are shown in Figure 21 for M=60,000 and M=100,000. Compared to the naive method, GCSL has quite stable performance gain over the naive method even with a wide range of misestimation.

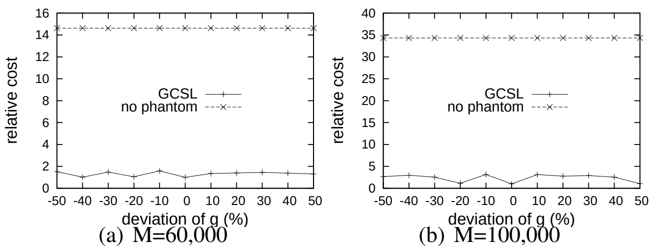


Fig. 21 Effect of misestimation in number groups

8.5.2 Peak Load Constraint and Running Time

The end-of-epoch (EE) cost E_e described in Section 3.2.2 can be calculated according to Equation 8. It must be smaller than the peak load constraint E_p. If E_e 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 since c_2 is much larger than c_1 and a major part of the update cost is incurred by queries. For the real data set and the query set {AB, BC, BD, CD}, given a space allocation, we calculate its E_e; then we set E_p to a percentage of E_e 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 22. When E_p is not much smaller than E_e, the

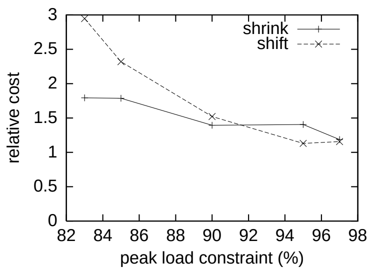


Fig. 22 Peak load constraint

shift method performs better; while when E_e is much larger than E_p, the shrink method performs better. The reason is that when E_e is close to E_p, a small shift to reduce E_e suffices. When E_e 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 is submillisecond, which is negligible compared to a time window of at least several seconds long. While typically we just have a few aggregate queries running together, our algorithm has good scalability since the computation cost is linear to the number of relations in the feeding graph.

8.6 Queries with Different Epoch Lengths

We summarize all the experimental results on queries with different epoch lengths in this subsection. We used the same data sets as before as well as a very similar sets of experiments to the ones presented in previous subsections. The main differences are that here the queries have different epoch lengths and Equation 29 is used to compute the overall cost. In terms of methods to measure costs, we followed a similar fashion to how we measured costs for queries with the same epoch length. Sections 8.6.1 and 8.6.2 uses option (1); Section 8.6.3 uses option (2).

8.6.1 Evaluation of Space Allocation Strategies

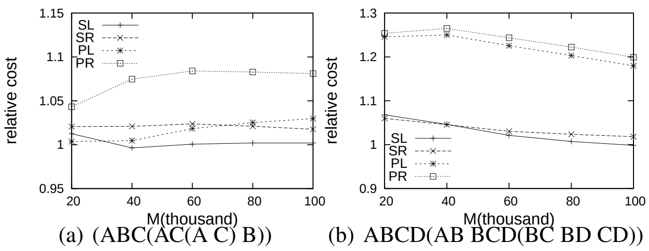
We used two query sets, {A, B, C} and {AB, BC, BD, CD}. For {A, B, C}, we evaluated various sets of epoch lengths, (2,3,5), (3,4,5) and (1,5,9); each number in a set of epoch lengths corresponds to the epoch length of a query. For example, (2,3,5) means that the epoch lengths of A, B and C are 2, 3 and 5, respectively. For {AB, BC, BD, CD}, we also used three sets of epoch lengths, (2,3,5,6), (2,3,5,7) and (3,5,7,9).

Tractable Configurations. In the new cost model, only the case with no phantoms is tractable. The cost obtained using analysis differs from the optimal cost by less than 1%.

Intractable Configurations. Based on the analysis in Section 7.2, we still use the heuristics SL, SR, PL, PR described in Section 6.3 for space allocation, but use Equation 29 for computing costs.

For the case with only one phantom, all the heuristics achieve a cost within 1.2% difference from the optimal cost for all experiments.

For other cases, we have tested various configurations. In most cases, SL is the best among all the heuristics. In rare cases where SL is not the best, it is very close to the best heuristic. Figure 23 shows the relative costs (normalized by the optimal cost) obtained using the four heuristics as functions of the available memory M for two representative configurations, (ABC(AC(A C) B)) with epoch lengths (2, 3, 5) for queries {A, B, C}, and ABCD(AB BCD(BC BD CD)) with epoch lengths (2, 3, 5, 6) for queries {AB, BC, BD, CD}.



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

Fig. 23 Comparison of space allocation schemes

Table 5 presents the average relative costs obtained using the four heuristics for all the experiments we have done. It also suggests that SL is the best for all values of M. In the following experiments evaluating phantom choosing algorithms, we will only consider the two most representative space allocation schemes, SL and PL.

M (thousand)	20	40	60	80	100
SL	1.038	1.021	1.011	1.031	1.001
SR	1.023	1.033	1.030	1.027	1.022
PL	1.097	1.105	1.107	1.101	1.092
PR	1.088	1.122	1.133	1.132	1.122

Table 5 Average relative costs of the four heuristics

8.6.2 *Evaluation of Phantom Choosing Algorithms*

Like what we did in the case of single query epoch length, we considered three algorithms GCSL, GCPL and GP for phantom choosing. The costs obtained using these algorithms are normalized by the optimal cost (obtained using EPES) and hence represented as relative costs. Costs are computed using Equation 29. We used the query set {A, B, C, D} with epoch lengths (2,3,5,6) on the 4- dimensional synthetic data set, M set as $40,000$. Figure 24 presents the relative costs obtained using different algorithms. The result is similar to the case of single query epoch length. GCPL and GCSL are both always better than GP. Even though a minimum point can be found for the cost obtained using GP as ϕ varies, GCSL and GCPL are always better than GP due to better phantom choosing strategies (supernode heuristics based on our analysis). GCSL is better than GCPL due to the better space allocation scheme.

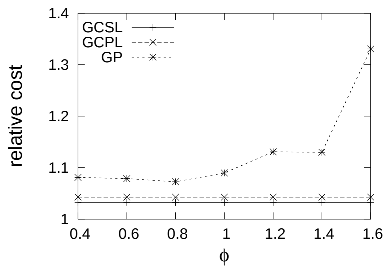


Fig. 24 Comparison of phantom choosing algorithms,M=40,000

Figure 25 shows comparison between the relative costs obtained using GCSL and GP with different ϕ values as the value of M changes. Again, the cost obtained using GP may have a minimum point at a certain M value for each ϕ value, but GCSL constantly outperforms GP for all cases.

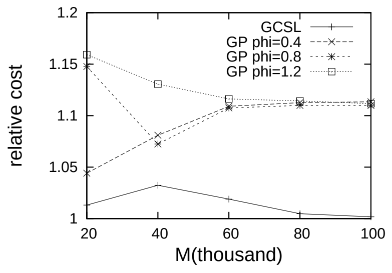
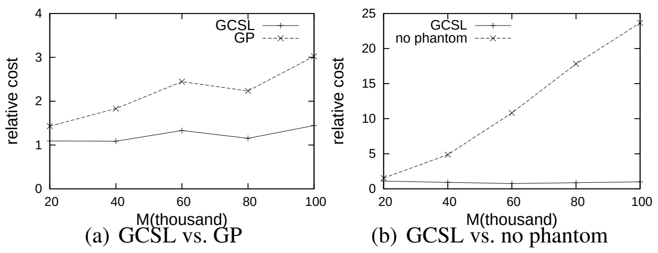


Fig. 25 Cost comparison

8.6.3 *Experiments with Real Collision Statistics*

Now we use option (2) as described in Section 8.1 to evaluate our algorithm for queries with different epoch lengths. We still used the query set {A, B, C, D} with epoch lengths (2,3,5,6) on 4-dimensional synthetic data. The relative costs (normalized by the optimal cost) obtained using GCSL and GP are compared in Figure 26(a) as functions of M. We tried different ϕ values for GP and picked the best one at

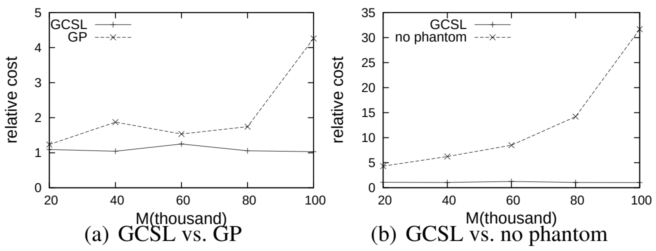


(a) GCSL vs. GP (b) GCSL vs. no phantom

Fig. 26 Comparison on synthetic data set

each M value to present in the figure. GCSL constantly outperforms GP even if we could always choose the best ϕ value. While GP can have a cost as high as 3 times the optimal cost, GCSL always has a cost within 1.4 times the optimal cost. With the same experimental setting, we also compared GCSL with the naive method, i.e., processing the queries without any phantom, and the result is shown in Figure 26(b). Compared to the naive method, our SMAP technique does yield significant cost reduction even for queries with different epoch lengths, as much as a factor of 20 at the point of $M = 100,000$.

We repeated the above experiment using the real data set as described in Section 8.1. We used the query set {AB, BC, BD, CD} with epoch lengths (2,3,5,6). Figure 27 presents the results. We still observe that GCSL always outperforms GP and processing without phantoms by a large margin.



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

Fig. 27 Comparison on real data set

It is easy to see that the peak load for the case of different query epoch lengths happens at the end of each cycle, when all relations need to perform EE updates. We tried the same two methods, *shrink* and *shift*, like in the case of single query epoch length, to keep the peak load within the constraint E_p. The results obtained exhibit very similar behavior to those of the case of single query epoch length and are omitted here. As for running time of our algorithms, we still observe several to tens of milliseconds, which is still extremely fast and suits our application very well.

9 Related Work

The focus of our work is how to optimize the computation when processing *multiple* aggregate queries over the network data streams. This work is closely related to the problem of multi-query optimization, that is, 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. Hall [21] uses operator trees to represent the queries and a bottom-up traversal algorithm to find common subexpressions. Roussopoulos [30] uses query graphs [32] to find views for indexing to minimize the total cost of answering all queries. Charkravarthy and Minker [7] extended query graphs to integrated query graphs and proposed an algorithm based on integrated query graphs. Finkelstein [17] studied how query processing can be improved using answers produced from early queries when no common expressions are found for multiple query optimization. The basic idea of all these techniques is the identification of common query subexpressions, whose evaluation can be shared among the query execution plans produced. Our technique of sharing computation common to multiple aggregate queries is based on the same idea, but our technique is specially devised for data stream applications where the data inputs stream by the system at extremely high speed.

Our problem has similarities to the view materialization problem, which has been studied extensively in the literature (a survey can be found in [20]). Ross et al. [29] 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. Harinarayan et al. [23] proposed a lattice framework to represent the dependencies among views and a greedy algorithm to determine which views to materialize. Our idea of additionally maintaining phantoms, and choosing which phantoms to maintain, to efficiently process multiple aggregations is based on the same conceptual idea. However, due to two differences: i) maintaining a view uses fixed space while maintaining a phantom can use flexible space; ii) maintaining a view is always beneficial while maintaining a phantom is not, the greedy algorithm proposed by Harinarayan et al. [23] is not directly applicable to our problem. Although we can somehow adapt the view materialization algorithm to our problem, the adapted algorithm is very cumbersome and shown to be inefficient by our experiments. We devised a novel greedy algorithm for our problem which optimizes the cost.

Conceptually, our proposed phantoms have similarities to partial preaggregates. Larson [27] shows that if an expensive query (join, select, union, etc) has aggregation on top, we can preaggregate on any attribute set that functionally determines the group-by attributes, to reduce the query cost. In our case, the preaggregation (phantoms) is shared by multiple queries rather than reducing the cost of one expensive query. Larson [27] provides a model for estimating the output size, while our collision rate model estimates the number of collisions of hash tables.

Many papers (see, e.g., [11,6,8]) have highlighted the importance of resource sharing in continuous queries. Chen et al. [11] and Madden et al. [28] 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. [16] considered the problem of sharing sketches for approximate join-based processing.

On computing aggregations over data streams, Gehrke et al. [18] proposed single-pass techniques for approximate computation of correlated aggregations such as "compute the percentage of international phone calls that are longer than the average duration of a domestic phone call", whose exact answer requires multiple passes over the data. Chaudhuri et al. [10] use sampling-based technique to provide approximate answers to aggregate queries, but different from previous studies, they treat the problem as an optimization problem whose goal is to minimize the error in answering queries in the given workload. Dobra et al. [15] use randomized techniques that compute "small" summaries of the streams to provide approximate answers for general aggregate SQL queries with provable guarantees on the approximation error. Gilbert et al. [19] use sketching techniques to approximate wavelet projections that represent the original data stream. However, none of these papers considered query optimization through sharing among different queries. Diao et al. [14] proposed filtering techniques for optimizing multiple queries on XML data streams. Demers et al. [13] presented a publish/subsribe system and considered multiple query optimization on event streams. Hong et al. [24] proposed a framework for multi-query optimization through the use of transformation rules.

Chandrasekaran and Franklin [9] and Arasu and Widom [3] studied sharing data structures among sliding window queries posed on the same input stream, aggregate function and aggregated attributes. The sharing queries only differ in the window specifications. The approach proposed in [9] is to maintain a ranked n-ary tree on the recent data that can cover all the sliding windows of the queries. Each leaf of the tree is a data record of the stream and the leaves are sorted in the order of their insertion times. Each internal node stores the value of the aggregate function computed over the descendent leaves of that node. This algorithm has an update and search cost of $O(\log n)$, where n is the number of elements maintained in the tree, which equals the number of elements needed to cover all the query windows. The other work, Arasu and Widom [3], proposes to exploit the *distributive* or *algebraic* properties of the aggregate functions. These properties ensure that the aggregation over the union of two sets of elements can be computed from the aggregation over each set (for example, $SUM(S_1 \bigcup S_2) = SUM(S_1) + SUM(S_2)$). The idea of Arasu and Widom [3] is to precompute the aggregation over some intervals and store them. When an aggregate query is issued, the window of the query is decomposed into intervals whose aggregations have been precomputed. Then the asked aggregation can be computed from the precomputed aggregations. This approach uses more space (to store the precomputed aggregations) and may have more update costs. Our approach shares the computation among different aggregated attributes, or relations in general. More recently, Krishnamurthy et al. [26] investigated sharing of resources among streamed aggregations by exploiting similarities in the queries. This study focused on methods for sharing time slices among queries with the same predicate but varying windows, sharing data fragments with different predicates but the same time window, and a combination of these two types of sharing. Our work focuses on detailed schemes for allocating resources to each query that shares computation.

10 Conclusions and Discussions

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.

Our work has focused on aggregate queries with the *count* function on tumbling windows. It is also possible to apply our technique to more general cases: (i) *Sharing among aggregate queries with the same other function, e.g., sum.* We simply maintain the sum of the records in hash table entries rather than the count, same with *max* or *min*. If the function is *average*, we can maintain both the count and the sum of records in hash table entries to obtain the average. (ii) *Sharing among aggregate queries with different functions, e.g., max and count.* We can maintain both the max and the count of records in hash table entries for phantoms. (iii) *Sharing among other predicates besides aggregate queries.* We can adopt a similar idea on selections and projections. However, as selections and projections can be processed rather straightforwardly, whether using phantoms is beneficial requires more cost analysis. We defer such investigations to future work. (iv) *Sharing among queries with sliding windows or a combination of sliding and tumbling windows.* We can adopt the technique proposed by Krishnamurthy et al. [26] to break a sliding window into two slices. Sliced windows from different queries are combined when they have the same period, which become essentially tumbling windows, and then we can use our technique on such combined queries. Since a tumbling window is a special case of a sliding window, a combination of sliding and tumbling windows can be processed in the same way.

Another interesting direction of further study is the replacement policy of the hash tables. In the current GS system, an old entry is evicted whenever a collision occurs. This is probably a good policy considering the clusteredness in the data – many following records would be in the same flow (group) and cause no collision. However, the experiments in Section 5.3 show that the average flow length is close to 1 for relations of depths greater than 2 in the configuration tree. The average flow length being 1 means the groups are randomly mixed. In this case, the best policy may be *not* to replace the old entry on collision, but to simply evict the new entry. This is because high-frequency groups are likely to be created earlier than low-frequency groups and keeping the high-frequency groups in the hash table would reduce the collisions in total. Therefore, a more sophisticated replace policy depending on the depth of a relation may further reduce the overall cost.

Acknowledgements We would like to thank Ted Johnson for pointing out that Equation 12 may be further simplified. This work is partially supported by the Australian Research Council's Discovery funding scheme (project number DP0880250).

References

1. N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *ACM Symposium on Theory of Computing (STOC)*, pages 20–29, Philadephia, USA, 1996.
2. A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, 2003.
3. A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *International Conference on Very Large Data Bases (VLDB)*, pages 336–347, Toronto, Canada, 2004.
4. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 1–16, Madison, USA, 2002.
5. A. D. Barbour, L. Holst, and S. Janson. *Poisson Approximation*. Oxford Science Publications, 1992.
6. D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. In *International Conference on Very Large Data Bases (VLDB)*, pages 215–226, Hong Kong, China, 2002.
7. U. Chakravarthy and J. Minker. Processing multiple queries in database systems. *IEEE Database Engineering Bulletin*, 5(3):38–44, 1982.
8. S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, S. K. W. Hong, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Conference on Innovative Data Systems Research (CIDR)*, Asilomar, USA, 2003.
9. S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *International Conference on Very Large Data Bases (VLDB)*, pages 203–214, Hong Kong, China, 2002.
10. S. Chaudhuri, G. Das, and V. Narasayya. A robust, optimization-based approach for approximate answering of aggregate queries. In *ACM International Conference on Management of Data (SIGMOD)*, pages 295–306, Santa Barbara, USA, 2001.
11. J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *ACM International Conference on Management of Data (SIGMOD)*, pages 379–390, Dallas, USA, 2000.
12. C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *ACM International Conference on Management of Data (SIGMOD)*, pages 647–651, San Diego, USA, 2003.
13. A. J. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. M. White. Towards expressive publish/subscribe systems. In *EDBT*, pages 627–644, 2006.
14. Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. M. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.
15. A. Dobra, M. N. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *ACM International Conference on Management of Data (SIGMOD)*, pages 61–72, Madison, USA, 2002.
16. A. Dobra, M. N. Garofalakis, J. Gehrke, and R. Rastogi. Sketch-based multi-query processing over data streams. In *International Conference on Extending Database Technology (EDBT)*, pages 551–568, Heraklion, Greece, 2004.
17. S. Finkelstein. Common expression analysis in database applications. In *ACM International Conference on Management of Data (SIGMOD)*, pages 235–245, Orlando, USA, 1982.
18. J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *ACM International Conference on Management of Data (SIGMOD)*, pages 13–24, 2001.
19. A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *International Conference on Very Large Data Bases (VLDB)*, pages 79–88, Roma, Italy, 2001.
20. A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, 1995.
21. P. A. V. Hall. Optimization of single expressions in a relational data base system. *IBM Journal of Research and Development*, 20(3):244–257, 1976.
22. M. A. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Y. Eltabakh, M. G. Elfeky, T. M. Ghanem, R. Gwadera, I. F. Ilyas, M. S. Marzouk, and X. Xiong. Nile: A query processing engine for data streams. In *ICDE*, page 851, 2004.
23. V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *ACM International Conference on Management of Data (SIGMOD)*, pages 205–216, Montreal, Canada, 1996.
24. M. Hong, M. Riedewald, C. Koch, J. Gehrke, and A. J. Demers. Rule-based multi-query optimization. In *EDBT*, 2009.
25. N. Koudas and D. Srivastava. Data stream query processing: A tutorial. In *International Conference on Very Large Data Bases (VLDB)*, page 1149, 2003.
26. S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD Conference*, 2006.
27. P.-Å. Larson. Data reduction by partial preaggregation. In *ICDE*, 2002.
28. S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *ACM International Conference on Management of Data (SIGMOD)*, pages 49–60, Madison, USA, 2002.
29. K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *ACM International Conference on Management of Data (SIGMOD)*, pages 447–458, Montreal, Canada, 1996.
30. N. Roussopoulos. View indexing in relational databases. *ACM Transactions on Database Systems (TODS)*, 7(2):256–290, 1982.
31. M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *USENIX Technical Conference*, New Orleans, USA, 1998.
32. E. Wong and K. Youssefi. Decomposition - a strategy for query processing. *ACM Transactions on Database Systems (TODS)*, 1(3):223–241, 1976.
33. R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple aggregations over data streams. In *ACM International Conference on Management of Data (SIGMOD)*, pages 299–310, Baltimore, USA, 2005.