Memory-Constrained Aggregate Computation over Data Streams

K.V.M. Naidu
Bell Labs Research, India
naidukvm@alcatel-lucent.com

Rajeev Rastogi
Yahoo! Labs Bangalore, India
rrastogi@yahoo-inc.com

Scott Satkin
CMU, USA
scott@satkin.com

Anand Srinivasan
Google, India
anandsr@gmail.com

***Abstract*— In this paper, we study the problem of efficiently computing multiple aggregation queries over a data stream. In order to share computation, prior proposals have suggested instantiating certain *intermediate aggregates* which are then used to generate the final answers for input queries. In this work, we make a number of important contributions aimed at improving the execution and generation of query plans containing intermediate aggregates. These include: (1) a different hashing model, which has low eviction rates, and also allows us to accurately estimate the number of evictions, (2) a comprehensive query execution cost model based on these estimates, (3) an efficient greedy heuristic for constructing good low-cost query plans, (4) provably near-optimal and optimal algorithms for allocating the available memory to aggregates in the query plan when the input data distribution is Zipf-like and Uniform, respectively, and (5) a detailed performance study with real-life IP flow data sets, which show that our multiple aggregates computation techniques consistently outperform the best-known approach.**

I. Introduction

Computing multiple aggregation queries over a data stream has applications in many domains: IP network monitoring, stock trading, analysis of Web logs, fraud detection in telecom networks and retail transactions, querying sensor node readings, *etc*. Salient characteristics of these applications include: (1) very high data arrival rates that make it impractical to perform multiple passes over the data, (2) hundreds of aggregation queries, and (3) limited CPU and memory resources.

As an example, consider an IP network monitoring system, which collects IP flow records exported by network routers and performs a variety of monitoring tasks like estimating traffic demands between IP endpoints, computing the top hosts in terms of IP traffic, profiling application traffic, and detecting network attacks and intrusions. These monitoring applications may require aggregate measurements over different sets of flow attributes. For instance, a Denial-of-Service (DoS) attack detection application will be interested in the *number of packets for every destination IP, destination port aggregated over 5 minute intervals* so that it can identify the destinations that are receiving an unusually large number of packets. On the other hand, a traffic engineering application will require the traffic demand matrix, that is, the *number of packets between every source IP-destination IP pair over 5 minute intervals*.

Now, a production service provider network contains hundreds of routers that can easily generate massive amounts of flow data. In fact, it has been reported that, even with a high degree of sampling and aggregation [7], the AT&T IP backbone network alone generates 500 GB of flow data per day (about ten billion fifty-byte records). Thus, for scalability in the presence of multiple queries and high-speed streams, the aggregate computations must be highly optimized both in terms of the CPU cycles consumed, and memory overheads that they entail.

Previous work has used the idea of *resource sharing* among multiple queries to optimize cube computation [2], [10], answer continuous queries over data streams [6], [13], process sliding-window aggregates [3], [12], and optimize the communication overhead for distributed queries [11]. In recent work, Zhang *et al.* [14] proposed an optimization that allows computation to be shared among queries over an input stream. Their proposal involves instantiating a few fine-grained *intermediate aggregates* (called *phantoms* in [14]) and then use these to generate the final query answers. The key idea here is that the intermediate aggregates will generally be much smaller than the input stream itself, and so computing multiple query results from an intermediate aggregate will be much cheaper than answering the queries directly from the input stream. For example, our DoS attack detection query above aggregates packets over attributes $\{\text{destination IP}, \text{destination port}\}$, while the traffic engineering query needs packets to be aggregated on attributes $\{\text{source IP}, \text{destination IP}\}$. In this case, it might be beneficial to maintain an intermediate aggregate over attributes $\{\text{source IP}, \text{destination IP}, \text{destination port}\}$ and then compute the required query answers from this intermediate aggregate.

An important problem here is selecting the right set of intermediate aggregates to instantiate. A key challenge is to allocate the limited memory among the hash tables for these aggregates so that the total number of hash computations is minimized. Zhang et al. [14] propose greedy heuristics for both problems. In this paper, we propose new strategies for generating and executing computation-efficient query plans containing intermediate aggregates. Our main contributions include:

1) Our hashing model uses chaining to support multiple entries per bucket; as a result, entries do not need to be evicted from the table every time there is a collision if the hash table has free capacity. Thus, our hashing model results in much lower eviction rates (and as a consequence, fewer hash computations) compared to existing hash models (*e.g.*, [14]) that push out entries every time there is a collision. (We limit the total number of entries in a hash table to keep the chains short.)

2) Our query execution cost model is based on accurate analytical estimates for hash table eviction rates for Zipf-like and Uniform distributions of the input data. (We validate our cost model using real-life data sets.) Further, our cost model is comprehensive and includes all hash computation costs at every stage of aggregate computation.

3) We show that the problem of finding a minimum-cost query plan is NP-hard, and propose an efficient greedy heuristic for constructing good low-cost query plans. Our heuristic is more efficient than many of the existing approaches (*e.g.*, [14]) that enumerate all possible aggregates and thus have a time complexity that is exponential in the number of attributes.

4) For a given query plan, we present a fully polynomial-time approximation scheme (FPTAS)[1] for obtaining near-optimal memory allocation assuming that the input distribution is Zipf-like. For the special case in which the input distribution is Uniform, we present a much simpler memory allocation algorithm that is provably optimal.

5) Finally, we present extensive experimental results with real-life IP flow data sets which show that our query plans and memory allocation algorithms provide significant benefits over the best-known approach for this problem by Zhang *et al.* [14].

For ease of exposition, proofs of theorems in the paper are presented in the appendix.

II. Related Work

Our multiple aggregate computation problem is closely related to the cube computation problem [2], but differs in that we are not trying to aggregate the input data on all possible attribute combinations. Our problem also has similarities to the materialized view selection problem studied in [10]. Harinarayan *et al.* [10] present greedy algorithms to select the optimal set of intermediate group-bys to materialize such that the cost of computing the remaining group-bys in the cube is minimized. In a stream setting, however, aggregate computation must be performed in real-time; thus, our problem formulation also takes into account the cost of computing intermediate aggregates which is not the case in [10].

More recently, many systems [6], [13], [3], [12] for processing continuous queries over data streams have employed resource sharing to achieve better scalability. For instance, [6], [13] use variants of predicate indexes for sharing predicate evaluation in the presence of joins; however, they do not consider aggregate computation. And [3], [12] propose techniques to share processing of sliding-window aggregates with varying windows by combining partial aggregates for different time slices. The papers, however, primarily focus on the handling of sliding windows, and do not utilize new intermediate aggregates for reducing the processing overheads. Recently, Huebsch *et al.* [11] presented algorithms for optimizing the communication overhead (instead of computation cost) in their distributed implementation of multiple aggregate computation.

The work that is closest to ours is that of Zhang *et al.* [14], where they propose to maintain additional *phantoms* to share computation across multiple aggregate queries in the Gigascope [7] IP stream processing system. Our intermediate aggregates are conceptually similar to phantoms, but our techniques for evaluating and selecting query plans are different. We highlight these differences below.

First, Gigascope's lower layer resides on a Network Interface Card (NIC) with limited memory (a few MBs). As a result, [14] assumes a simple model of hashing that handles a collision (caused by two entries hashing to the same bucket) by evicting the existing entry. In contrast, we carry out all the query processing in a compute server's main memory. Our hash table implementation uses chaining to allow multiple entries per bucket, and only evicts an entry when the hash table is full. (In Section III-C, we show that chain lengths are short for typical hash table configurations.) Thus, our eviction policy leads to much lower hash eviction rates, and very different query processing cost models.

Second, analytically modeling hash eviction rates is much more complicated for the eviction policies of [14]. Consequently, to derive eviction rate formulas to plug into the cost model, the authors rely on empirical approximations, and restrict themselves to certain low-collision operating regions, which are characterized by a small number of hash table entries compared to the hash table size, and may be rare in practice (especially in memory-constrained environments). Further, to simplify their analysis, Zhang *et al.* assume that the input data distribution is Uniform. In contrast, we are able to derive clean analytical estimates for the eviction rates for both Uniform as well as Zipf-like distributions under our hashing model. (Studies on Web proxy cache traces [4] reveal that the distribution of Web page requests is Zipf-like. Similarly, we found the data distribution in real-world IP flow data [1] to be Zipf-like as well.)

Third, the query processing cost model in [14] only considers the hash computation costs incurred when tuples are streaming in, while hash operations performed at the end of each aggregation period (when hash tables are flushed) are not included in the total query cost that is optimized. The latter cost is especially significant for larger memory sizes; as a result, in our experiments, we found that their scheme does not generate good query plans as memory size is increased. Our query plans, on the other hand, work well for the entire range of memory sizes since our cost model includes both hashing costs.

Fourth, the query plan generation heuristic of [14] enumerates all possible aggregates – this is clearly exponential in the number of group-by attributes, and thus impractical for large attribute sets. In contrast, our plan generation heuristic is more efficient and, at each point in its execution, only considers aggregates obtained by combining input query (as well as already chosen) aggregate pairs.

And finally, [14] shows that the memory allocation problem is essentially unsolvable for trees with depth more than two, and so resorts to heuristics for space allocation. In contrast, our space allocation algorithms are provably optimal (for Uniform distributions) or close to optimal (for Zipf-like distributions).

[1]A *Fully Polynomial-Time Approximation Scheme* (FPTAS) is an approximation algorithm which (1) for a given $\epsilon > 0$, returns a solution whose cost is within $(1 \pm \epsilon)$ of the optimal cost, and (2) has a running time that is polynomial in the input size and $1/\epsilon$.

III. Query Evaluation Framework

A. System Model

We consider a single data stream consisting of an infinite sequence of tuples, each with *group-by* attributes $a_1, \ldots, a_m$ (e.g., source/destination IP addresses, source/destination ports), and a measure attribute a_0 (e.g., byte count). We are interested in answering a set of aggregate queries $\mathbb{A} = \{A_1, \ldots, A_n\}$ defined over the stream of tuples. Each A_i specifies the subset of group-by attributes on which aggregation is performed; a result tuple is output for each distinct A_i value. The measure attribute values are aggregated using one of the standard SQL aggregation operators (e.g., SUM, MIN). Similar to [14], we assume that queries are processed in an *epoch by epoch* fashion. An epoch is essentially a fixed time interval over which aggregation is performed; at the end of each epoch , result tuples for each unique combination of group-by attribute values and the associated aggregated measure attribute value are output[2]. We denote the number of stream tuples that arrive in an epoch by n_R.

The following example query returns the total traffic in bytes between every pair of IP addresses aggregated over 15 minute intervals.

select srcIP, dstIP, SUM(bytes)
from *stream*
group by srcIP, dstIP
every 15 minutes

Note that the queries in $\mathbb{A}$ differ only in their grouping attributes. We denote the number of distinct values observed for A_i over one epoch by g_{A_i}[3]. Our query processing engine has a limited amount of memory which is large enough memory to hold the $\sum_i g_{A_i}$ result tuples for the aggregate queries in $\mathbb{A}$. We will denote the excess memory available to our engine (for storing auxiliary structures different from result tuples) by M.

B. Naive Query Evaluation Strategy

A straightforward way to support multiple queries is to simply process each aggregation query independently for each incoming stream tuple. For each query A_i, we maintain a separate hash table on its group-by attributes. Processing the query for a tuple then involves hashing on the attributes in A_i to locate the hash bucket for the tuple, and then updating the aggregate statistic for the group-by attribute values. In the second step, when a tuple with a specific combination of grouping attribute values is encountered for the first time, then a new entry for that group is created and initialized in the bucket. If an entry for the group already exists in the bucket, then only the aggregate statistic for the group is updated. (We describe space allocation and collision-handling for our hash tables in more detail in Section III-C.)

At the end of each epoch, the result tuples for all the aggregates are output by scanning the non-empty buckets in the hash table for each aggregate query, and writing to an output file the group-by attribute values and the aggregate value in every bucket entry. Once all the result tuples are written, all the hash tables are re-initialized by setting their buckets to be empty.

The aggregate computation cost is dominated by the CPU cycles required for hashing incoming stream tuples, finding and updating the appropriate bucket entry, etc. Thus, the total computation cost is proportional to the number of hash operations. So in the rest of the paper, we use hashing costs as our cost metric.

C. Reducing Computation Using Intermediate Aggregates

Processing each query independently as described above can easily lead to redundant computation. This, in turn, can adversely impact the ability of our query engine to handle high-speed data streams, and cause it to drop some of the incoming tuples. To prevent such a scenario, it is imperative that the (hash) computation overhead of our query evaluation schemes be as low as possible.

To reduce the total number of hash operations performed during query execution, Zhang et al. [14] introduce the notion of *intermediate* aggregates (which they refer to as *phantoms* in [14]). The key idea is to instantiate new aggregates B for different subsets $\mathbb{A}' \subseteq \mathbb{A}$ of the input queries. The aggregate B for a query subset $\mathbb{A}'$ contains all the attributes that appear in $\mathbb{A}'$, that is, $B = \cup_{A' \in \mathbb{A}'} A'$. Now, it is easy to see that intermediate aggregate B can be used to compute any aggregate $A' \in \mathbb{A}'$. This is because all the group-by attribute values for A' are present in the result tuples for B. Thus, since the intermediate aggregate B is typically much smaller than the raw tuple stream, we can obtain a significant reduction in the number of hash computations by computing the aggregates $A' \in \mathbb{A}'$ using the result tuples of B as input instead of stream tuples.

More formally, let g_B denote the size of aggregate B, *i.e.*, g_B is the number of distinct groups observed for the group-by attributes B in the tuple stream over an epoch. Then, computing aggregate A' directly from the raw stream incurs n_R hash computations, where n_R is the number of tuples in the raw stream. On the other hand, further aggregating the result tuples for an intermediate B to compute an aggregate $A' \in \mathbb{A}'$ requires g_B hash operations. Thus, by ensuring that $g_B \ll n_R$, we can realize substantial savings in hash computation costs. There is, of course, the additional cost of computing each B from the input stream, which involves n_R hash computations. However, if we select the Bs carefully, then this cost can be amortized across the multiple aggregates A' that are computed from each B.

Hashing Model in the Presence of Limited Memory. In the above discussion, we assumed that the hash table for intermediate aggregate B has sufficient space to store the g_B (aggregated) result tuples for B until the end of an epoch (when they are used to compute the A's). However, as mentioned earlier in Section III-A, our query engine has a bounded amount M of memory for storing the intermediate hash tables. In such a memory-constrained scenario, the intermediate hash table for B may not have the required memory for storing all

[2]If queries have different time periods, then one option is to set the epoch duration equal to the gcd of the time periods; alternately, techniques such as time slicing [12] can also be used to calculate epoch durations.

[3]The g_{A_i}'s can be estimated by maintaining random samples of past stream tuples and applying standard sampling-based techniques such as the one by Charikar *et al.* [5].

the g_B result tuples, and partially aggregated result tuples may need to be pushed out from B's hash table during the epoch. It is thus possible that more than g_B result tuples are evicted from B's hash table over one epoch. Below, we describe our (partially aggregated) tuple eviction policy when hash tables need to store more tuples than the space allocated to them.

We implement a b-bucket hash table with capacity to store b result tuples as an array of b pointers. Each bucket pointer points to zero, one, or more tuples that hash into the bucket. We use chaining to link the multiple tuples that hash into a bucket. Note here that the space consumed by our b-bucket hash table is very close to the raw storage required for b tuples since the sizes of tuples are typically much larger than the pointers themselves. Our hash table implementation has the advantage that a bucket collision does not lead to an eviction if the table has fewer than b tuples or the incoming tuple is already in the table. Only if there are b tuples already present in the hash table, and a new tuple that is different from these b tuples arrives, does one of the tuples need to be evicted from the table. There are several policies such as the *least-recently-used* (LRU) policy or the *least-frequently-used* (LFU) policy, which can be used to evict tuples. In this paper, we use LRU because it is easy to implement, has very low overhead, and works well in practice.

One potential concern with chaining is that chains may become very long, substantially increasing the CPU overhead for every update. However, it has been shown that if at most b tuples are stored in a b-bucket hash table, then the length of a chain is at most $\log(b)$ with high probability [9]. To verify this, we conducted experiments to measure the worst-case and average chain lengths observed for b values ranging from $10,000$ to $100,000$ (which are the bucket sizes that we expect to see in practice). Averaging over 1000 runs, we found that the worst-case chain length lies between 6 and 8, while the average chain length (obtained after ignoring the empty buckets) is always around 1.6. This clearly indicates that the overhead associated with chaining is extremely small, especially when compared to the hash computation costs, which dominate the overall running time.

Our hashing model results in much smaller eviction rates compared to the hash model in [14]. The hash tables in [14] do not employ chaining, and so every collision leads to a tuple being evicted (even if the hash table is not full). In contrast, in our case, because of chaining, collisions do not cause evictions when the hash table has fewer than b tuples. Further, unlike [14], we are able to derive clean analytical formulas for the eviction rates of our hash tables for both Zipf and Uniform data distributions. ([14] only considers Uniform distribution.) As we show later in Sections VI and VII, the eviction rate depends on the amount of memory allocated for a particular hash table as well as on the statistical distribution satisfied by the tuples.

D. Query Plans

From the above discussion, it follows that to compute a good query execution plan (with low computational overhead), we need to answer the following two questions: (1) *What is the best set of intermediate aggregates B to instantiate?*, and (2) *How should the available memory M be allocated among the various intermediate aggregates?*

Each of the above two questions is difficult in its own right. But when considered together, the situation becomes even more tricky. This is because the two questions are actually interlinked and involve trade-offs. For instance, if we choose to instantiate more intermediate aggregates, then each intermediate will get less memory, which will result in higher eviction rates and neutralize the benefit of having intermediates in the first place. In general, the choice of intermediates to materialize will depend on the size of aggregates, data distribution of aggregate tuples, and the amount of available memory. We illustrate the various trade-offs using an example.

Example 1: Consider a stream with attributes a, b, c and d. Let $\mathbb{A}$ consist of the following three aggregates: $A_1 = \{a, b\}$, $A_2 = \{b, c\}$, and $A_3 = \{c, d\}$. Figure 1 shows three strategies for computing these aggregates.

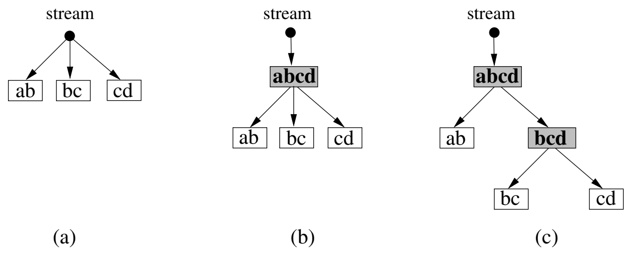


Fig. 1. Possible aggregate processing strategies.

Strategy 1. The naive strategy is to compute each aggregate A_i is directly from the stream (see Figure 1(a)). In this case, the number of hash operations is simply $3 \cdot n_R$.

Strategy 2. As shown in Figure 1(b), this strategy instantiates a new aggregate $B_1 = \{a, b, c, d\}$ that contains all the aggregates A_i. In this case, the entire memory M is allocated to B_1, and so B_1's hash table has $b_1 = M$ buckets. (For simplicity, in this example, we assume that the size of each hash table entry is 1 unit.) If f_1 is the eviction rate of B_1, then the total number of tuples pushed out from B_1's hash table is $b_1 + f_1 \cdot n_R$. Of these, $f_1 \cdot n_R$ tuples are evicted during the epoch and b_1 additional tuples are pushed out at the end of the epoch. Note that $b_1 + f_1 \cdot n_R \geq g_{B_1}$, and further if $b_1 = g_{B_1}$, then $f_1 = 0$. Thus, with Strategy 2, n_R stream tuples are inserted into the hash table for B_1, and $b_1 + f_1 \cdot n_R$ of B_1's tuples are inserted into the hash tables for A_1, A_2, and A_3. Thus, the total number of hash operations is given by $n_R + 3 \cdot (b_1 + f_1 \cdot n_R)$.

Strategy 3. This strategy takes the idea in Strategy 2 a bit further and instantiates one more aggregate $B_2 = \{b, c, d\}$ in addition to B_1 (see Figure 1(c)). Thus, the available memory M is now split between the two intermediates. Let B_1's and B_2's hash tables contain b'_1 and b'_2 buckets, respectively. Further, let f'_1 and f'_2 be the eviction rates of B_1 and B_2, respectively. (Note that $b'_1 \leq b_1$, which implies that $f'_1 \geq f_1$.) Now, the number of tuples pushed out from B_1's hash table into A_1 and B_2's hash tables is $b'_1 + f'_1 \cdot n_R$, and B_2 in turn pushes out $b'_2 + f'_2 \cdot (b'_1 + f'_1 \cdot n_R)$ of these tuples. Thus, the total number of hash operations is equal to $n_R + 2 \cdot (b'_1 + f'_1 \cdot n_R) + 2 \cdot (b'_2 + f'_2 \cdot b'_1 + f'_2 \cdot f'_1 \cdot n_R)$.

It is easy to see that the best solution will vary depending on the relationship between the values of n_R, g_{B_1}, g_{B_2}, and M. In particular, if $n_R \gg g_{B_1}$ and M is reasonably large so that f_1 is low, then $b_1 + f_1 \cdot n_R \approx g_{B_1}$ and Strategy 2 will result in much fewer hash computations compared to Strategy 1. On the other hand, if $n_R \approx g_{B_1}$ or M is small causing f_1 to be high, then $b_1 + f_1 \cdot n_R \approx n_R$ and Strategy 1 will likely be computationally more efficient that Strategy 2. Finally, if $n_R \gg g_{B_1} \gg g_{B_2}$ and M is reasonably large so that f'_1 and f'_2 are both low, then $b'_2 + f'_2 \cdot (\cdots) \approx g_{B_2}$ and Strategy 3 turns out to be the best solution.

Observe that each of the query plans considered above is essentially a directed tree with the root node corresponding to the stream, and other nodes corresponding to intermediate and input aggregates. A directed edge in the tree indicates that the destination aggregate is computed from the source aggregate. We next formalize this using the notion of aggregate trees.

Aggregate Tree. An aggregate tree T is a directed tree with (1) a special *root* node corresponding to the input stream, and (2) other nodes corresponding to aggregates. We use N_i to denote a node in the tree, and T_i for the subtree rooted at N_i. Without loss of generality, we assume that N_0 is the root node of T. A directed edge $\langle N_1, N_2 \rangle$ from node N_1 to node N_2 implies that the aggregate for N_1 is used to generate the aggregate for N_2. We use b_i to denote the space (in terms of hash buckets) allocated to N_i and s_i to denote the size (in bytes) of a single hash table entry at node N_i, which includes the tuple, the measure attribute value, and the pointer to the next entry. It follows that the total memory required at node N_i for b_i buckets is $b_i s_i$. Let g_i denote the number of distinct groups in the aggregate at N_i. Then, g_i can be viewed as the maximum storage requirement at N_i, while $b_i \leq g_i$ is the actual assignment.

Note that each node that corresponds to an aggregate query from $\mathbb{A}$ is given its complete memory requirement. Otherwise, it can lead to incorrect output or loss of data, neither of which is desirable. It is easy to incorporate these requirements since they can be given full allocations separately in the beginning; the goal then would be to distribute the remaining available memory M among the other nodes (corresponding to intermediate aggregates) of the tree. Thus, we will require the b_i's to satisfy the relation $\sum_i b_i s_i \leq M$, with the implicit understanding that we only consider the intermediate nodes N_i for allocation of memory M and ignore the query nodes.

Intuitively, an aggregate tree corresponds to a query plan capable of generating answers for every aggregate contained in the tree. The plan for a tree specifies the actions performed during the epoch as well as at the end of the epoch to generate aggregates.

•*During the epoch.* As new stream tuples arrive at the root node, they are inserted into the hash tables of each of the root's children. It is possible that an insertion may result in another (partially aggregated) tuple getting evicted in accordance with the LRU policy. Every tuple that is evicted from a node N_i is inserted into the hash tables of the child nodes of N_i. We denote the eviction rate for node N_i by f_i.

•*End of the epoch.* At the end of every epoch, all the tuples are flushed out from each node. This flushing of tuples is done in a top-down fashion, starting with the children of the root node down to the leaves. Thus, for a node N_i, once the tuples from its parent have been flushed out, all the tuples in N_i's hash table are evicted and aggregated into the hash tables of each of its children.

IV. Problem Formulation

In this section, we present our query execution cost model and then formally define the problem that we address in this paper.

We first estimate the hash computation costs incurred by aggregate tree T. Let p_i denote the index of the parent node of N_i, *i.e.*, N_{p_i} is N_i's parent. Let out_i denote the size of the stream coming out of node N_i. Observe that for the root node $out_0 = n_R$. For the remaining (non-root) nodes N_i, out_i can be written in terms of out_{p_i} and b_i as follows.

$$out_i = b_i + f_i \cdot out_{p_i} \qquad (1)$$

where f_i denotes the fraction of incoming tuples that are evicted out of N_i during an epoch. The first term on the RHS of Equation (1) is simply the number of distinct tuples stored in the hash table at N_i; these are the tuples that will be evicted at the end of the epoch. The second term corresponds to the tuples evicted out of the node during the epoch due to the limited memory allocated at the node, *i.e.*, because $b_i < g_i$. This term is determined by two factors: the input at the node, which is out_{p_i}, and f_i, which depends on the values of b_i, g_i, and the input distribution. (We show how to compute f_i for Zipf and Uniform distributions in Sections VI and VII, respectively.) Thus, the value of out_i, which is the number of tuples evicted from N_i, is given by the sum of the number of tuples evicted during the epoch and at the end of the epoch.

Note that the value of out_i cannot be less than g_i since that is the number of distinct groups observed at N_i. Further, note that for $b_i = g_i$, there is enough memory for all the tuples and hence, no tuple will be evicted during the epoch, *i.e.*, f_i is 0. Then, Equation (1) implies that $out_i = g_i$. Thus, the minimum value of out_i occurs at $b_i = g_i$. In other words, if there is sufficient memory available, then every node N_i will be given its maximum requirement $g_i s_i$.

Let h_i denote the sum of the hashing costs at the child nodes of N_i, *i.e.*, h_i is the sum of the costs of inserting a tuple evicted from N_i into the hash tables of N_i's children. Then, the total cost of tree T is given by the following.

$$C(T) = \sum_i out_i h_i \qquad (2)$$

Thus, the cost of aggregate tree T reflects the total computation cost of producing all the aggregates in the tree. Hence, our problem of finding a good query plan reduces to the following.

Problem Statement. Given a set of aggregates $\mathbb{A}$, compute the minimum-cost aggregate tree T that contains all the aggregates in $\mathbb{A}$, subject to the constraint that the sum of the memory allocations given to each (intermediate) node in the tree is at most M. □

It turns out that this problem is NP-hard even when there are no constraints on the total available memory. The reduction is from the *subset-product* problem, which is known to be NP-hard [8]. We omit this proof due to lack of space. Given this result, the NP-hardness of our problem easily follows given that we can assign a sufficiently-large value to M ($= n_R|\mathbb{A}|2^{|\mathbb{A}|}$) such that any aggregate tree is feasible, thus reducing it to the case where there are no memory constraints.

Note that, unlike [14] which looks to optimize only during-epoch costs, our cost model includes both during-epoch and end-of-epoch costs. During-epoch costs are more prominent when the available memory is small, but as the memory size increases, end-of-epoch costs start becoming more dominant. Thus, since our approach optimizes both costs simultaneously, our aggregate tree computation algorithms work well for a broad range of memory sizes.

V. Aggregate Tree Construction

In this section, we present an efficient greedy heuristic for computing a good aggregate tree. Algorithm 1 contains the pseudo-code for this greedy heuristic. The heuristic applies a series of local modifications to the tree: at each step, it selects the modification that leads to the biggest cost reduction. In particular, it considers the following two types of local tree modifications in each iteration: (1) Addition of a new aggregate C obtained as a result of merging sibling aggregates A, B (Steps 4–14), and (2) Deletion of an aggregate A (Steps 15–23). For each modification, we compute the best possible memory allocation and estimate the minimum cost possible using that tree. In each iteration, the local modification that results in the biggest cost decrease is applied to the tree. The heuristic terminates when the cost improvement due to the best local modification falls below a (small) constant threshold ϵ.

Now, let us look at the rationale behind the two local modifications. Let A, B be a pair of aggregates whose union C is much smaller in size than their current parent P, and let k be the number of children of P that are subsets of C. Then, our first modification leads to $(k-1)\cdot out_P - k\cdot out_C \approx (k-1)\cdot out_P$ fewer hash operations by adding the new aggregate C to the tree. This is because C's memory requirements are small because of its smaller size, and also $out_C \ll out_P$. Thus, generating C from P requires out_P hash computations, and then generating A, B and the other children from C incurs an additional $k \cdot out_C$ hash operations, while generating all the children directly from P requires $k \cdot out_P$ operations. The second modification considers the opposite situation when the size of an aggregate A is close to the size of its parent P in the tree – in this case, the extra cost of generating A from P does not offset the cost reduction when A's children are generated from A instead of P. Thus, it is more beneficial in this case to delete A from the tree and compute A's children directly from P.

Note that, in the worst case, we may need to consider a quadratic (in the number of input aggregates) number of local modifications in every iteration. To compute the optimal allocation of the available memory to the nodes of each modified tree, our heuristic invokes procedure **AllocateMemory**.

Algorithm 1 Greedy heuristic for finding aggregate tree.

```
Greedy($\mathbb{A}$)
 1: $T_{best}$ is initialized to the aggregate tree in which all $A_i \in \mathbb{A}$ are
    children of the root node.
 2: while $T_{best}$ cost improves by at least $\epsilon$ do
 3:   $T_{cur} \leftarrow T_{best}$;
 4:   for all pairs of sibling aggregates $A, B$ in $T_{cur}$ do
 5:     Let aggregate $C = A \cup B$;
 6:     Let $P$ be the parent of $A, B$ in $T_{cur}$;
 7:     Let $X$ denote the set of $P$'s children that are subsets of $C$;
 8:     Let $T$ be the tree derived from $T_{cur}$ by (1) adding $C$ as $P$'s
        child, and (2) making all the nodes in $X$ as the children of
        $C$;
 9:     $\mathbb{M} \leftarrow$ AllocateMemory($T$, $M$);
10:     Let $cost(T, \mathbb{M})$ be the cost of $T$ with allocation $\mathbb{M}$;
11:     if $cost(T, \mathbb{M}) < cost(T_{best}, \mathbb{M}_{best})$ then
12:       $T_{best} \leftarrow T$,   $\mathbb{M}_{best} \leftarrow \mathbb{M}$;
13:     end if
14:   end for
15:   for all aggregates $A \notin \mathbb{A}$ in $T_{cur}$ do
16:     Let $P$ be the parent of $A$ in $T_{cur}$;
17:     Let $T$ be the tree derived from $T_{cur}$ by deleting $A$, and
        making $A$'s children the children of $P$;
18:     $\mathbb{M} \leftarrow$ AllocateMemory($T$, $M$);
19:     Let $cost(T, \mathbb{M})$ be the cost of $T$ with allocation $\mathbb{M}$;
20:     if $cost(T, \mathbb{M}) < cost(T_{best}, \mathbb{M}_{best})$ then
21:       $T_{best} \leftarrow T$,   $\mathbb{M}_{best} \leftarrow \mathbb{M}$;
22:     end if
23:   end for
24: end while
25: return $T_{best}$;
```

The next two sections describe in more detail our memory allocation algorithms for Zipf-like and Uniform distribution of the tuple values, respectively.

VI. Space Allocation for Zipf-like Distributions

Several empirical studies have shown the distribution of IP addresses and port numbers in Internet traffic to be Zipf-like [4]. Under a Zipf-like distribution with parameter α, the k^{th} ranked tuple is expected to appear $\frac{1}{k^{\alpha}}$ times the occurrence of the most commonly appearing tuple. (An α value of 0 reduces this distribution to the Uniform distribution in which every tuple is equally likely.) We studied the NetFlow record traces for two backbone router nodes from the Abilene observatory [1], and found that many of the group-by attributes do indeed follow Zipf-like distributions. In particular, the Zipf parameters for $\{\mathrm{srcIP}\}$, $\{\mathrm{srcIP, dstIP}\}$ and $\{\mathrm{srcIP, dstIP, srcPort, dstPort}\}$ were observed to be 0.85, 0.67 and 0.48 respectively.

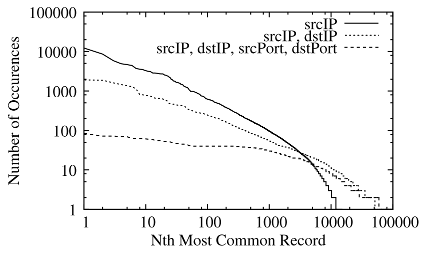


Fig. 2. Plot of record frequencies illustrating a Zipf-like distribution.

As the curves in Figure 2 indicate, the value of α depends on the set of group-by attributes under question. For instance, there are some attributes (such as ToS) whose occurrence is more uniform, and hence has an α close to zero. The value of α may also vary as the number of attributes in the set increases. In particular, we observed that α for $\{\texttt{srcIP}, \texttt{dstIP}, \texttt{srcPort}, \texttt{dstPort}\}$ is considerably smaller than the α for $\{\texttt{srcIP}\}$ (see Figure 2). In some cases when the number of attributes is large, we observed that the distribution is very close to Uniform (*i.e.*, a Zipf parameter of 0). As a result, in practice, one can encounter a set of queries all of which follow the Uniform distribution. Hence, we also present allocation algorithms for such a case in Section VII.

A. Eviction Rates

We now show how to determine f_i, *i.e.*, the probability of a tuple getting evicted out of node N_i. In the following lemma, we assume that perfect LFU is used to choose the tuple that is evicted. There are two reasons for doing so: it is considerably easier to analyze perfect LFU for Zipf and more importantly, the results in [4] as well as our experiments show that, in practice, the performance of LRU closely matches that of perfect LFU in terms of cache hits/misses.

Lemma 6.1: Assuming perfect LFU, if the input distribution at a node N_i is Zipf-like with parameter α_i, then the fraction of tuples that are pushed out from N_i during an epoch is approximately $1 - (\frac{b_i}{g_i})^{1-\alpha_i}$ for $0 \leq \alpha_i < 1$, and $1 - \frac{\log(b_i)}{\log(g_i)}$ if $\alpha_i = 1$. □

Figure 3 compares the formula from Lemma 6.1 with the actual evictions observed in practice when LRU is used. As the plots indicate, the analytical model is reasonably accurate for modeling the actual eviction rates.

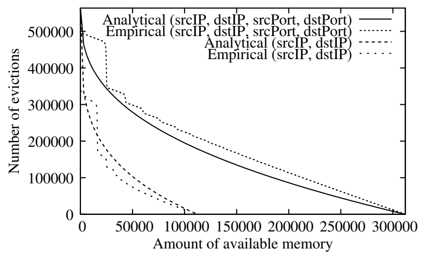


Fig. 3. Plot shows the analytical and empirically-observed eviction rates (using LRU) as the amount of available memory is varied.

Using Lemma 6.1 and the fact that the minimum value of out_i occurs at $b_i = g_i$, it is easy to show that for a given g_i and out_{p_i}, out_i is a decreasing function of b_i over the range $[0, g_i]$. It follows that the cost of T is a decreasing function of the available memory. This is because, the extra memory can be given to any node N_i in the tree, and as discussed above, this will result in a smaller out_i. Intuitively, this implies that the input to the children of N_i reduces resulting in a lower cost for T_i. This argument shows that there is at least one possible memory allocation that reduces the cost, and hence the best allocation will also result in a lower cost.

Our objective is to allocate memory to the nodes of an aggregate tree T such that the cost $C(T)$ is minimized subject to $\sum_{N_i \in T} b_i s_i \leq M$. For the sake of brevity, we refer to C as the cost of the memory allocation in the rest of this section. Recall that $out_i h_i$ corresponds to the cost incurred due to hashing at the children of N_i and does not include the cost incurred at N_i. Henceforth, we refer to $out_i h_i$ as the cost contributed by node N_i since the allocation at N_i determines the value of out_i.

We also define two specific instantiations of C, which correspond to lower and upper bounds on the value of C.

$$C_{min} = \sum_i g_i h_i, \quad C_{max} = \sum_i n_R h_i \qquad (3)$$

The first relation follows from our earlier observation that $out_i \geq g_i$. The second relation follows from the fact $out_i \leq n_R$, since n_R is the total number of tuples.

C^i_{min} and C^i_{max} are defined analogously and correspond to the minimum and maximum costs that can be contributed by the subtree rooted at N_i.

$$C^i_{min} = \sum_{N_j \in T_i} g_j h_j, \quad C^i_{max} = \sum_{N_j \in T_i} n_R h_j$$

B. Algorithm Overview

In this section, we show how to distribute the available memory among the various nodes such that the total cost is minimized. In order to do this, we present a dynamic programming algorithm that solves the inverse problem: *given an input size n, what is the minimum amount of memory needed to achieve a target cost of C.*

The basic idea is to compute the optimal memory allocation in each subtree for every possible pair of n and C values, where n is the size of the input at the subtree and C is the cost contributed by that subtree. Given this information for every possible C at the root node, since n is known ($= n_R$), we can perform a simple binary search in order to identify the minimum cost corresponding to a total memory allocation of M. (Note that binary search is applicable here because the cost is a decreasing function of the available memory. In particular, at every step, we can keep halving the range of C values under consideration by comparing M with the memory needed at the midpoint of the range.)

Note that the input at a node N_i is at least g_{p_i}, the minimum value of out_{p_i}, and at most n_R, the total number of tuples in the raw stream. Similarly, the cost contributed by a subtree T_i is at least C^i_{min} and at most C^i_{max}. Hence, the optimal memory allocation in T_i (*i.e.*, the subtree rooted at N_i) needs to be computed for each (n, c) pair, where $n \in [g_{p_i}, n_R]$ and $c \in [C^i_{min}, C^i_{max}]$. Thus, at each node N_i, a table $M(N_i)$ of size at most $n_R \cdot C_{max}$ is created, with each entry in the table containing a vector corresponding to the optimal allocations at the node and its children. The entries in these tables can be computed in a bottom-up fashion as follows.

1) Let c_i denote the portion of c that is contributed by N_i due to the hashes that are performed at the child nodes of N_i. Since $g_i h_i$ is the minimum value that c_i can take, we iterate over all values in the range $[g_i h_i, c]$. For each value, we can calculate the corresponding b_i by performing a binary search over the interval $[0, g_i]$. In particular, at each step, we can halve the interval to which b_i can belong by comparing the cost of mid-point with c_i.

2) Since c_i equals $out_i \cdot h_i$, the input to the children of N_i for a given c_i is simply $\frac{c_i}{h_i}$. We now need to calculate the optimal allocation across the child subtrees of N_i given that the sum of the costs contributed by them can be at most $c - c_i$. Since the input size to these child subtrees is known ($= out_i$), we can iterate over all combinations that result in a sum of $c - c_i$ and choose the combination that results in the minimum total memory used. More specifically, suppose that $V_1, V_2, \ldots, V_k$ are the children of N_i. Then, we construct a table with k rows as follows. The i^{th} row of this table computes the minimum amount of memory necessary to ensure that the total cost of the i subtrees rooted at $V_1, V_2, \ldots, V_i$ is at most c' for every $c' \leq c - c_i$. Note that the first row can be easily filled using the values from $M(V_1)$, while the i^{th} row can be filled using the information from $(i-1)^{th}$ row and $M(V_i)$. In particular, the latter can be filled by considering all possible splits of the cost between the subtree rooted at V_i and the subtrees rooted at $V_1, \ldots, V_{i-1}$.

In essence, the above approach gives us a dynamic programming algorithm that is polynomial in terms of the values of n_R and C_{max}. Thus, we have a pseudo-polynomial algorithm for computing the minimum memory allocation for a given cost. We now design an FPTAS by quantizing the ranges $[g_{p_i}, n_R]$ and $[C^i_{min}, C^i_{max}]$. In other words, we get a polynomial time algorithm by computing the optimal allocations at fewer values of n and c depending on the approximation criteria ϵ.

C. Detailed Description

We now describe the FPTAS (see Algorithm 2) in more detail. We first describe how to reduce the number of table entries to be computed and then discuss the actual table construction.

Quantization: As mentioned earlier, we need to quantize n and C so that the computation of the optimal memory allocation is performed only for a polynomial number of values of n and C. Though this results in a loss of optimality, we show later that this error can be bounded in terms of the quantization parameter.

Let $C_{ij} = C^i_{min}\beta^j$, and $n_{ij} = g_{p_i}\beta^j$, where β (> 1) depends on the approximation criteria for the FPTAS and will be fixed later. To see why we consider these particular values of C_{ij} and n_{ij}, recall that g_{p_i} is the smallest possible value for the input size at node N_i, and C^i_{min} is the smallest possible value of the cost of T_i.

Let M_{ijk} denote the minimum amount of memory needed for T_i given that the size of the input to T_i is n_{ij} and the cost contribution of T_i is C_{ik}.[4] Thus, M_{ijk} corresponds to a table of size $\lceil \log_\beta \frac{n_R}{g_{p_i}} \rceil \times \lceil \log_\beta \frac{C^i_{max}}{C^i_{min}} \rceil$ for some $\beta > 1$. Here, for the sake of simplicity of exposition, we assume that the allocation of every node in T_i is also stored in M_{ijk}. Instead of this, we can reduce the storage requirement by storing the allocation at N_i and indexes of the tables at the children of N_i. This will necessitate a minor change in the algorithm. At the end, a top-down pass must be performed in order to compute the memory allocation at all the nodes.

[4]Strictly speaking, this is incorrect because the cost of this allocation may be greater than C_{ik}. However, we later show that the actual cost can be off from C_{ik} by a factor of β^x for some x. We refer to C_{ik} in the definition for the sake of simplicity.

Algorithm 2 FPTAS

```
AllocateMemory($T$, $M$)
1: for each node $N_i$ in the post-order traversal of $T$ do
2:   for all $n_{ij} \in [g_{p_i}, n_R]$ such that $n_{ij}$ is $g_{p_i}\beta^j$ for some $j$ do
3:     for all $C_{ik} \in [C^i_{min}, C^i_{max}]$ such that $C_{ik} = C^i_{min}\beta^k$ for some $k$ do
4:       $m \leftarrow \infty$;
5:       for all $c_{i\ell} \in [g_i h_i, C_{ik}]$ such that $c_{i\ell} = g_i h_i \beta^\ell$ for some $\ell$ do
6:         Use binary search over $[0, g_i]$ to compute the smallest $b_i$ such that $(b_i + f_i \cdot n_{ij})h_i \leq c_{i\ell}$;
7:         $b'_i \leftarrow$ MinSumChildren$\big(children(N_i), \ell, \beta^2 C_{ik} - \beta c_{i\ell}\big)$;
8:         $m \leftarrow \min(m, b_i + b'_i)$;
9:       end for
10:      $M_{ijk} \leftarrow m$;
11:    end for
12:  end for
13: end for
14: Use binary search in the row $M_{0j}$ to determine the largest $M_{0jk}$ smaller than $M$, where $j = \left\lceil \log_\beta \frac{n_R}{g_{p_i}} \right\rceil$;
15: return $M_{0jk}$;

MinSumChildren($\mathcal{C}$, $y$, $cost$)
1: Let $N_x$ be some node in $\mathcal{C}$;
2: $S[1] \leftarrow M_{xy}$, $j \leftarrow 2$;
3: for each $N_i$ in $\mathcal{C} \setminus \{N_x\}$ do
4:   for each $k$ in $\big[1, \lceil \log_\beta(cost) \rceil\big]$ do
5:     $S[j,k] \leftarrow \min_{1 \leq k' \leq k} \big(M_{iyk'} + S[j-1, k-k']\big)$;
6:   end for
7:   $j \leftarrow j + 1$;
8: end for
9: return $S\big[|C|, \lceil \log_\beta(cost) \rceil\big]$;
```

Table Computation: We split M_{ijk} into two distinct allocations: the allocation at node N_i and the allocations for all the child subtrees of N_i. Once these two allocations can be computed, computing M_{ijk} is just a simple minimization over the sum of these two quantities.

$$M_{ijk} = \min_{\ell} \; (P_{ij\ell} + Q_{ik\ell}) \tag{4}$$

Here, $P_{ij\ell}$ corresponds to the allocation at N_i and $Q_{ik\ell}$ corresponds to the allocations in the subtrees rooted at the children of N_i. Before formally defining $P_{ij\ell}$ and $Q_{ik\ell}$, we make one observation. The number of possible ways in which C_{ik} can be split between N_i and the subtrees of its children depends on the value of C_{ik}; hence, we again need to quantize the cost contributed by node N_i in powers of β.

Let $c_{i\ell}$ be $g_i h_i \beta^\ell$, and let $P_{ij\ell}$ denote the best allocation at N_i given that the input to N_i is n_{ij} and the cost contributed by N_i is at most $c_{i\ell}$. Thus, using Equation (1) and Lemma 6.1, $P_{ij\ell}$ is given by the following.

$$P_{ij\ell} = \min\left\{ b \cdot s_i \;\middle|\; \left(b + n_{ij}\left(1 - \frac{b^{1-\alpha_i}}{g_i^{1-\alpha_i}}\right)\right) h_i \leq c_{i\ell} \right\}$$

Note that the expression that is being compared to $c_{i\ell}$ in the above minimization is simply the cost due to the hashes at the children of N_i. $P_{ij\ell}$ can be easily computed using a binary search on b over the range $[0, g_i]$ since the cost is a decreasing function of b.

We now describe how to compute $Q_{ik\ell}$ using the allocations at the child subtrees of N_i. We define $Q_{ik\ell}$ as follows in terms of $M_{x\ell y_x}$, where N_x is a child of N_i.

$$Q_{ik\ell} = \min \left\{ \sum_x M_{x\ell y_x} \mid i = p_x \text{ and } \sum_x C_{xy_x} \le \beta^2 C_{ik} - \beta c_{i\ell} \right\}$$

Intuitively, we select individual allocations for nodes N_x such that the following two conditions hold.

• $n_{x\ell}$ is the input size to N_x when N_i is allocated $P_{ij\ell}$. This follows from the fact that $n_{x\ell} = g_{p_x}\beta^\ell = g_i\beta^\ell = \frac{c_{i\ell}}{h_i}$.

• The sum of the costs of the subtrees rooted at all the children of N_i is within some function of the target cost $(C_{ik} - c_{i\ell})$. The particular choice of the function is dictated primarily by our proof.

Note that the third subscript y_x in $M_{x\ell y_x}$ determines the cost contribution made by child node N_x and can vary across the different children. Hence, there is a need to further subscript it with the node index as shown. As we shall see later, the additional β factors in the definition of $Q_{ik\ell}$ are needed to ensure that the allocation M_{ijk} is at most the optimal allocation for a cost of C_{ik} and an input size of n_{ij}. This simplifies the correctness proof considerably; however, it implies that the actual cost of the allocation M_{ijk} could be higher than C_{ik}. Later, we also give an upper bound on the cost of M_{ijk} in terms of C_{ik} and β.

It turns out that $Q_{ik\ell}$ can be determined using a dynamic programming approach that is similar to the one used for solving the traditional Knapsack problem [8]. In particular, we create a table S of size $D_i \times \lceil \log_\beta(\beta^2 C_{ik} - \beta c_{i\ell}) \rceil$, where D_i is the number of children of N_i. As discussed in Section VI-B, each term $S(d, c)$ in the table corresponds to the minimum memory allocation needed to ensure that the first d children of N_i contribute a cost of at most c. $S(d, c)$ can be easily defined in terms of $S(d-1, c')$ $(c' \le c)$ and $M_{x\ell y_x}$ where N_x is the d^{th} child of N_i. In particular, we can iterate over all the $\log_\beta(C^x_{max})$ values in the row M_{xy}; for each of these costs, we can use the row $S(d-1)$ to determine the minimum total memory necessary for the first $d-1$ children.

Now, from the definition of $P_{ij\ell}$ and $Q_{ik\ell}$, it follows that if $P_{ij\ell}$ and $Q_{ik\ell}$ are calculated in an optimal fashion, then the cost for the allocation given by M_{ijk} is at most $\beta^2 C_{ik}$. Later, in Section VI-D, we give a more exact bound on this cost by taking into account the fact that the calculation for $Q_{ik\ell}$ depends on values of $M_{x\ell y_x}$, which are also approximations.

Finally, once all the M_{ijk}'s are calculated, the minimum-cost allocation can be determined by identifying the entry in the row M_{0j} such that the total memory allocation is at most M. (Here, j corresponds to the input size of n_R.) Since the cost is a decreasing function of the memory, it follows that the memory needed also decreases as the target cost increases. In other words, $M_{0jk} \le M_{0jk'}$ for all $k' > k$. Hence, the minimum-cost allocation corresponding to a total memory allocation of at most M can be computed using a binary search over the row M_{0j}.

D. Analysis

The following theorems analyze the running time of Algorithm 2 and prove the necessary approximation bounds to show that it is an FPTAS.

Theorem 1: The asymptotic time complexity of Algorithm 2 is $O(\log^5_\beta(n_R)|T|\Delta)$. □

Theorem 2: Algorithm 2 gives a β^{2L+1} approximation, where L is the number of levels in the tree. □

It easily follows from Theorems 1 and 2 that Algorithm 2 is an FPTAS for minimizing the cost of the tree subject to a constraint on the total available memory. In particular, we obtain a $(1+\epsilon)$-approximation by choosing β to be $1 + \frac{\epsilon}{2L+1}$. Then, by Theorem 2, the approximation factor of Algorithm 2 is $(1 + \frac{\epsilon}{2L+1})^{2L+1}$, which is approximately $(1+\epsilon)$. By Theorem 1, the running time of Algorithm 2 is given by

$$O\left(\frac{\log^5(n_R)}{\log^5(\beta)}|T|\Delta\right) = O\left(\left(\frac{L}{\epsilon}\right)^5 \log^5(n_R)|T|\Delta\right)$$

(This follows from the relation $\log(1+x) \approx x$ for small x.) Thus, Algorithm 2 has a running time that is polynomial in the size of the problem as well as in $\frac{1}{\epsilon}$. Hence it is an FPTAS for the problem.

VII. Space Allocation for Uniform Distribution

In this section, we focus on the scenario in which the input distribution at every node is Uniform. We prove a certain property about optimal memory allocations, and then show how to exploit this property to obtain a more efficient algorithm. Before proceeding, we first compute the eviction rate under a Uniform distribution.

Lemma 7.1: If the input distribution at N_i is Uniform, then the fraction of queries pushed out from N_i during an epoch is $1 - \frac{b_i}{g_i}$. □

Unlike Lemma 6.1, Lemma 7.1 is independent of the policy (LRU or LFU) that is used to choose the tuple to be pushed down. Lemma 7.1 implies the following, where n_i denotes the size of the input at node N_i, and m_i is the memory allocated at N_i.

$$out_i = \frac{m_i}{s_i} + n_i\left(1 - \frac{m_i}{g_i s_i}\right) \tag{5}$$

(This follows from the fact that $b_i = \frac{m_i}{s_i}$.)

We now prove that one optimal memory allocation is the *all-or-nothing* policy, under which every node (except one) gets an allocation equal to its maximum requirement or nothing.

Theorem 3: Given an aggregate tree T in which the input distribution for every aggregate is Uniform, there exists an optimal memory allocation in which $m_i = 0$ or $m_i = g_i s_i$ holds for all nodes except (at most) one node. □

Given Theorem 3, we can modify Algorithm 1 as follows. Note that nodes that get an allocation of 0 are, in fact, not aggregating anything and can be safely removed from the aggregate tree. Thus, we need to consider an aggregate as a candidate for addition during the steps 4–14 of Algorithm 1 only if after including that node, we can give full allocation to all nodes except one. Thus, while computing the cost, we can iterate over all the possibilities in which only one node is allocated less than its maximum requirement.

VIII. Performance Study

In this section, we discuss our experimental results in which we compare our approach with the best-known approach. We describe our experimental setup and then discuss our results.

A. Experimental Setup

We implemented a simulated NetFlow Collector (NFC), which performs real-time aggregation on streaming NetFlow records, and ran this collector on a PC running Ubuntu Linux 7.04 with an Intel Xeon 3.0GHz processor and 2GB of RAM.

Real-life Data Sets. All our experiments were performed on NetFlow record traces obtained from the Abilene network, which is an Internet2 high-performance backbone network. We downloaded the NetFlow record traces for the Indianapolis (IPLSng) and New York (NYCMng) backbone routers from the Abilene observatory [1]. For IPLSng, the traces correspond to four consecutive five-minute intervals of data from 23:20 to 23:40 on October 16, 2006, with each interval containing approximately 400,000 records. The NYCMng trace consists of four consecutive five-minute intervals of data from 11:20 to 11:40 on May 8, 2006, each interval containing approximately 350,000 NetFlow records.

Aggregate Queries. In our experiment, we varied the number of queries from 20 to 78. It turns out if one query is a subset of another, then the Zhang *et al.* approach needs to be significantly modified while performing the memory allocations. (On the other hand, Algorithm 2 can easily handle such cases.) Hence, to avoid such scenarios, we considered all possible queries from the set formed by pairs of group-by attributes. Since the number of distinct attributes in a NetFlow record is 13, the maximum number of unique queries that we consider is $\binom{13}{2} = 78$.

Memory Allocation Strategies. We compared the following two memory allocation strategies with our greedy tree construction algorithm (Algorithm 1). We used Algorithm 1 for the approach of [14] as well, because their tree construction algorithm does not scale to the number of queries for which we ran our experiments.

1) *Supernode with Linear Combination* (SLC). Supernode with Linear Combination yielded the best results in [14]; hence, we use it as a benchmark for our experiments. As mentioned in Section III, SLC uses a hash table in which each bucket has size 1, *i.e.*, every collision results in an eviction. Under this model, they show that allocating in proportion to $\sqrt{g_i s_i}$ is optimal if the height of the aggregate tree is two. For taller trees, they combine each subtree rooted at level 2 into a single *supernode* X, and define g_X to be the sum of g_i's of all the nodes in the subtree. Once each supernode is allocated memory, the memory allocation algorithm is then recursively applied to the nodes within the supernode.

In [14], the end-of-epoch cost is not accounted for in the cost optimization function; instead [14] uses a seperate peak load constraint to bound the end-of-epoch cost. In our experiments, we tuned the peak load constraint each time by empirically finding the best value that yielded the smallest total cost and used this value when running this scheme.

2)*Fully Polynomial-Time Approximation Scheme* (FPTAS). We implemented the hashing model (with LRU) as described in Section III-C and Algorithm 2 as the memory allocation strategy. We used $\epsilon = 0.05$ in our experiments; this yielded excellent results and took only a few minutes to run.

Performance Metric.

As mentioned earlier, aggregrate query processing costs mainly comprise the CPU cycles for hashing, traversing chains and updating record entries, etc. These are proportional to the number of hash operations. Hence, we use the total number of hash operations as the performance metric, when comparing the performance of FPTAS and SLC.

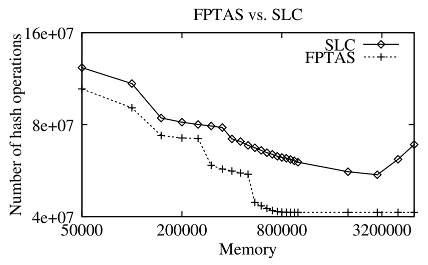
We remark that both the approaches took only a few minutes to produce query plans. We believe that this is sufficient given that plans will typically need to be generated infrequently; for instance, when there is a change in the query workload or a considerable change in the stream data distribution.

B. Results

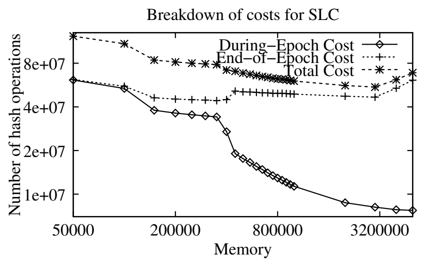
Comparison of FPTAS with SLC. Figure 4(a) compares the performance of FPTAS and SLC; in this experiment, we used all the 78 possible pairs of group-by attributes in this experiment over the 20 minutes of data from the IPLSng. The plot shows the number of hash operations required by the two approaches as the amount of memory available for intermediate aggregates is varied. (In our experiments, we measure the amount of available memory in terms of number of hash buckets.) As the available memory increases beyond 1M, our algorithm continues to generate the same query plan. This is because the hash tables of all the intermediate aggregate are allocated their full requirement, *i.e.*, $b = g$, and the addition of any new intermediate aggregate in the aggregate tree results in a higher cost.

Breakdown of costs. The primary focus of the SLC approach was on optimizing the intra-epoch processing cost, and it ignores the cost incurred at the end of an epoch. Thus, in cases in which the end-of-epoch cost dominates the overall cost, their solution does not give good performance. Figures 4(b) and 4(c) show the breakdown of the processing costs using SLC and FPTAS, respectively. For SLC, it is easy to see that although the cost during epoch decreases inversely with the memory allocation, the end-of-epoch cost fluctuates. Because of this, the overall cost of the query plans generated with SLC may actually *increase* as the amount of available memory increases. In fact, we do observe this around 3.2M of available memory (see Figure 4(b)). On the other hand, in our approach, since we account for end-of-epoch cost as well, the overall cost always reduces with increasing memory.

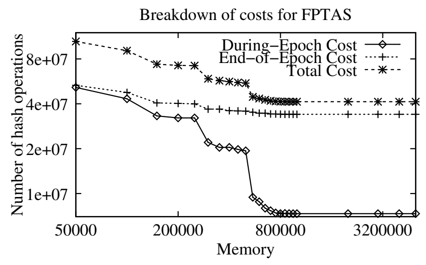
Additional benchmarking. In the previous experiment, we used a single query set for simplicity of exposition. We now present the results of repeating these tests several times using various randomly-generated query sets and memory constraints. We present the results for the NYCMng traces, which also indicates that our results are consistent across various datasets (which are both geographically and temporally separated). Figure 5 shows the results of this experiment. The three curves plotted correspond to three different limits on the amount of available memory (in terms of hash buckets): $\{$50K, 500K, 5000K$\}$. We varied the number of queries from 20 to 78. Each data-point represents the average processing of 3 different query sets of the specified size. As the graphs show, our approach consistently outperforms SLC. In particular, our approach results in a speedup of 50% as compared to SLC



(a)



(b)



(c)

Fig. 4. Inset (a) compares the number of hash operations with our approach versus SLC. Insets (b) and (c) show the breakdown of processing costs for SLC, and our approach, respectively.

when the amount of memory available is 5000K. (Here, by speedup we mean the percentage reduction in the number of hash operations performed in our approach compared to the approach of [14].)

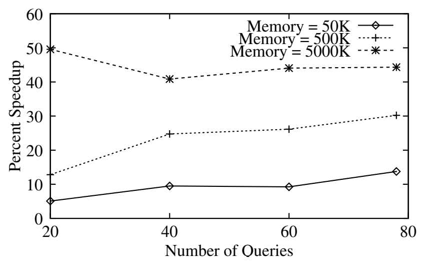


Fig. 5. Comparison of speedup for various query set sizes.

IX. Conclusions

In this paper, we re-investigated the problem of efficiently computing multiple aggregation queries over a data stream. We have made several important contributions to improve the execution and generation of query plans containing intermediate aggregates. We studied a new hashing model, which has lower eviction rates than the hash model considered in prior work, and which allows us to provide accurate analytical estimates for the number of hash operations. Based on these estimates, we presented a comprehensive query execution cost model and an efficient greedy heuristic for constructing good low-cost query plans as well as provably near-optimal and optimal algorithms for allocating the available memory to aggregates in the query plan when the input data distribution is Zipf-like and Uniform, respectively. Finally, we have also presented a detailed performance study with real-life IP flow data sets, which show that our multiple aggregates computation techniques consistently outperform the best-known approach of Zhang *et al.* [14].

References

[1] Abilene Observatory Data Collections. http://abilene.internet2.edu/observatory/data-collections.html.

[2] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *VLDB*, 1996.

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

[4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM*, pages 126–134, 1999.

[5] M. Charikar, S. Chaudhari, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *PODS*, 2000.

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

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

[8] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[9] G. Gonnet. Expected length of the longest probe sequence in hash code searching. *J. ACM*, 28(2):289–304, 1981.

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

[11] R. Huebsch, M. Garofalakis, J. Hellerstein, and I. Stoica. Sharing aggregate computation for distributed queries. In *ACM SIGMOD*, pages 485–496, 2007.

[12] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *ACM SIGMOD*, pages 623–634, 2006.

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

[14] R. Zhang, N. Koudas, B. Ooi, and D. Srivastava. Multiple Aggregations over Data Streams. In *SIGMOD*, 2005.

Appendix

Proof of Theorem 1

Proof: (Theorem 1) It is easy to see that the running time of Algorithm 2 is determined by the complexity of **MinSumChildren** and the number of times it is executed. It is easy to see that Line 8 is executed $O(|T|\log_\beta^2(C_{max})\log_\beta(n_R))$ times and the complexity of the procedure **MinSumChildren** is $O(\Delta\log_\beta^2(C_{max}))$, where Δ is the maximum degree of a node in T. Hence, by Equation (3), the running time of Algorithm 2 is $O(\log_\beta^5(n_R)|T|\Delta)$. ■

Proof of Theorem 2

We now present a series of Lemmas, which show that Algorithm 2 is a β^{2L+1} approximation, where L is the height of T.

Note that out_i is a linear, increasing function in the size of the input at N_i (see Equation 1). This implies that for a given memory allocation, if the input size increases by a factor x, then the cost increases by the same factor as well. Therefore, the minimum cost for the increased input size will also increase by *at most* the same factor. This is formally stated in the following lemma.

Lemma 9.1: The minimum cost of any subtree T_i for an input size of βout_{p_i} $(\beta > 1)$ is at most β times the minimum cost for an input size of out_{p_i}.

Proof: We show the following two things at N_i, the root of T_i: (a) the cost contributed by N_i increases by at most β (b) the input size to each of its children also increases by at most β. Inductively, it follows that the cost at each node of T_i and hence the total cost of T_i increases by a factor of at most β. Since the size of the input to N_i's children is given by out_i and the cost contributed by N_i is simply $out_i h_i$, it suffices to show that out_i changes by a factor of at most β. Now, by Equation (1), out_i is $(b_i + f_i \cdot out_{p_i})$. The Lemma follows because $\beta \cdot out_{p_i} \cdot f_i + b_i < \beta(out_{p_i} \cdot f_i + b_i)$ for any $\beta > 1$. ∎

The following two lemmas are necessary for comparing the allocation given by Algorithm 2 with an optimal allocation.

Lemma 9.2: The memory allocation by Algorithm 2 for a cost of βC_{ik} with an input size of βn_{ij} is at most the memory allocation for a cost of C_{ik} with an input size of n_{ij}.

Proof: The required result follows by showing that the memory allocation in T_i for input size n_{ij} and cost C_{ik} is also a valid allocation for input size βn_{ij} and cost βC_{ik}. This follows from Lemma 9.1 which states that the cost of that allocation for an input size of βn_{ij} is at most βC_{ik}. Since this is only one possible allocation, there may exist better allocations for βC_{ik} and βn_{ij}. ∎

Lemma 9.3: Let M^o_{ijk} denote the optimal memory allocation in T_i for input size n_{ij} and cost C_{ik}. Then, $M_{ijk} < M^o_{ijk}$.

Proof: We prove this by induction. Suppose M^o_{ijk} corresponds to an allocation of P^o and Q^o at N_i and N_i's children, respectively. Then, we show that there exists a $c_{i\ell}$ such that the allocation $P_{ij\ell} + Q_{ik\ell}$ is smaller than M^o_{ijk}. Since M_{ijk} corresponds to the minimum such sum, the result follows.

Let c^o denote the cost contributed by N_i corresponding to the allocation of P^o. Then the input size at the children of N_i, denoted by n^o, is $\frac{c^o}{h_i}$. Let $c_{i\ell}$ be such that $\frac{c_{i\ell}}{\beta} < c^o \le c_{i\ell}$. Since $P_{ij\ell}$ corresponds to a cost of $c_{i\ell}$, $P_{ij\ell} \le P^o$. This also proves the base case, in which N_i has no children that need to be allocated.

Now, the allocation of Q^o corresponds to a cost of $C_{ik} - c^o$, which is at most $C_{ik} - \frac{c_{i\ell}}{\beta}$. In other words, the sum of the costs contributed by T_x (for all N_x, children of N_i) is at most $C_{ik} - \frac{c_{i\ell}}{\beta}$. Let c^o_x denote the cost of T_x. Then, we have

$$\sum_x c^o_x \le C_{ik} - \frac{c_{i\ell}}{\beta}.$$

Note that, for every x, there exists a y_x such that $C_{xy_x} \le c^o_x < \beta C_{xy_x}$. Now, $\sum_x \beta^2 C_{xy_x} \le \beta^2(\sum_x c^o_x)$, which is at most $\beta^2 C_{ik} - \beta c_{i\ell}$. Thus, the $\beta^2 C_{xy_x}$'s form a valid split of costs for $Q_{ik\ell}$. Also, we have $n_{x\ell} = g_{p_x}\beta^\ell = g_i \beta^\ell = \frac{c_{i\ell}}{h_i}$. Since $c_{i\ell} < \beta c^o$, we have $n_{x\ell} < \beta n^o$. Hence, by Lemma 9.2, the memory allocation by Algorithm 2 in T_x for $\beta^2 C_{xy_x}$ with input n_{xy} is at most the memory allocation for βC_{xy_x} with input size of n^o. By induction, this is at most the optimal memory allocation for a cost of βC_{xy_x} and hence, at most the optimal memory allocation for c^o_x (which is $< \beta C_{xy_x}$).

Thus, it is easy to see that choosing the $\beta^2 C_{xy_x}$'s as the cost split for $Q_{ik\ell}$ makes $Q_{ik\ell}$ smaller than Q^o. Since $Q_{ik\ell}$ is the minimum allocation considering all possible cost splits, we have $Q_{ik\ell} \le Q^o$. Thus, the required result follows. ∎

The following lemma gives an upper bound on the actual cost of an allocation done by Algorithm 2.

Lemma 9.4: The cost of the allocation M_{ijk} chosen by Algorithm 2 is at most $\beta^{2d} C_{ik}$, where d is the height of N_i.

Proof: We again use induction to show this. Let P_{ijl} and Q_{ikl} be the allocations given by Algorithm 2 for N_i and its children. And let $M_{x\ell y_x}$ be the allocation chosen for T_x, where N_x is a child of N_i. By induction, the cost of $M_{x\ell y_x}$ is at most $\beta^{2(d-1)} C_{xy_x}$. This along with the fact that $\sum_x C_{xy_x} \le \beta^2 C_{ik} - \beta c_{i\ell}$ implies that the cost of $Q_{ik\ell}$ is at most $\beta^{2d} C_{ik} - \beta^{2d-1} c_{i\ell}$. Now, by the definition of $P_{ij\ell}$, the cost of $P_{ij\ell}$ is at most $c_{i\ell}$. Hence, the cost of M_{ijk} is at most $c_{i\ell} + \beta^{2d} C_{ik} - \beta^{2d-1} c_{i\ell}$, which is at most $\beta^{2d} C_{ik}$. ∎

Theorem 2 follows directly from Lemmas 9.3 and 9.4. The extra β in the approximation factor comes from the fact that the optimal cost may not equal $\beta^i C_{min}$ for any i. Hence, Algorithm 2 can end up choosing the entry in M corresponding to a C_{0k} where C_{0k} is β times the optimal cost.

Proof of Theorem 3

Proof: (Lemma 7.1) If a tuple is pushed down, then it implies that the newly arrived tuple is not one of the b_i tuples for which aggregation is being done at that node. Under Uniform distribution, the probability of this event is simply $\frac{g_i - b_i}{g_i}$, which implies the required result. ∎

Proof: (Theorem 3) Suppose there exists an optimal memory allocation O for which the stated property does not hold. Then there exist two nodes N_1 and N_2 such that

$$0 < m_1 < g_1 s_1 \text{ and } 0 < m_2 < g_2 s_2. \tag{6}$$

We now show that such an assignment can be converted into another optimal solution in which at least one of m_1 and m_2 is an extreme value. It is easy to show the total cost will be of this form:

$$S_0 = c_0 - c_1 m_1 - c_2 m_2 + c_3 m_1 m_2,$$

where the c_i's are some non-negative constants. Intuitively, the second and the third term arise from the cost incurred at the subtrees of N_1 and N_2, respectively; while the last term arises when one of them lies in the subtree of the other.

Let S_1 denote the solution cost in which the allocations for N_1 and N_2 are $m_1 + x$ and $m_2 - x$, respectively for some $x > 0$. Similarly, let S_2 denote the solution cost in which the allocations for N_1 and N_2 are $m_1 - x$ and $m_2 + x$, respectively. Note that such an x exists because of our assumptions (refer to (6)). Then,

$$S_1 = c_0 - c_1(m_1 + x) - c_2(m_2 - x) + c_3(m_1 + x)(m_2 - x)$$
$$S_2 = c_0 - c_1(m_1 - x) - c_2(m_2 + x) + c_3(m_1 - x)(m_2 + x)$$

Note that the average $\frac{S_1 + S_2}{2}$ is simply $S_0 - c_3 x^2$. Clearly, if c_3 is non-zero, then the above relation implies that the average of S_1 and S_2 is smaller than S_0. This, in turn, implies that one of S_1 and S_2 is smaller than S_0 contradicting S_0's optimality.

If c_3 is zero, then the average of S_1 and S_2 equals S_0. In this case, either both S_1 and S_2 equal S_0 or one of them is smaller than S_0. The latter again leads to a contradiction. Since $x > 0$, the former is possible only if $c_1 = c_2$; in this case, we can easily choose an x such that (6) does not hold, without any increase in the optimal cost. Thus, we can always obtain an optimal solution in which (6) does not hold. This concludes the proof. ∎