Continuous Sampling for Online Aggregation Over Multiple Queries [*]

Sai Wu, Beng Chin Ooi, Kian-Lee Tan
School of Computing, National University of Singapore, Singapore, 117417
{wusai, ooibc, tankl}@comp.nus.edu.sg

ABSTRACT

In this paper, we propose an online aggregation system called COSMOS (Continuous Sampling for Multiple queries in an Online aggregation System), to process multiple aggregate queries efficiently. In COSMOS, a dataset is first scrambled so that sequentially scanning the dataset gives rise to a stream of random samples for all queries. Moreover, COSMOS organizes queries into a dissemination graph to exploit the dependencies across queries. In this way, aggregates of queries closer to the root (source of data flow) can potentially be used to compute the aggregates of descendent/dependent queries. COSMOS applies some statistical approach to combine answers from ancestor nodes to generate the online aggregates for a node. COSMOS also offers a partitioning strategy to further salvage intermediate answers. We have implemented COSMOS and conducted an extensive experimental study in PostgreSQL. Our results on the TPC-H benchmark show the efficiency and effectiveness of COSMOS.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

General Terms

Algorithms, Design

Keywords

Online aggregation, sampling, random, dissemination graph

1. INTRODUCTION

Queries involving aggregates are widely used in many applications. For example, in Online Analytical Processing (OLAP), large amount of data stored in a data warehouse are typically aggregated and summarized to support business analysis and decision making. Unfortunately, aggregate queries are computationally expensive to process (which leads to long processing time) and have traditionally been evaluated based on a *blocking* execution model (which leads to long initial response time).

To handle the challenges, Hellerstein et. al. [8] proposed (non-blocking) *online aggregation* mechanisms that return approximate answers to users (almost) instantly. Instead of giving users a precise answer of an aggregate, online aggregation methods continuously generate running aggregates and their corresponding confidence intervals for the users. The idea is to continuously draw samples from a dataset, compute an approximation of an aggregate quickly with the samples, and refine the approximate aggregate as more samples are picked. Error bounds and confidence intervals are also computed and updated at the same time to give the user an indication of the quality of the approximate answers. Since samples are much smaller than the size of the dataset, the computation cost to return the first approximate answer is short. Moreover, as aggregate queries are mainly used to get a rough picture of a large dataset, it is expected that users will terminate the evaluation of their queries prematurely as long as they are satisfied with the quality of the answers. Thus, the (effective) processing time of the query may be lower than it would take to compute the precise aggregate value.

In this paper, we study how to efficiently deploy online aggregation for processing multiple aggregate queries. Multiple "aggregate" queries come in several flavors. A single complex query may involve multiple aggregates. For example, a nested query that involves aggregates in both the outer and inner query blocks. As another example, consider the following query (that finds the average supplier-quantity supplied by suppliers of a particular part) which is not uncommon in data warehouse applications (the lineitem schema is taken from the TPC-H benchmark):

```
SELECT AVG(quantity)
FROM (SELECT supp, part, SUM(quantity) as quantity
        FROM lineitem
        WHERE part = 10
        GROUP BY supp, part);
```

It is also possible for a single user to be switching between multiple aggregate queries. This is common in OLAP applications where a user may roll up and drill down between nodes of his/her data cube operation. Finally, aggregate queries issued by different users may share the same base tables. We note that there are potential dependencies among

[*]This work is funded by Singapore NRF grant R-252-000-376- 279

(sub-)queries, i.e., we can potentially derive the online aggregate of a (sub-)query from the online aggregates of other (sub-)queries. Moreover, we can potentially reuse samples retrieved to evaluate a number of queries (in contrast to each query retrieving its own samples).

We propose an online aggregation system called COSMOS (Continuous Sampling for Multiple queries in an Online aggregation System), to process multiple aggregate queries efficiently. COSMOS has a number of distinguishing features. First, it randomly *scrambles* a dataset so that tuples retrieved from anywhere can be considered as a random sample for all queries. Second, it continuously scans the scrambled dataset sequentially to generate a stream of samples. Third, to exploit the dependencies across queries, COSMOS organizes queries into a dissemination graph so that queries closer to the root feed their descendent/dependent queries with their estimated answers (as they are refined). Such a structure also minimizes the computational overhead. Fourth, COSMOS offers a partitioning strategy to further salvage intermediate answers. Finally, COSMOS applies some statistical approach to combine answers from ancestor nodes to generate the online aggregates for a node.

Several other works have also adopted the idea of continuously scanning a dataset. For example, in the DataCycle project, the entire database is broadcast cyclically over high-bandwidth communication networks to a large number of data filters that perform complex associative search operations in parallel [3]. More recently, George et. al. proposed the CJOIN pipeline that continuously pipes tuples from a fact table to a sequence of filters of a large number of queries [4]. Our COSMOS shares similar motivation; however, COSMOS cycles through a scrambled dataset to produce a stream of random samples. COSMOS is different from previous schemes as it is designed for approximate query processing. Each node in our dissemination graph is a query (compared to a dimension filter in CJOIN) and nodes share aggregate results (not raw tuples).

We have implemented COSMOS, and evaluated it against PostgreSQL using the TPC-H benchmark. Our extensive performance study shows that COSMOS can converge to good approximation answers quickly. Moreover, it is shown to be more efficient than naive approaches.

The rest of this paper is organized as follows. In the next section, we give an overview of COSMOS. In Section 3, we present our approach of organizing the queries as a dissemination graph. Section 4 presents the statistical estimators used in our online aggregation methods. In Section 5, we report results of an experimental study. Section 6 reviews some related works. Finally, we conclude this paper in Section 7.

2. COSMOS: THE BIG PICTURE

Figure 1 shows the system architecture of COSMOS. COSMOS comprises two major components: the *scrambler* and the *dissemination graph*.

2.1 The Scrambler

Given a dataset, the task of the scrambler is to generate a stream of random samples from the dataset. This has traditionally been performed at runtime by picking samples randomly from the dataset. However, we advocate storing the sequence of random samples as a *scrambled* dataset. In this way, by scanning the scrambled dataset, we can produce a stream of random samples. This design is motivated by a number of simple and yet interesting observations (which also correspond to the strengths of such an approach). First, we obtain only *one* random sample from a random page access of the original dataset, while we have *one page* of random samples from a page of the scrambled dataset. Second, we have effectively eliminated random look-ups, and can now sequentially scan the scrambled dataset for samples. This means the I/O cost for sampling is significantly reduced. In fact, the "gain by doing all data accesses in sequential order can be enormous" as "completely random disk access can be five orders of magnitude slower than sequential access" [9]. The results reported in [9] showed that the throughput of random access verses sequential access are 316 values/sec and 53.2M values/sec respectively (this study was done on a Windows 2003 server with 64 GB RAM and eight 15,000 RPM SAS disks in RAID5 configuration, and each record is 16 bytes). Our own preliminary study indicates that sequentially scanning a 100 GB dataset is only about 1,400 sec, which corresponds to the time to randomly sample 1.5% of the dataset. Third, by reading k pages of samples, we essentially have a larger number of samples (than k samples had k random pages been accessed as in conventional approaches). This will also translate to a lower error bound and a higher confidence (our experimental study in Section 5 confirms this). Putting it in another way, it means that we can arrive at the same accuracy as the conventional method much faster with fewer number of page accesses, or with better accuracy with k sequential page accesses. Fourth, we can continuously cycle through the scrambled dataset - when the last page is read, processing continues with the first page, and so on. This means that we can start sampling for a query at any arbitrary page at the time when the query is admitted for processing. We refer to the first page read for each query as its *anchor* page. A query completes with the precise answer, if necessary, when the same anchor page is read again. Moreover, a query that arrives at a later time can potentially reuse sample *results* (i.e., not the data samples themselves) from earlier queries (which are still running), and hence can "back-date" its anchor page to an even earlier anchor page, further improving the accuracy of the estimated aggregates within a shorter time. In addition, such an approach is easy to manage and incurs low overhead.

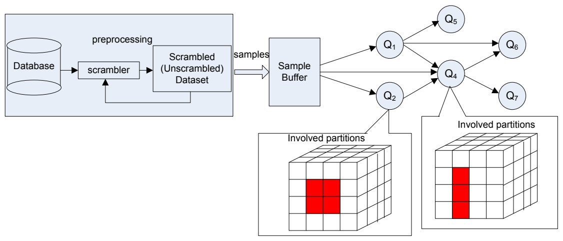


Figure 1: System Architecture of COSMOS

To scramble a dataset, we adopt a simple strategy: we scan a file sequentially; as we scan each tuple, we place it in a randomly picked position in a scrambled dataset (which is initially empty). The scrambled dataset is then further scrambled in the same manner. This process is repeated a number of times. Eventually, the scrambled dataset is used as the source dataset for online aggregation.

For queries that involve multiple relations, it may be necessary to generate a random sample stream for the intermediate results. Basically, two approaches can be applied. We can precompute the join result and scramble the result. Or alternatively, we build indexes on the join attributes of queries to exploit index nested loops join. To generate a random sample stream from a two-way join, we pick one relation as the outer relation (R) and the other relation as the inner relation (S). We retrieve samples from R as in the single relational case, i.e., by sequentially scanning the scrambled dataset. For each sample record from R, we search the index of S. The join output forms a random sample streams for the join of the two relations. The first scheme trades storage for efficiency and accuracy, while the second scheme sacrifices efficiency and accuracy for flexibility. In data warehouse systems, most queries are primary-key/foreign-key joins (e.g. TPC-H schema). We can apply the first scheme to scramble data. On the contrary, if queries are issued on the fly. The second scheme is more appropriate.

In our system, we adopt an offline batch-update strategy. The updates to the data warehouse system are collected and performed in a batch mode. During this period, no queries will be processed. If a large number of updates need to be performed, we will rebuild the scrambled dataset after all updates are committed. Otherwise, for each newly inserted tuple t, we randomly select a tuple t' from scrambled dataset and replace t' with t. t' is appended to the end of the scrambled dataset.

2.2 Organization of Queries into a Dissemination Graph

The second component is the *dissemination graph*, which is a directed acyclic graph (DAG), where nodes represent queries, and a directed edge between two nodes represent the dependency between the two queries. Such dependency allows the results of a parent node to be used by its child node. The data source is the root of the graph. As shown in Figure 1, data pages read from the scrambled dataset are first held in a *sample buffer*. Records in the buffer are then disseminated to queries, and discarded so that the buffer space is freed for other pages to be read.

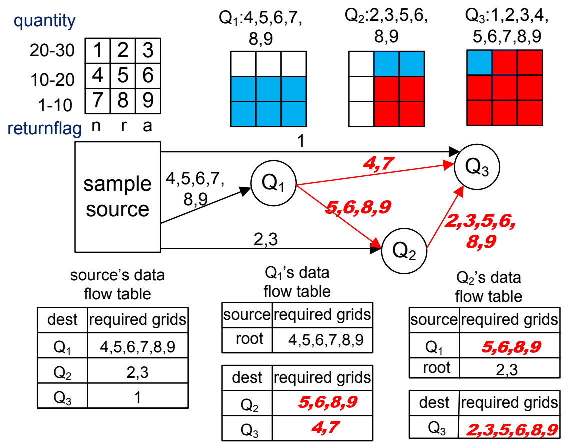
In one extreme, we can have a fat (two-tier) dissemination tree where all queries are dependent on just the sample stream source only. This corresponds to the case where each query is processed independently However, each sample read is reused by all queries. A general dissemination graph offers more opportunities for sharing intermediate answers (not data). For example, consider the following three queries (using TPC-H schema as the example):

Q_1 select avg(discount) from lineitem where quantity<20 group by returnflag;

Q_2 select avg(discount) from lineitem where returnflag='r' or returnflag='a';

Q_3 select avg(discount) from lineitem;

Suppose these queries arrive in the order Q_1, followed by Q_2, and then Q_3. Now, although Q_1 and Q_2 share a subset of the data, it is not possible to salvage the answer of Q_1 for Q_2 if we naively maintain only one single running average and error bound for Q_1. Fortunately, we observe the following: if we had partitioned the answers of Q_1 based on $returnflag$ (e.g. we generate running averages and error bounds for *discount* with different $returnflag$s), then we can reuse the



source's data flow table

dest	required grids
Q_1	4,5,6,7,8,9
Q_2	2,3
Q_3	1

Q_1's data flow table

source	required grids
root	4,5,6,7,8,9

dest	required grids
Q_2	5,6,8,9
Q_3	4,7

Q_2's data flow table

source	required grids
Q_1	5,6,8,9
root	2,3

dest	required grids
Q_3	2,3,5,6,8,9

Figure 2: Salvaging Results of Queries

output of two partitions ($returnflag$='r' and 'a') for Q_2 (Q_1 only needs to examine tuples whose quantity $\geq$ 20 for $returnflag$='r' or 'a')! Likewise, it is clear that we can (possibly) reuse the output of Q_2 for Q_3. The problem, however, becomes more challenging for Q_3 since it can reuse the partitions of both Q_1 and Q_2. It is difficult to combine Q_1 and Q_2's partitions to generate a result for Q_3. Thus, we need a more aggressive strategy to be able to salvage intermediate results.

Figure 2 shows the example of reusing partitions among the queries. Suppose the table is partitioned by *quantity* and $returnflag$ into 9 grids. Q_2 can reuse the results of grids 5, 6, 8 and 9 from Q_1, while Q_3 can reuse the results of all the grids except the first one. It is interesting to note that the order of query arrivals can affect reuse. For example, had Q_3 arrived first, then it could have been partitioned in a manner that benefit both Q_1 and Q_2.

Now, we can identify four key challenges in the design of the dissemination graph. First, we need to determine how samples and answers should be partitioned. Our idea is to partition samples and answers into grids (as shown in Figure 2), and to salvage grids that are contained with the scope of a query. Second, we need a way to measure the relationships (or dependencies) between queries. Based on the relationships, we can build the DAG to reuse the sample answers. Third, we need to dynamically maintain the DAG as queries are submitted/admitted and completed. Finally, each query's answers are derived from possibly multiple answers of its parent nodes (including the sample stream source). We need statistical estimators to merge these partial answers into a query answer. We shall discuss our solutions to the first three issues in Section 3, and the last issue in Section 4.

3. THE DISSEMINATION GRAPH

Given a set of queries, we organize them into a data-flow graph. Each node in the graph corresponds to a query, and the nodes are connected via data streams. Samples and partial results are shared via the data streams between nodes. When a new query joins the system, it selects some nodes as its data source and notifies these nodes about its required samples/answers. The corresponding nodes will generate a data stream for the new node. Samples or partial results are then streamed to the new node which then combined these into its own final results (and possibly maintaining some partial results for reuse). In this section, we discuss how to build and maintain this dissemination graph.

3.1 Query-Based Partitioning

When a new query joins the system, we need to evaluate its relationship with existing queries. In particular, if two queries need to process a common subset of tuples, we can share the samples or partial results of this subset between the queries, e.g, queries Q_1 and Q_2 in our discussion in Section 2. However, as noted, Q_2 can potentially exploit the output of Q_1 only if Q_1 had maintained partitions of its answers. Moreover, even if Q_1 had actually partitioned in a manner that Q_2 can reuse the answers, it is possible that another query Q_4 (e.g., select avg(discount) from lineitem groupby returnflag, quantity;) may come along that cannot take advantage of the partitioning of Q_1 (which is beneficial for Q_3). To deal with this problem, we adopt, as a first cut, a straightforward solution - to partition the data space statically into hypergrids (grids in short) so that only girds that are fully contained in an answer can be salvaged.

To partition the data space into grids, we need to identify candidate attributes for partitioning. We identify two types of columns as candidates for partitioning. The first type of columns (type-1) are those appearing in "Group By" clauses. By partitioning such columns, we reuse the samples for answering "Group By" queries. The second type of columns (type-2) are columns appearing in "Where" predicates. We can reuse the samples to answer queries with different selection predicates after partitioning type-2 columns.

For a type-1 column, we partition the column based on the group names if the number of groups is fewer than N_g (a predetermined system parameter). Otherwise, we partition the column as a type-2 column. For a type-2 column, we partition the column based on the values. In the following discussion, we focus on how to adaptively partition type-2 column. One straightforward method is to partition the space uniformly (e.g. for each column of a table, we partition it into k equal-size cells). Unfortunately, this method suffers from the curse of dimensionality - a table with 10 partitioning columns where each column is split into 10 ranges will result in 10^{10} grids. The number of grids may be even larger than the number of tuples. Maintaining partial results for all grids is not practical.

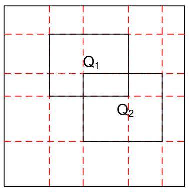


Figure 3: Query-based Partitioning

An alternative approach, which we refer to as Partition, is to partition the space based on the query workload. A query training set can be collected as representative of the workload. The idea of this scheme is to partition the space for each query such that grids are fully contained in some queries and we can easily share the partial results between queries. Figure 3 illustrates the idea of this approach. We partition the space based on two queries Q_1 and Q_2. A 5×5 partition is generated and the search ranges of a query composes 4 grids. Q_1 and Q_2 share a common grid. This approach works well if the combination of search conditions is limited. However, in most cases, this approach will also generate too many grids. To reduce the maintenance overheads, we propose an adaptive query-based partitioning scheme, QueryPartition, to merge some partitions.

Algorithm 1 `QueryPartition(QuerySet` S_q`, int` $pNum$`)`

```
 1: $P = Partition(S_q)$
 2: if $|P| \leq pNum$ then
 3:   return $P$
 4: else
 5:   while $|P| > pNum$ do
 6:     $S_a = getAttributes()$
 7:     for $\forall a_i \in S_a$ do
 8:       $R_i = getRange(a_i, P)$
 9:       for $\forall r_j \in R_i$ do
10:         $P'_i = combine(r_j, r_{j-1})$
11:         $x = computeGridUtility(P'_i)$
12:         if $x > max$ then
13:           $max = x, candidate = P'_i$
14:     $P = P'_i$
```

Algorithm 1 presents the algorithmic description of our adaptive scheme. S_q is the training set and $pNum$ is a system set maximal number of partitions. We first use scheme Partition to partition the space. If the number of generated partitions is fewer than $pNum$, we can terminate the process (lines 1-3). Otherwise, we proceed to merge partitions iteratively (lines 5-14). In each iteration, we combine a set of partitions. Specifically, for each partitioned attribute, suppose it has been split into k sub-ranges $r_1, r_2, ..., r_k$. We try to combine the consecutive ranges (line 10), After the corresponding partitions are merged, we evaluate the utility of the current partitioning. Among all possible mergings, we select the combination that maximizes the utility. In this work, we adopt a simple metric for utility - we simply count the number of grids that can be reused. The number of grids is reduced in each iteration and we can terminate the processing when the number of grids is fewer than the threshold.

3.2 Dissemination Graph

The purpose of parititioning the space is two fold. First, it allows us to easily determine how a query can be partitioned to potentially benefit other queries. In Figure 2, we have pre-partitioned the space into 9 grids (based on two attributes, quantity and returnflag). When Q_1 arrives, it organizes its (partial) answers into 6 partitions (based on the pre-partitioning). Second, it allows us to easily determine what can be shared. In Figure 2, when Q_2 joins, it organizes its (partial) answers into 6 partitions, and "notices" that it can reuse the partial answers from 4 partitions of Q_1.

The dissemination graph is a directed acyclic graph (DAG). Each node represents a query. We also use Q_i to denote node i and query Q_i interchangeably. The sample stream source, denoted Q_0, forms the root node of the DAG. An edge $Q_i \rightarrow Q_j$ indicates that there is a data stream from Q_i to Q_j. In particular, for an edge of the form $Q_0 \rightarrow Q_i$, the actual sample data are transmitted from the sample source (Q_0) to node Q_i. Q_i will then compute the aggregates for the partitions based on the samples collected. If a query involves only part of a grid (rather than a full grid), the summary data are also computed for them; however, such partial partitions cannot be salvaged and shared. The samples are then discarded. Note that it is possible that a query does not fit the pre-partitioning (e.g., a query that groups by discount). In this case, such a query also draws its sample data from Q_0 and computes its own answers. For such queries, our current solution does not offer any opportunity for salvaging their answers. More aggressive sharing methods are needed, which we plan to explore in future work.

On the other hand, for an edge of the form $Q_i \rightarrow Q_j$ ($i, j \neq 0$), no actual samples are transmitted; instead, it is the aggregates of those grids that are shared that are transmitted from Q_i to Q_j. Referring to Figure 2, the numbers on the edges denote the data (sample or partial results) of grids that should be transmitted - the normal font denotes actual samples, while the italized bold font denotes summary/aggregate data.

Let $uplink(Q_i)$ and $downlink(Q_i)$ denote the uplink nodes and downlink nodes of Q_i, respectively. Let S_q denote the current query set. We have

$$\forall Q_j \in S_q, \exists(Q_j \rightarrow Q_i) \Rightarrow Q_j \in uplink(Q_i)$$

$$\forall Q_j \in S_q, \exists(Q_i \rightarrow Q_j) \Rightarrow Q_j \in downlink(Q_i)$$

Q_i accepts the data streams from nodes in $uplink(Q_i)$ and it streams data to the nodes in $downlink(Q_i)$. Each query node registers its required grids in its uplink nodes, and records where it is getting these grids from. Referring to Figure 2, at Q_1, we see that Q_2 has registered that it needs the aggregate data for grids 5, 6, 8 and 9; and Q_3 has registered its interests in the summary data for grids 4 and 7. We also see that Q_2 keeps track of the source of its data (it is receiving from Q_1 partial results of grids 5, 6, 8 and 9, and sample data for grids 2 and 3 from the root). The node in the dissemination graph disseminates its samples or partial results according to the registration information of its downlink grids. The dissemination graph defines how the samples are disseminated among the queries.

We are now ready to discuss how the dissemination graph is constructed. The idea of constructing a dissemination graph is to facilitate the samples' sharing. As we share the samples in the granularity of grids, the relationship between two queries is also defined by the involved grids. Generally, if two queries involve the same grid, we can reuse the partial answers of the grid. The grids can be classified into three types for a specific query Q_i.

1. Grid g_x is fully contained by Q_i's search range.

2. Grid g_x overlaps with Q_i's search range and Q_i does not fully contain g_x.

3. Grid g_x does not overlap with Q_i.

For a type-1 grid, Q_i needs to retrieve samples from the sample source (Q_0) if there are no existing queries for it to salvage; otherwise, if partial result is available, it will reuse it. For a type-2 grid, Q_i needs to retrieve sample from Q_0 to compute its partial result on the fly, as partial result of such a grid cannot be used by Q_i (since they are obtained based on samples from the entire grid space). We use $f_1(Q_i)$ and $f_2(Q_i)$ to denote the type 1 and type 2 grid sets of Q_i, respectively. To simplify the presentation, we define function F as follows.

Definition 1. *Let S_g and S_q denote the set of grids and queries respectively. $F : S_q \times S_q \rightarrow S_g$. Let $S_r = F(Q_i, Q_j)$, S_r denotes the set of grids, which are fully contained by both Q_i and Q_j.*

F returns the grids that can be shared between the queries. It is obvious that $F(Q_i, Q_j) = F(Q_j, Q_i)$.

Let S_q denote the current query set in the system. When an incoming query Q_i joins the system, we search queries in S_q and connects Q_i to the queries, whose partial results can be reused. The intuition is to maximize the result's sharing. Algorithm 2 shows the idea of retrieving the corresponding queries. First, we rank queries in S_q with regard to Q_i by function F in a descending order (lines 2-5). If two queries Q_1 and Q_2 have the same F value (e.g. $F(Q_1, Q_i) = F(Q_2, Q_i)$), Q_1's score is higher than that of Q_2's, i.f.f. $|f_1(Q_1)| < |f_1(Q_2)|$. In other words, we prefer the queries that involve fewer type 1 grids. Then, we continuously retrieve the top query from the list until the search range of the incoming query has been fully covered or there is no candidate query left (lines 6-16). After inserting a query into the result set, we recompute the scores and (reusable) grids of the remaining queries (lines 13-15). We reduce the query's score by removing the effect of queries in the result set. We overload F to compute the set of grids between two grid sets.

Algorithm 2 `RankQuery(Query` Q_i`, QuerySet` S_q`)`

```
 1: $S_r = \emptyset$
 2: **for** $\forall Q_j \in S_q$ **do**
 3:     $grid[j] = F(Q_i, Q_j)$
 4:     $score[j] = |F(Q_i, Q_j)|$
 5: $List = sort(S_q, score)$
 6: **while** true **do**
 7:     $Q_k = List.removeFirst()$
 8:     **if** $score[k] < 0$ **then**
 9:         return $S_r$
10:     $S_r = S_r \cup \{Q_k\}$
11:     **if** $S_r$ fully covers $Q_i$ or $List$ is empty **then**
12:         return $S_r$
13:     **for** $\forall Q_j \in List$ **do**
14:         $score[j] = score[j] - |F(grid[j], F(grid[k], grid[i]))|$
15:         $grid[j] = grid[j] - F(grid[j], F(grid[k], grid[i]))$
16:     $List = sort(List, score)$
```

Algorithm 3 `Join(Query` Q_i`, QuerySet` S_q`)`

```
1: $S_r = RankQuery(Q_i, S_q)$
2: $G = f_1(Q_i) \cup f_2(Q_i)$
3: **for** $\forall Q_j \in S_r$ **do**
4:     add edge $Q_j \rightarrow Q_i$
5:     register $F(Q_i, Q_j) \cap G$ in $Q_j$ for $Q_i$
6:     $G = G - F(Q_i, Q_j)$
7: **if** $G \neq \emptyset$ **then**
8:     add edge $root \rightarrow Q_i$
```

Suppose Algorithm 2 returns S_r for the incoming query Q_i. Q_i joins the system as a descendent of queries in S_r. Specifically, for any query Q_j in S_r, we add an edge $Q_j \rightarrow Q_i$. Algorithm 3 illustrates the join process of a query node. In line 2, G represents the grids involved in Q_i. We iterate through all queries in S_r and add the corresponding edges (line 3-6). When adding an edge $Q_j \rightarrow Q_i$, Q_i registers the grids that are required to be retrieved from Q_j. Finally, if some grids cannot be found in the existing query set, Q_i directly draws the samples from the root node (sample source).

The data stream from the root to a query composes of samples retrieved from the database, whereas the data stream from one query to another composes of partial results of grids.

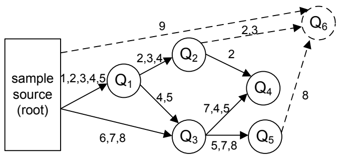


Figure 4: Q6 joins the dissemination graph

Figure 4 illustrates the process of adding a new query into the dissemination tree. In this example, we assume that the data space has been partitioned into 9 grids (labelled 1 to 9). The figure shows that Q_1 basically requests for sample tuples that fall into grids 1 to 5. At Q_1, the partial answers of grids 1 to 5 will be maintained. Q_3 requires samples from grids 4 to 8. However, it can reuse the summary of grids 4 and 5 from Q_1, but will need to retrieve sample tuples of grids 6-8 from the stream source. Now, suppose Q_6 joins the system and it fully covers grids 2,3 and 8, and partially overlaps with grid 9. Q_6 searches existing queries and try to reuse the results of grids 2, 3 and 8. Query Q_1, Q_2 and Q_4 can share their results of grid 2. Q_2 is chosen as it only maintains results for three grids and thus has lower overheads. Similarly, Q_2 and Q_5 are chosen to provide the results of grids 3 and 8. As Q_6 only partially overlaps with grid 9, it directly retrieves samples from the sample source. To improve performance, it inherits the buffered samples for this grid from the sample source.

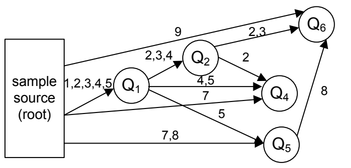


Figure 5: Q3 leaves the dissemination graph

When a query terminates, we need to remove it from the stream graph and reorganize the query nodes. Suppose query Q_i is finished. We need to remove the registered information from the queries in $uplink(Q_i)$, and reorganize the queries in $downlink(Q_i)$ since these queries depended on Q_i for partial answers. Algorithm 4 illustrates the idea of handling query departure. For each query Q_j in $downlink(Q_i)$, we rank the queries in $uplink(Q_i)$ for Q_j. Then, based on the results, we add new edges between the queries. Figure 5 shows an example of the reorganization of data streams between the queries. Here, Q_3 (in Figure 4) completes its processing and leaves the systems. Its downlink nodes Q_4 and Q_5 need to adjust their links. Based on Algorithm 4, the new data stream is shown in Figure 5, where Q_4 and Q_5 are now linked to the root and Q_1.

3.3 Online Aggregation Over The Dissemination Graph

In this section, we describe how COSMOS operates. In the dissemination graph, the root continously scans the scrambled dataset and retrieves samples to its buffer. Each node

Algorithm 4 `Leave(Query` Q_i`)`

```
1: remove edges from nodes in $uplink(Q_i)$ to $Q_i$
2: remove edges from $Q_i$ to nodes in $downlink(Q_i)$
3: for $\forall Q_j \in downlink(Q_i)$ do
4:     $G = f_1(Q_j) \cup f_2(Q_j)$
5:     $S_r = RankQuery(Q_j, uplink(Q_i))$
6:     for $\forall Q_k \in S_r$ do
7:         add an edge $E(Q_k \rightarrow Q_j)$
8:         register $F(Q_j, Q_k) \cap G$ in $Q_k$ for $Q_j$
```

in the graph receives two types of data streams (one of which may be empty). The data stream connecting the root to the query node provides the actual samples for grids. The data stream connecting two query nodes transfers aggregate values and statistics of type-1 grids (the grids that are fully contained by the query). To reduce the update overheads, the root only forwards the samples (as a batch) to other nodes when its buffer is full. As each node receives the batch of samples, it recomputes its aggregates and error bounds, and updates its downlink nodes. The incoming stream data will trigger an update event for a query node to employ Algorithm 5 to update its result and error bound.

Algorithm 5 `Update(StreamData` S`, Query` Q_i`)`

```
1: $Q_i$.updateGrid($S$)
2: GridSet $G$=$Q_i$.getGrid()
3: for $\forall g_i \in G$ do
4:     compute aggregate and variance
5: update error bound and confidence using variance
6: forward $S$ to $Q_i$'s downlink nodes
7: if $Q_i$.isComplete() || error bound and confidence satisfy
   the requirement then
8:     return result with error bound and confidence
```

In Algorithm 5, the query node first updates the results of its involved grids based on the new stream data (line 1). If the stream data are partial results of type-1 grids, the node just copies the new results.[1] If the stream data are new samples from the database, we need to recompute the aggregate results of the grids (type 1 and type 2) (lines 3-4). We shall defer the discussion on the formulas for computing the aggregates for type-1 and type-2 grids to Section 4. It suffices to note that at the end of this step, we have a set of aggregates and error bounds for each grid. Note that the actual aggregate and error bound of a query may require merging the partial results (line 5). Again, we shall defer our discussion on how such partial results are merged to Section 4.

After the query updates its result, it starts to disseminate the grid results to its downlink nodes (line 6). For a downlink node Q_i and its registered grid set G, $\forall g_i \in G$, if the results of g_i has been updated, we forward g_i's new results to node Q_i.

The query can be terminated in two cases (line 7). First, the query has scanned the whole dataset and generates the precise result. Initially, when a query receives a sample for a grid, it will mark the grid with the anchor page of the sample. If the grid sees the same anchor page again, it stops receiving new samples as it has scanned the whole data set.

[1]In our implementation, we did not duplicate multiple copies of partial results. Instead, we only store one copy of the summarized result, and provide a pointer to the appropriate grids.

If all grids involved in the query stops receiving samples, the query can generate the precise result. Moreover, if a query node receives the information of a grid from its uplink node, it will inherit the anchor page information from the uplink node. The second condition of query termination is determined by the user. If the user is satisfied with the result, he can stop the query processing before scanning the whole dataset.

In the above discussion, we show the maintenance strategy of the dissemination graph, which is designed to salvage samples of the grids that are fully contained by queries. If the grid partially overlaps with a query, we must draw its samples from the sample stream source. Let the grid set that overlaps with the current queries be S_g. To facilitate the query processing in S_g, we maintain a sample cache for the grids in S_g. Specifically, the sample source keeps k recent samples for each grid g_i in S_g. When a new sample is retrieved for g_i, it will be inserted into the cache and if the cache is full, the oldest sample is discarded.

Given a new query Q, suppose g_i partially overlaps with Q. After Q joins the dissemination graph, we check g_i's sample cache in the source node. For the samples that pass the predicates of Q, we use them to compute the initial result of g_i (To efficiently locate the samples that satisfy the predicates, we can apply an index structure, such as R-tree, to maintain the samples in cache). After initialization, g_i's result will be updated by the samples that are directly streamed from the source.

4. STATISTICAL ESTIMATION

In COSMOS, for each query/node in the dissemination graph, there are three kinds of estimates to evaluate: (a) estimates for type-1 grid - this is needed for those nodes that are connected directly to the sample source; for those that salvage the results, they simply reuse them; (b) estimates for type-2 grid - this is needed for a query that requires grids that partially overlap its query region and samples have to be drawn from the sample source; (c) estimates for the query itself - this requires integrating the above two kinds of estimates. In this section, we first present the statistical estimates used to compute estimates for type-1 and type-2 grids. Then, we look at how multiple estimates can be aggregated.

4.1 Estimators for Grids

In this paper, we use *avg*, *sum* and *count* as examples. Other aggregate operators can be handled in the same way. For each grid g_i, we also maintain a count, denoted as $H(g_i)$, of the number of tuples in g_i seen so far. $H(g_i)$ will be used in the result estimation.

For a table $R = \{c_0, c_1, ..., c_k\}$, we refer to c_i as an aggregate column if it can appear as an aggregate attribute in a query. For example, *returnflag* in *lineitem* is not an aggregate column (since there is no query that computes an aggregate on *returnflag*), while *discount* is an aggregate column. We use $\mathcal{C}$ to denote the set of aggregate columns of a table.

Generally speaking, we need to keep two types of partial results for a grid g_i, the aggregate results and the corresponding statistics. For a table $R = \{c_0, c_1, ..., c_k\}$, if $c_i \in \mathcal{C}$, we precompute all its possible aggregations. For example, suppose grid g_i has received N samples. Let the current aggregate values of c_i be $avg(c_i) = V_{avg}$, $sum(c_i) = V_{sum}$ and $count(c_i) = V_{count}$, respectively. Since we maintain $H(g_i)$, we have $V_{count} = H(g_i)$. For the other two aggregate values, after receiving a new tuple $t = \{v_0, v_1, ..., v_k\}$ of R, if t belongs to grid g_i, we update the aggregate values of c_i as:

$$avg(c_i) = \frac{V_{avg}N + v_i}{N+1} \tag{1}$$

$$sum(c_i) = \frac{V_{sum}N + H(g_i)v_i}{N+1} \tag{2}$$

In this way, we maintain the aggregate values of all columns in $\mathcal{C}$. And, they can be potentially reused by different queries.

Besides the aggregate results, we need the corresponding statistics to estimate the error bound and confidence. First, we need to record how many samples have been received for this grid (e.g. N). Then, we need to compute the variances for the aggregate values. To simplify the computation of variance computation, we apply the computational formula of variance.

$$var(X) = E(X^2) - (E(X))^2 \tag{3}$$

We define E^2 for $avg(c_i)$ and $sum(c_i)$ as:

$$E^2_{avg} = E^2_{avg} + v_i^2; \quad E^2_{sum} = E^2_{sum} + v_i^2 H(g_i)^2 \tag{4}$$

And the variances of $avg(c_i)$ and $sum(c_i)$ are estimated as:

$$var(avg(c_i)) = \frac{E^2_{avg}}{N} - V^2_{avg} \tag{5}$$

$$var(sum(c_i)) = \frac{E^2_{sum}}{N} - V^2_{sum} \tag{6}$$

For type-2 grids, the estimates can be computed in a similar manner. However, for type-2 grids, besides the number of retrieved samples (N), we also keep the statistics of the number of samples in the query range (N'). As type-2 grid partially overlaps with the query, $N' < N$. Thus, we need to replace N with N' for Equations 1, 5 and 6. Besides, new formulas are required to estimate $count(i)$.

$$count(c_i) = \frac{N' H(g_i)}{N} \tag{7}$$

$$E^2_{count} = E^2_{count} + f(v_i)^2 N^2 \tag{8}$$

$$var(count(c_i)) = \frac{E^2_{count}}{N} - \frac{N'^2 H(g_i)^2}{N^2} \tag{9}$$

In Equation 8, if the sample is in the query range, $f(v_i)$ returns 1. Otherwise, it returns 0.

4.2 Integrating Estimators

At a query node, it basically has two types of estimates - estimates for type-1 grids that it can obtain from its parent node(s) (or compute itself if the parent is the sample source), and estimates for type-2 grids (computed from actual samples that it retrieves from the sample stream source). To produce the final estimates for this node, we need to merge these estimates into a single estimate and error bound. We assume that each grid can be considered as an independent sample set for the database. And we apply weighted sampling approach to combine the results. Suppose the partial result is X_i for grid g_i, the weighted result is computed as:

$$X = \sum_{i=0}^{k} w_i X_i + \sum_{i=0}^{k}\sum_{j=0}^{k} cov(X_i, X_j) \tag{10}$$

w_i is the weight assigned to g_i and $\sum w_i = 1$. $cov(X_i, X_j)$ is the covariance between two estimations. As no grid shares the same sample, the covariance between different estimations is very small. We discard the covariance part in Equation 10 to simplify the computation. Similar approach is adopted in [10], as well. Suppose the variance of the estimation of g_i is σ_i^2, we estimate the variance of the final result as:

$$\bar{\sigma}^2 = \sum w_i^2 \sigma_i^2 \tag{11}$$

The optimal weight that minimizes the variance is computed as [10]:

$$w_i = \frac{1}{\sigma_i^2 \sum \frac{1}{\sigma_i^2}} \tag{12}$$

We note that when combining estimations from multiple grids, sometimes, we may under-estimate the variance as some grids lack enough samples. As $\sum w_i = 1$, $\sum w_i^2 \leq 1$. Some small σ generated by a grid with few samples may be assigned a large weight, which results in a biased estimation. This is more significant when a large number of grids are involved in the query. Therefore, in this paper, we use a modifier to handle the under-estimation.

As a matter of fact, variances computed by Equations 5, 6 and 9 are approximations. We use the sample variances as the variances of the whole dataset. From the analysis of [15], a better estimation of variance σ is

$$\sigma^2 \sim \frac{\sigma_s^2 \chi^2}{K-1} \tag{13}$$

where σ_s^2 is the variance estimated from the samples, χ^2 follows chi-square distribution and K is the number of grids (e.g. number of independent samples).

By extending χ, we get the final estimation of σ^2 as:

$$\sigma^2 = \bar{\sigma}^2 \times \frac{1}{K-1} \sum_{i=1}^{K} \frac{(X_i - X)^2}{\sigma_i^2} \tag{14}$$

The variance is used to generate the statistics error bound and confidence for the approximate result (line 7 in Algorithm 5). Suppose the accurate result is $\bar{X}$. Based on central limit theory, $\bar{X} - X$ follows normal distribution, if X is generated by a large enough sample set. Therefore, we have

$$P(\bar{X} - X < \epsilon) \simeq 2\phi(\frac{\epsilon\sqrt{\bar{N}}}{\bar{X}}) - 1 \tag{15}$$

where $\bar{N}$ is the total number of samples retrieved for the query, $P(\bar{X} - X < \epsilon)$ is the confidence that the error bound is ϵ.

To guarantee the validity of Equation 15, we use the result of a grid, only if the number of its samples is larger than a predefined threshold $\mathcal{T}$. In our experiment, we set $\mathcal{T}$ to 50. Another exception is that if the aggregate column is also used in partitioning, we need to collect results from enough grids to avoid biased estimation. Specifically, if we partition c_i into k equal-size ranges $[l_1, u_1]$, $[l_2, u_2]$,...,$[l_k, u_k]$ and a query asks for the average value of c_i, we cannot provide an unbiased result until we have collected the samples for all sub-ranges. Suppose based on tuple count information in H, the query range covers about n_0 tuples and range $[l_i, u_i]$ have n_i tuples approximately. If $[l_i, u_i]$ overlaps with the query and $\frac{n_i}{n_0} \geq \alpha$, the query node will generate an error bound only if it has obtained samples from a grid that overlaps with $[l_i, u_i]$. Otherwise, the error bound is set to "unknown". We set *alpha* to 0.1% in the experiments. And such setting does not affect the performance significantly. As if we draw samples uniformly from the dataset, we can soon get samples for all grids.

5. EXPERIMENTAL STUDY

5.1 Experiment Settings

Our COSMOS system is implemented in Java and deployed on a DELL server with Quad-Core AMD Opteron(tm) Processor 8356 and 128GB memory. The system keeps a pool to accept incoming queries. If the pool is not full, the query is inserted into the pool and joins the stream graph. Otherwise, the query must wait until some existing queries finish their processing. The system continuously retrieves samples from the scrambled dataset. Each time, S samples are extracted. S equals to the buffer size of the root node in the stream graph. The incoming samples will trigger Algorithm 5 at query nodes sequentially. And the queries will update their results and the corresponding statistics. We use TPC-H 10G dataset as our test data and generate queries based on the following two query templates.

Query Template 1 (T1):

SELECT sum(c_i)|count(c_i)|avg(c_i)
FROM LINEITEM
WHERE [orderdate> x and orderdate< x+2 year]|[discount > x and discount< x+0.03]| [extendedprice> x and extendedprice< x+30000]|[quantity> x and quantity< x+30]
GROUP BY [returnflag]|[linestatus];

where $c_i = quantity|discount|extendedprice$ and x is some random values.

Query Template 2 (T2):

SELECT sum(c_i)|count(c_i)|avg(c_i)
FROM LINEITEM L, ORDERS O
WHERE L.orderkey = O.orderkey and [shipdate> x and shipdate< x+2 year]|[discount> x and discount< x+0.03]| [extendedprice> x and extendedprice< x+30000]|[quantity> x and quantity < x+30]|[totalprice> x and totalprice< x+300000]
GROUP BY [returnflag]|[linestatus]|[orderpriority];

where $c_i = quantity|discount|extendedprice|totalprice$ and x is some random values.

Two stream sources are established to process queries generated from different query templates. In the preprocessing phase, we generated 1000 random queries for each template and partitioned the search space based on the queries. In this way, search spaces of template 1 and template 2 are partitioned based on 6 and 8 columns, respectively. Based on TPC-H schema, the selectivity of each aggregation group varies from 0.2 to 0.01 (depending on queries).

We use the average processing time of the query as the metric. In each test, we process 1000 queries and compute the average time. Each experiment is repeated 10 times to remove any side effects. The number of concurrent queries is

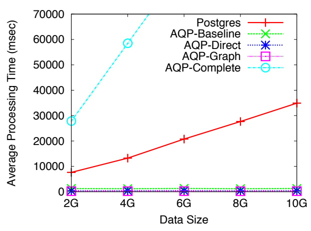


Figure 6: Effect of Data Size (T1)

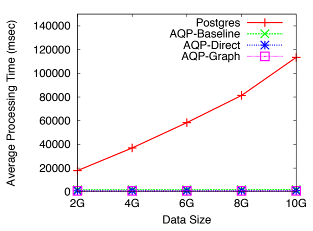


Figure 7: Effect of Data Size (T2)

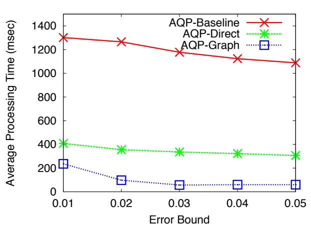


Figure 8: Effect of Error Bound (T1)

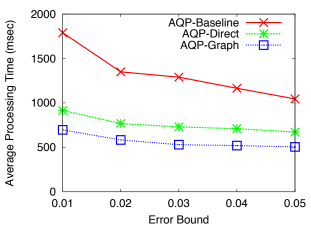


Figure 9: Effect of Error Bound (T2)

set to 20. And each time, 200 samples are retrieved from the database to update the results. We set a predefined error bound and confidence to simulate the user's behavior. Let ϵ and c denote the predefined error bound and confidence, respectively. Suppose the approximate result is $\bar{V}$ and the accurate result is V, then $P(\frac{|\bar{V}-V|}{V}) < \epsilon = c$. The default values of ϵ and c are 0.01 and 95%, respectively. When a group achieves the predefined error bound and confidence, we stop updating its result. And, the query is said to be complete only if all of its groups have stopped processing. We use the query-based grid partitioning scheme as the default partitioning method and the space is initially partitioned into 1000 grids.

For comparison purposes, we implemented 3 methods. In $AQP - Baseline$, the pool size is set to 1. In other words, we process queries one by one and do not share samples in $AQP - Baseline$. In $AQP - Direct$, all queries are directly connected to the root node. While samples are retrieved once and shared among the queries, there is no sharing of partial answers among them. In $AQP - Graph$, the queries are maintained in a dissemination graph and partial answers and samples are shared via the sub-streams.

5.2 Experiment Evaluation

5.2.1 Effect of Data Size

In the first set of experiments, we vary the data size from 2G to 10G and evaluate the performance of different schemes. We also run the same set of queries in PostgreSQL to get the precise results. When processing queries of template 2, we actually build a scrambling dataset for the results of $lineitem \bowtie order$. Figure 6 and Figure 7 show the average processing time for template 1 and template 2, respectively. Both figures indicate that online aggregation is scalable with regard to the data size. And the cost of online aggregation is much lower than the complete query processing (PostgreSQL). Among the online aggregation schemes, $AQP - Graph$ performs the best and $AQP - baseline$ is the worst (we will be able to see the difference more clearly in subsequent experiments). This is expected as $AQP - Graph$ salvages both samples and partial answers. Note that the predefined error bound and confidence are 0.01 and 95%, which actually provide a good enough estimation. The scalability of online aggregation can be explained as follows. The processing time of online aggregation is affected by the number of samples retrieved. And the number of required samples are estimated via the variance. Therefore, the performance of online aggregation is actually determined by the data distribution, rather than the data size. For comparison purpose, $AQP - Graph$ is also run to return precise results. We use $AQP - Complete$ to denote such a scheme. In Figure 6, we can see $AQP - Complete$ performs worse than PostgreSQL, because it needs to update the results and statistics of grids. On the contrary, PostgreSQL applies cache and index (index is built for orderdate and extendedprice) to improve its performance. The diagram of $AQP - Complete$ is omitted in Figure 7, as it is much worse than other schemes. This experiment shows that AQP-based schemes are most useful when precise results are not necessary.

5.2.2 Effect of Error Bound and Confidence

The predefined error bound affects the number of retrieved samples. If we try to get a tighter bound, we need much more samples from the database. Figure 8 and Figure 9 confirm this. The predefined error bound ranges from 0.01 to 0.05. $AQP - Graph$ performs best as it shares both the samples and partial results among the queries. In particular, recall that whenever partial results can be salvaged, $AQP - Graph$ essentially has a larger "sample size" than $AQP - Baseline$ and $AQP - Direct$. This is because when queries join the dissemination graph, they inherit some existing results (i.e., it back-dates its anchor page to an earlier time). As such, it has a better description about the data distribution, namely variances of the data. On the contrary, $AQP - Baseline$ and $AQP - Direct$ retrieve samples on the fly. Sometimes, they underestimate the variances of the data when not enough samples are obtained.

$AQP - Direct$ performs better than $AQP - Baseline$, because once a sample is retrieved, it is fed to all query nodes. The samples are shared among the nodes. $AQP - Graph$ performs better for template 1, because we partition the space into 1000 grids for both cases. In template 2, we have more columns involved in the queries and we need more grids to partition them to improve the reuse of the partial results. When we decrease the error bound from 0.02 to 0.01, we need to retrieve more samples than the case of decreasing the error bound from 0.03 to 0.02. If some query requires high accuracy (e.g. 0.0001 error bound), online aggregation may eventually be worse than the complete query processing.

The predefined confidence gives the probability that the accurate result is bounded by the estimated error bound. Figure 10 and Figure 11 show the effect of confidence for template 1 and template 2, respectively. The confidence varies from 80% to 99% in both cases. As the confidence increases, more samples are required. This can be explained by Equation 15. $AQP - Graph$ still performs best as the partial results are shared between the nodes.

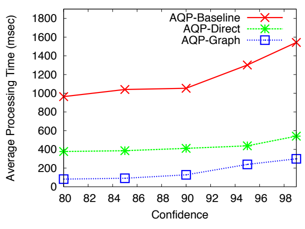


Figure 10: Effect of Confidence (T1)

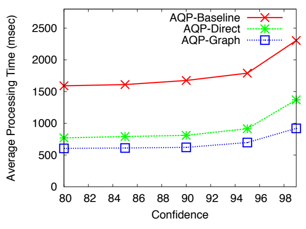


Figure 11: Effect of Confidence (T2)

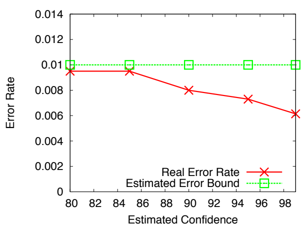


Figure 12: Accuracy of Confidence (T1)

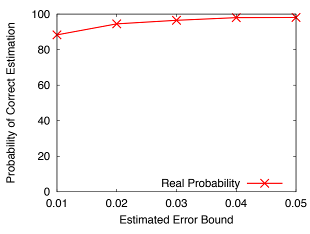


Figure 13: Accuracy of Error Bound (T1)

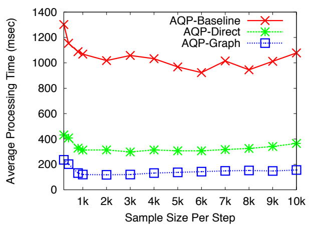


Figure 14: Effect of Sample Buffer (T1)

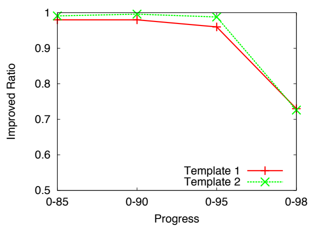


Figure 15: Effect of Result Sharing

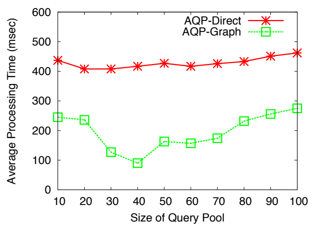


Figure 16: Effect of Concurrency (T1)

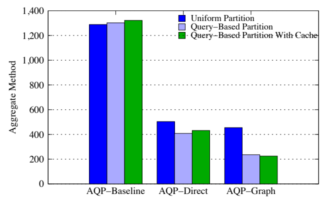


Figure 17: Comparison of Partitioning Method (T1)

5.2.3 *Accuracy of Estimation*

In this and subsequent experiments, as results of template 1 and template 2 show similar trend, we only present the results of template 1. Figure 12 and Figure 13 depict the accuracy of $AQP - Graph$.

In Figure 12, we show the accuracy of the estimated error bound. When a query achieves the default error bound (0.01), it stops its processing and terminates. Then, we compare the approximate result $\bar{V}$ with the precise result V computed by PostgreSQL. The real error rate is calculated as $\frac{V-\bar{V}}{V}$. When the real error rate is less than the estimated bound, the estimation provides an accurate result. Based on the result of Figure 12, online aggregation performs quite well - the real error is always lower than the estimated error. When we increase the confidence, the real error rate also decreases. This is because high confidence causes more samples to be retrieved, which generates a better estimation.

In Figure 13, we test the correctness of the estimated confidence. The confidence is set to the default value, e.g. 95%. We compute the probability of the accurate result being bounded by the estimated result. Specifically, we record the results, error bounds and confidences generated by $AQP-Graph$. And the same set of queries are submitted to PostgreSQL for obtaining the accurate results. Suppose for a query Q_i, its precise result is V_i and the estimated result is $\bar{V}_i$. Let ϵ_i be the corresponding error bound. For the query set S_q, the correct estimation set is defined as:

$$S_c = \{Q_i | \frac{|V_i - \bar{V}_i|}{V_i} \leq \epsilon\}$$

The real probability is computed as $\frac{|S_c|}{|S_q|}$. As shown in Figure 13, $AQP - Graph$ offers very good estimation - for most queries, the user can assume that V is bounded by $\bar{V} \pm \epsilon\bar{V}$ with the probability of at least the predefined confidence. In Figure 13, we change the error bound from 0.01 to 0.05. The results suggest that as we tighten the bound, $AQP - Graph$ may over-estimate the confidence. Fortunately, the error bound remains acceptable and very good. For example, if we compare with the results in Figure 12 (which has an estimated error bound of 0.01), as mentioned earlier, the actual error bound is lower than 0.01.

5.2.4 *Effect of Sample Buffer Size*

The sample stream root continuously retrieves samples to its buffer. When the buffer is full, it forwards the samples to the subsequent query nodes as a batch. This strategy aims to reduce the update cost of query nodes. If we update the partial results and statistics for each incoming sample, the update cost will dominate the performance and online aggregation may degrade. Figure 14 illustrates the effect of the sample buffer. We vary the size of the sample buffer from 200 to 10000 tuples. At first, when we increase the sample buffer, the performance of all approximate schemes improve. However, after the size of the sample buffer increases to 1000 tuples, the performance does not improve any more. Instead, if we continue to increase the buffer size to beyond 3000 tuples, the performance began to degrade. As a query will terminate if it satisfies the predefined error bound and confidence, having a large sample buffer translates to a longer waiting time since the updates happen periodically. Moreover, setting the sample buffer to a large value will slow down the update of results, which affects the user's experience. Therefore, the size of sample buffer is a tradeoff between the performance and user's experience.

5.2.5 *Effect of Result Sharing*

In Figure 15, we show the effect of result sharing. Specifically, the first 100 queries are used to warm up the system and we compare the progresses of the remaining 900 queries in $AQP - Direct$ and $AQP - Graph$. The error bound is set to 0.01 and we record the costs of reaching confidence 85%, 90%, 95% and 98%. We show the improved ratio, which is computed as $1 - \frac{cost(AQP-Graph)}{cost(AQP-Direct)}$. With result sharing, $AQP - Graph$ is much faster than $AQP - Direct$, especially

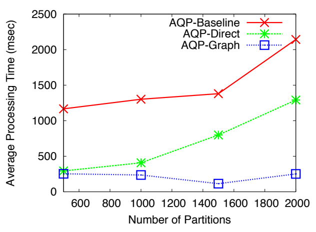


Figure 18: Effect of Partition Numbers (T1)

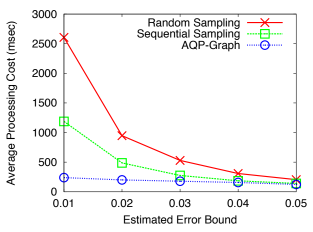


Figure 19: Scan Vs Sampling (Processing Cost)

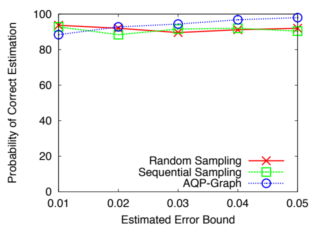


Figure 20: Scan Vs Sampling (Accuracy)

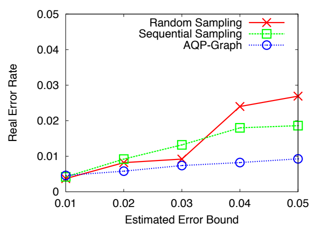


Figure 21: Scan Vs Sampling (Error Rate)

for low confidences, as low confidences do not require too many samples. In fact, for some queries, by exploiting existing samples and results, we can directly provide good enough results (e.g. error bound=0.01 and confidence=90%).

5.2.6 *Effect of Concurrency*

In $AQP-Direct$ and $AQP-Graph$, multiple queries are processed concurrently. Specifically, the incoming query will be inserted into the pool, if it is not full. Otherwise, it must wait until some existing queries finish their processing. The size of the query pool affects the degree of concurrency. In Figure 16, we change the size of query pool from 10 to 100 (default value is 20). Both $AQP-Direct$ and $AQP-Graph$ show similar behavior. At first, their performance improves as more queries are being processed concurrently. This is because the queries can share their samples and partial results. However, after the size of the query pool increases to a certain value, the performance starts to degrade. This shows that concurrent processing also incurs some additional overheads. In both $AQP-Direct$ and $AQP-Graph$, the stream root needs to disseminate samples to different query nodes. Given a sample, the root computes the grid ID of the sample and checks whether a query node has registered for the grid. The cost of this process increases significantly as more queries are connected to the root. In $AQP-Graph$, more concurrent queries also lead to a more complex dissemination graph, which will incur higher cost when we share results among the queries.

5.2.7 *Comparison of Partitioning Methods*

So far, in our experimental study, we have used the query-based grid partitioning method to facilitate the reuse of samples. In Figure 17, we compare the uniform partitioning method, query-based partitioning method and query-based partitioning method with cache. In uniform partitioning, we partition each column into equal-size sub-ranges. And all columns have approximately the same number of partitions. In both the uniform partitioning and query-based partitioning, the total number of partitions is set to 1000. Figure 17 shows that for $AQP-Baseline$ and $AQP-Direct$, partitioning methods do not affect the performance significantly, while for $AQP-Graph$, query-based partitioning performs much better than the uniform one. This is because in $AQP-Graph$ we also share the partial results among the queries. Query-based partitioning helps to improve the probability of reusing a grid. When enchanced with cache, all methods provide a better accuracy. However, in terms of processing time, $AQP-Graph$ performs slightly better, while the other methods perform even worse. This is because maintaining cache incurs additional overheads, which may reduce its benefit.

In Figure 18, we evaluate the performance of the query-based partitioning method by varying the number of partitions from 500 to 2000. For $AQP-Baseline$ and $AQP-Direct$, when the number of partitions increases, the performance decreases. This is because 1) we compute the grid ID of each sample to disseminate the samples and 2) the final result is generated by combining results of all involved grids. The above two costs are proportional to the number of grids. On the contrary, $AQP-Graph$'s performance improves when we increase the number of partitions from 500 to 1500, as more partial results can be shared among nodes. But if we continuously increase the number of partitions to 2000, $AQP-Graph$'s performance also decreases due to the computation overheads.

5.2.8 *Sequential Scan Vs. Random Sampling*

In this experiment, we compare three approximate processing methods, *Sequential Scan*, *Random Sampling* and *AQP-Graph*. We set the predefined confidence to 90%. *Sequential Scan* reads the scrambled dataset continuously and applies the retrieved tuple as a random sample. *Random Sampling* randomly selects a tuple from the unscrambled dataset as a valid sample. Figure 19, Figure 20 and Figure 21 show the comparison results of processing time, accuracy of confidence and real error rate, respectively. *Sequential Scan* performs significantly better than *Random Sampling* due to its sequential I/O. *Sequential Scan* achieves a similar performance to *AQP-Graph*, when the estimated error bound is set to 0.05. In that case, only a few samples are retrieved and therefore sharing samples cannot benefit a lot. However, 0.05 is quite a large error for computing *sum* or *count*. Figure 20 and Figure 21 show that all methods can provide a good estimation.

6. RELATED WORKS

6.1 Online Aggregations

Online aggregation was first proposed in [8]. Several modifications to the database engine was proposed to support online aggregation. These include techniques to randomly access data, to evaluate operations (such as join and sort) without blocking, to incorporate statistical analysis [6], etc.

In [7], Haas and Hellerstein proposed a new family of join algorithms, called ripple joins, which are effective when an aggregate is to be performed online. [13] enhances the original ripple join algorithms by combining parallelism with sampling to speed query convergence. It maintains a good performance even when the memory overflows. More recently, Wu et. al. studied online aggregation in a distributed setting [19]. The scheme maintains synopses which are essentially samples that can be reused by different queries. However, there is no discussion on how partial answers can be exploited.

In real systems, queries from different users are submitted concurrently. Existing work on online aggregation focuses on single query optimization, while in this paper, we share samples and partial answers among concurrent queries. Our approach effectively reduces the processing overheads.

Online aggregation is based on the assumption of random samples. In [5], a new sampling technique, outlier-indexing, is proposed to retrieve random samples for dataset with skewed distribution. By combining weighted samples from uniform sampling and outlier-indexing, [5] can provide an aggregate result with significantly reduced approximation error. Most work assumes the samples are small in size, whereas in [11], an online algorithm is used to maintain large-scale on-disk samples. The algorithm is suitable for both biased and unequal probability sampling. In [1, 2], precomputed samples are maintained to support approximate query processing. Samples are selected in the preprocessing phase. The scrambled dataset in COSMOS can be considered as precomputed samples as well. COSMOS retrieves the samples in an online way. It allows the sample size to dynamically change from a small number that satisfies the precision requirement to a full scan that returns complete result.

6.2 Multi Query Optimization

The basic idea of multi query optimization is to share results of common sub-queries. In [18], three cost-based heuristic algorithms are proposed to search common sub-expressions among concurrent queries. The query plan is changed adaptively to share the results of sub-expressions. In [12, 14], views are dynamically materialized and maintained to process incoming queries. The views are selected based on the common sub-expressions and maintenance costs. In [16, 17], to speed up aggregate queries, the authors proposed to precompute aggregates, and develop data structures that allow these precomputed aggregates to be accessed quickly. The data space is also partitioned, and additional meta-data are maintained in the data structures to enable the system to present the error bounds of the answers.

In COSMOS, we also share partial answers among queries. But as we adopt online aggregation techniques, COSMOS does not materialize the views or buffer the tuples of completed queries. Instead, the samples are continuously drawn and discarded. Only partial results of specific grids are maintained to facilitate query processing. In other words, queries share the table scan asynchronously. They can join or leave the dissemination tree at any time. Compared to existing solutions, our approach has low maintenance cost and can adaptively adjust its buffered results.

7. CONCLUSIONS

In this paper, we have examined the problem of deploying online aggregation to evaluate multiple aggregate queries. We proposed COSMOS, a system that (a) preprocesses a dataset so that we can sequentially scan it for random samples, (b) organizes queries into a dissemination graph to salvage partial results generated from certain queries that share the space the query covers, (c) estimates query answers using a two-step mechanism. We have evaluated COSMOS using the TPC-H benchmark, and our results showed the superiority of COSMOS over traditional and naive methods.

8. REFERENCES

[1] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *SIGMOD*, pages 487–498, 2000.

[2] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *SIGMOD*, pages 539–550, 2003.

[3] T. F. Bowen, G. Gopal, G. E. Herman, T. M. Hickey, K. C. Lee, W. H. Mansfield, J. Raitz, and A. Weinrib. The datacycle architecture. *Commun. ACM*, 35(12):71–81, 1992.

[4] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *PVLDB*, 2(1):277–288, 2009.

[5] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. R. Narasayya. Overcoming limitations of sampling for aggregation queries. In *ICDE*, 2001.

[6] P. J. Haas. Large-sample and deterministic confidence intervals for online aggregation. In *SSDBM*, 1997.

[7] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD Conference*, 1999.

[8] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD Conference*, 1997.

[9] A. Jacobs. The pathologies of big data. *Commun. ACM*, 52(8):36–44, 2009.

[10] C. Jermaine, A. Dobra, S. Arumugam, S. Joshi, and A. Pol. A disk-based join with probabilistic guarantees. In *SIGMOD Conference*, 2005.

[11] C. Jermaine, A. Pol, and S. Arumugam. Online maintenance of very large random samples. In *SIGMOD Conference*, pages 299–310, 2004.

[12] Y. Kotidis and N. Roussopoulos. Dynamat: a dynamic view management system for data warehouses. *SIGMOD Rec.*, 28(2):371–382, 1999.

[13] G. Luo, C. J. Ellmann, P. J. Haas, and J. F. Naughton. A scalable hash ripple join algorithm. In *SIGMOD Conference*, pages 252–262, 2002.

[14] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *SIGMOD*, pages 307–318, 2001.

[15] J. K. Patel and C. B. Read. Handbook of the normal distribution. 1996.

[16] M. Riedewald, D. Agrawal, and A. E. Abbadi. pcube: Update-efficient online aggregation with progressive feedback and error bounds. In *SSDBM*, 2000.

[17] M. Riedewald, D. Agrawal, and A. E. Abbadi. Flexible data cubes for online aggregation. In *ICDT*, 2001.

[18] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD Rec.*, 29(2):249–260, 2000.

[19] S. Wu, S. Jiang, B. C. Ooi, and K. L. Tan. Distributed online aggregation. *PVLDB*, 2(1):443–454, 2009.