G-OLA: Generalized On-Line Aggregation for Interactive Analysis on Big Data

Kai Zeng[†] Sameer Agarwal[‡] Ankur Dave[†] Michael Armbrust[‡] Ion Stoica[†]
[†]University of California, Berkeley [‡]Databricks Inc.
[†]{kaizeng, ankurd, istoica}@cs.berkeley.edu [‡]{sameer, michael}@databricks.com

ABSTRACT

Nearly 15 years ago, Hellerstein, Haas and Wang proposed online aggregation (OLA), a technique that allows users to (1) observe the progress of a query by showing iteratively refined approximate answers, and (2) stop the query execution once its result achieves the desired accuracy. In this demonstration, we present G-OLA, a novel mini-batch execution model that generalizes OLA to support general OLAP queries with arbitrarily nested aggregates using efficient delta maintenance techniques. We have implemented G-OLA in FluoDB, a parallel online query execution framework that is built on top of the Spark cluster computing framework that can scale to massive data sets. We will demonstrate FluoDB on a cluster of 100 machines processing roughly 10TB of real-world session logs from a video-sharing website. Using an ad optimization and an A/B testing based scenario, we will enable users to perform real-time data analysis via web-based query consoles and dashboards.

1. INTRODUCTION

More and more organizations are increasingly turning towards extracting value from huge amounts of data. In many cases, such value primarily comes from data-driven decisions. Hundreds of thousands of servers, in data centers owned by corporations and businesses, log millions of records every second. These records contain a variety of information—highly confidential financial or medical transactions, visitor information or even web content—and require analysts to run exploratory queries on huge volumes of data. For example, continuously analyzing large volumes of system logs can reveal critical performance bottlenecks in large-scale systems or analyzing user access patterns can give useful insights about what content is popular, which in turn can affect the ad placement strategies. While the individual use cases can be wide and varied, many such scenarios value timeliness of query results over perfect accuracy. Furthermore, in many such cases, the analysis is a human-driven interactive process, and the accuracy needs or the time constraints are usually not known a priori for they can dynamically change based on unquantifiable human factors and the insights gained during the analysis.

SIGMOD'15, May 31 - June 04, 2015, Melbourne, VIC, Australia.

http://dx.doi.org/10.1145/2723372.2735381.

A natural solution to support such interactive exploratory analysis on large volumes of data is online aggregation [17] (OLA). With OLA, given an aggregate query, the system presents the user with some *approximate* but *meaningful* result with an associated error estimate (e.g., confidence intervals), as soon as it has processed a small portion of the whole dataset. The approximate result will be continuously refined, at a speed comfortable to the user, while the system is crunching a larger and larger fraction of the whole dataset. This process continues until either the user is satisfied with the accuracy of the query result and stops the query, or the system has processed all the data. This way, OLA gives the user a smooth observation and control over processing, allowing her to make the accuracy-time trade-off on the fly, based on dynamic and unquantifiable factors including accuracy needs and time constraints.

While being incredibly useful, existing OLA techniques are limited to simple SPJA (Select-Project-Join-Aggregation) queries.[1] In particular, existing OLA only works well for queries that are (practically) monotonic, i.e., whose result can be updated by only looking at the new input tuples. Unfortunately, many aggregate queries, such as the ones in popular OLAP workloads, are *not* monotonic. As an example, consider a simplified `Sessions` log, storing the web sessions of users accessing a video-sharing website, with three columns: `session_id`, `buffer_time`, `play_time`. The "Slow Buffering Impact" (`SBI`) query (Example 1) can be used to find out how a longer (than average) buffering time impacts user retention on the website. While `SBI` is a fairly straightforward nested aggregate query, note that it is non-monotonic as any refinement of the inner aggregate `AVG(buffer_time)` could cause to recompute the whole query. This makes it extremely costly for online processing.

Example 1 (Slow Buffering Impact).

```
SELECT AVG(play_time)
FROM Sessions
WHERE buffer_time >
   (SELECT AVG(buffer_time)
    FROM Sessions)
```

In this demonstration, we introduce the G-OLA (Generalized On-Line Aggregation) execution model for interactive exploratory analysis on large volumes of data. G-OLA supports the OLA interface on general OLAP queries—nested aggregate queries with arbitrary nesting and user-defined functions and aggregates. G-OLA accomplishes this by efficient delta maintenance techniques that exploit the convergence property of nested aggregate subqueries.

In addition to generalizing OLA, we argue that G-OLA can significantly simplify the design of Sampling-based Approximate Query Processing (S-AQP) systems, by addressing one of the main chal-

[1]SPJA queries are those that consist of any combinations of select, project and join operators followed by an aggregation operator.

lenges of these systems—that of predicting the optimal sample size given a user-specified accuracy or time constraint. Specifically, due to the limitations of error estimation techniques, it is very hard for existing S-AQP systems to accurately predict the smallest sample size that could satisfy an accuracy criterion. On one hand, closed-form error estimation techniques, such as the ones based on Central Limit Theorem (CLT) or deviation inequalities (e.g., Hoeffding Bounds) can only predict sample size for simple SPJA queries. On the other hand, resampling methods such as bootstrap [14], although can estimate errors for complex SQL queries, lack the ability to predict the sample size. One workaround proposed in BlinkDB [3, 4] is to bootstrap the query on different sample sizes, and fit a profile curve of error vs. sample size based on an extrapolated cost model. However, there are several limitations with this technique. First, it complicates the system design making it harder to tune and manage. Second, it wastes valuable resources and increases the response time which goes against the very purpose of AQP. Finally, the hypothetical model may not hold for complex queries. Similarly, figuring out the optimal sample size given a time constraint requires building an error-latency profile, which is both challenging and error-prone for complex OLAP queries. On the contrary, `G-OLA` gives the user a smooth observation and control of query processing, which does not need to predict the sample size at all.

`G-OLA` is a mini-batch execution model for online processing which randomly partitions the dataset into smaller batches of uniform size, and processes them one by one. `G-OLA` iteratively refines the query results by computing a delta update on each mini-batch of data. The batch granularity is determined by how frequently the user wants the query result to be updated. To avoid recomputing the whole query during delta updates for non-monotonic queries (as shown in Example 1), `G-OLA` carefully partitions the intermediate output of each subquery into two parts — the *uncertain* and *deterministic* sets. The tuples in the deterministic set will not change across mini-batches, while those in the uncertain set may change. Therefore, during delta updates, we only need to consider the tuples in the uncertain set and the new incoming mini-batch. This partitioning is based on the observation that *during online processing, the running results produced by any aggregate subquery should converge to the corresponding final result computed on the whole dataset, and thus their variation decreases with the increase in data size.* Since the uncertain sets are very small in practice (as shown in Section 5), `G-OLA` can achieve continuous and smooth feedback to the user.

We have implemented `G-OLA` in `FluoDB`, an online query execution framework running on Spark[2]. `FluoDB` extends SparkSQL[3] and BlinkDB[4], and is backward compatible with Apache Hive[5]. In this demo, we will demonstrate `FluoDB` running on 100 Amazon EC2 nodes, providing interactive analysis over 10TB session logs from a video-sharing website. The audience will be able to interact with the system through query consoles and dashboards in order to diagnose issues related to ad revenue and user retention in real time. We will also run a traditional SparkSQL engine in batch mode side by side to show `FluoDB`'s superior performance and great user experience.

2. THE G-OLA EXECUTION MODEL

`G-OLA` is designed to support interactive aggregate OLAP queries over massive sets of data. In particular, this new execution model can enable any query processing framework to present the users with meaningful approximate results (with error bars) that are continuously refined and updated during query execution. This not only completely alleviates the need for sampling the data in advance, but also enables the user to observe the progress of a query and control its execution on the fly. `G-OLA` is capable of handling a wide variety of SQL queries including the core relational algebra, standard aggregates (such as `COUNT, SUM, AVG, STDEV` and `QUANTILES`), user-defined functions (`UDFs` and `UDAFs`), subqueries and nested aggregates.

In order to provide statistically unbiased approximate answers to queries on an arbitrary granularity of data, we iteratively execute the query on its input dataset in random order. Furthermore, `G-OLA` also gives precise control to users in specifying only a subset of input relations that needs to be processed in an online fashion. For example, if the `SBI` query were to contain more than one input relations, the user could explicitly specify to stream through a large *fact* table like `Sessions` while reading smaller dimension tables in entirety. By default, `G-OLA` supports partition-wise randomness by randomly picking data partitions to process. This works well when the attributes needed in the query are not correlated with the partitions. However, if this assumption does not hold, `G-OLA` also provides data pre-processing tools to randomly shuffle the entire input dataset, so that any subset of the shuffled data is a uniform sample of the original dataset. Similar to the `POSTGRES-OLA` implementation [17], the users can stop the execution at any time when the result meets their desired accuracy criterion.

2.1 Mini-Batch Execution Model

`G-OLA` uses a mini-batch execution model for online processing. Specifically, we randomly partition the dataset into multiple mini-batches of uniform size, and process through the data by taking a single mini-batch at a time. The batch granularity is determined by how frequently the user wants the query result to be updated. At any point during query processing, `G-OLA` presents the user with the approximate query result and its associated error estimate (e.g., confidence interval) as if the query was computed on all the mini-batches seen so far.

When processing each mini-batch of data, `G-OLA` incrementally maintains the query result previously obtained by computing a delta update on the incoming data. To compute this delta update efficiently, we carefully partition the intermediate output at each operator in the query plan into two parts — the *uncertain* and the *deterministic* sets. With very high probability, tuples in the deterministic set are unlikely to change as we refine the query result. In contrast, the uncertain set of tuples are likely to change when we recompute the query on more data. Therefore, when we process the next mini-batch, we only need to consider the tuples in the uncertain set and the ones in the new mini-batch to compute a delta update. We will explain our delta maintenance techniques in detail in Section 3.

2.2 Query Semantics

This subsection gives a brief overview of the query execution semantics in `G-OLA`. Consider that the user submits a query Q on a dataset D. During query processing, the system first randomly partitions D into k parts $\Delta D_1, \cdots, \Delta D_k$, where $|\Delta D_1| = \cdots = |\Delta D_k|$. At the i-th iteration, $1 \le i \le k$, it processes partition ΔD_i, and returns $Q(D_i, \frac{k}{i})$ as an approximation to the final result $Q(D)$, along with the error estimation (*e.g.,* a confidence interval) of $Q(D_i, \frac{k}{i})$, where D_i is the dataset $\Delta D_1 \cup \cdots \cup \Delta D_i$; $Q(D_i, \frac{k}{i})$ is simply evaluating Q on enhanced D_i where each tuple is annotated with a multiplicity $m = \frac{k}{i}$. This annotated multiplicity means that seeing a tuple in $\Delta D_1 \cup \cdots \cup \Delta D_i$ is roughly equivalent to it being seen m times in D (as $\Delta D_1 \cup \cdots \cup \Delta D_i$ is a random sample from D). Computing $Q(D_i, \frac{k}{i})$ follows the standard multiset semantics that are exposed

[2]https://spark.apache.org/
[3]https://spark.apache.org/sql/
[4]http://blinkdb.org
[5]https://hive.apache.org/

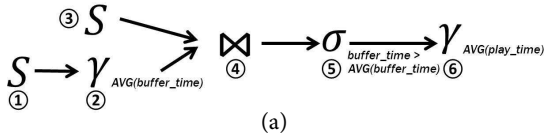
in many commercial databases. For the sake of simplicity, we will use $Q(D_i)$ and $Q(D_i, \frac{k}{i})$ interchangeably throughout the paper.

In G-OLA, we use the *bootstrap* [14] to estimate the confidence interval of $Q(D_i)$ with respect to $Q(D)$, that consists of a simple Monte-Carlo procedure—it repeatedly carries out a sub-routine called a *trial*. Each trial generates a simulated database, say $\hat{D}_{i,j}$, which is of the same size as D_i (by sampling $|D_i|$ tuples i.i.d. from D_i *with* replacement), and then computes query Q on $\hat{D}_{i,j}$. The collection $\{Q(\hat{D}_{i,j})\}$ from all the bootstrap trials forms an empirical distribution, based on which a confidence interval can be computed.

3. DELTA MAINTENANCE IN G-OLA

In this section, we describe G-OLA's delta maintenance techniques in detail. As an illustration, consider answering the SBI query (Example 1) on the dataset shown in Figure 1(b). Figure 1(a) depicts the query plan of the SBI query.

To process the SBI query, G-OLA partitions the Sessions relation into k mini-batches $\{\Delta D_1, \cdots, \Delta D_k\}$ where each ΔD_i is of size n, e.g., $\Delta D_1 = \{t_1, \cdots t_n\}$ and $\Delta D_2 = \{t_{n+1}, \cdots t_{2n}\}$ and so on. G-OLA processes through these chunks incrementally, delivering an approximate result of the SBI query (denoted by Q) at the i-th mini-batch as if Q is computed on $D_i = \Delta D_1 \cup \cdots \cup \Delta D_i$ (See Section 2.2). However, instead of computing $Q(D_i)$ directly, we treat D_i as D_{i-1} updated by inserting new tuples ΔD_i (i.e., $D_i = D_{i-1} \cup \Delta D_i$), and compute $Q(D_i)$ by applying a delta query to the result obtained in the $(i-1)$-th iteration (i.e., $Q(D_i) = Q(D_{i-1}) + \Delta Q(D_{i-1}, \Delta D_i)$)[6]. The intuition is that computing $\Delta Q(D_{i-1}, \Delta D_i)$, and using the result to update $Q(D_{i-1})$, would be much faster than recomputing $Q(D_i)$ from scratch, enabling quicker feedback to the user. Similar intuition is shared by online aggregation [17] and the work on incremental view maintenance [5, 16, 19].



(a)

	session_id	buffer_time (s)	play_time (s)
t_1	s_1	36	238
t_2	s_2	58	135
⋯	⋯	⋯	⋯
t_n	s_n	17	617
t_{n+1}	s_{n+1}	56	194
t_{n+2}	s_{n+2}	19	308
⋯	⋯	⋯	⋯
t_{2n}	s_{2n}	26	319
⋯	⋯	⋯	⋯

(b) *Sessions*

Figure 1: (a) The query plan of the SBI query in Example 1, and (b) an example of the Sessions relation.

3.1 Issues with Classical Delta Maintenance

Unfortunately, the above intuition does not always hold: for complex OLAP queries involving nested aggregates, e.g., the SBI query, computing $\Delta_i Q(D_{i-1}, \Delta D_i)$ using classical delta maintenance is no better than computing $Q(D_i)$.

This is due to the non-monotonicity of some relational operators. In particular, the aggregation operator, which is of extreme importance in OLAP, is blocking in nature [7], i.e., aggregation will not produce an accurate answer until it sees all the input. This blocking nature of aggregation makes delta maintenance of OLAP queries very challenging—as more and more data comes in, the query engine may flip its previous decision, resulting in completely recomputing the query on all the data that was seen previously.

For instance, consider the first two mini-batches of the SBI query. In the first batch, computing the inner aggregate query AVG(buffer_time) on D_1 results in 37. As a consequence, tuple t_1 is filtered out as its buffer_time is lower than the running average buffering time. However, in the second batch, after taking ΔD_2 into account, AVG(buffer_time) is updated to 35.3, which in turn requires t_1 to be selected. Unfortunately, since t_1 is already dropped in the first batch, the query engine has to read through D_1 again in order to compute the correct answer for the second batch. Even worse, such wasteful re-computation may recur at each batch. In general, at the i-th batch, we need to compute the query on all the tuples seen previously (i.e., D_{i-1}) alongside the new data ΔD_i. Therefore, the update cost increases linearly with batches, hindering continuous update. Roughly, processing through all the k mini-batches will process $O(k^2) \cdot n$ of data in total, which could be much larger than the original dataset (of size kn).

3.2 Discovering Certainty in Uncertainty

We have seen that due to their blocking nature, the running results of aggregates that are computed on samples, are approximate and uncertain. This makes complex OLAP queries with nested aggregates non-monotonic, making simple delta maintenance techniques extremely inefficient. However, there is a key principle behind all S-AQP techniques—running aggregate results will eventually converge to the ground truth (i.e., the aggregate result computed on the full dataset) as the sample size increases. Therefore, as the query engine processes through the mini-batches, the intermediate aggregate results will concentrate in a relatively small range around the ground truth, and this range will shrink as more and more batches are processed.

G-OLA utilizes this important property in developing an efficient delta maintenance technique. As an example, assume that all the intermediate results of AVG(buffer_time) throughout its online processing are within the range of 37 ± 8.1. This implies that across all mini-batches, tuple t_2 will be selected, while tuple t_n will be filtered out. For these tuples, the decisions made by the query engine will never change across all mini-batches. Thus, if we know this fact a priori, we can prune these non-uncertain tuples during online processing. Formally, for an uncertain[7] attribute u in an intermediate tuple, we define its *variation range* as the set of all the possible values that u may take during the online execution, denoted by $\mathfrak{R}(u)$. For simplicity, we uniformly define the variation range of a deterministic value d as itself, i.e., $\mathfrak{R}(d) = \{d\}$.

Next, for simplicity, we will explain G-OLA's delta-maintenance algorithm by first assuming that these variation ranges are given, and then explain how these variation ranges can be approximated in practice. In the i-th mini-batch, at any predicate $x\ \theta\ y$ involving uncertain values,[8] G-OLA classifies the input tuples into two sets: the *uncertain* set U_i in which tuples satisfy $\mathfrak{R}(x) \cap \mathfrak{R}(y) \neq \emptyset$, and the *deterministic* set C_i in which tuples satisfy $\mathfrak{R}(x) \cap \mathfrak{R}(y) = \emptyset$. For instance, if $\mathfrak{R}(\text{AVG(buffer\_time)}) = [28.9, 45.1]$, then $t_2, t_n \in C_1$, while $t_1 \in U_1$. Clearly, for the tuples in U_i, the predicate may evaluate to different answers in different batches, while for the tuples in C_i, the predicate will evaluate to the same answer across all batches. Therefore, in batch $(i-1)$, we cache U_{i-1}; and in batch i, instead

[6] Here, + is a way of combining a relation with a change to it. This could include inserting new tuples, deleting old tuples, or updating an existing tuple with new attribute values.

[7] An attribute is uncertain if it is computed by a nested aggregate subquery.

[8] θ is some comparison operator.

of computing $Q(D_i)$ from scratch, G-OLA only needs to compute a delta update based on U_{i-1} and ΔD_i.

Of course, the variation ranges cannot be known until we have finished the query. In practice, G-OLA approximates the variation ranges using running estimates. Recall that we use bootstrap to estimate the accuracy of the running query results. As a by-product of this process, we can obtain a set of bootstrap outputs $\hat{u}$ for each uncertain value u, where $\hat{u}$ is shown to be an accurate approximation of the true distribution of u[9]. In practice, we use the range defined by $\hat{\mathfrak{R}}(u) = [min(\hat{u}) - \varepsilon, max(\hat{u}) + \varepsilon]$ to approximate $\mathfrak{R}(u)$, where ε is a slack variable which can controlled by the user. $\hat{\mathfrak{R}}(u)$ may fail in the sense that some running value of u or a bootstrap output in $\hat{u}$ exceed the variation range in some batch, which will result in incorrect query answers. However, the system can detect this *failure*, and can correct the answer by recomputing the query on the data seen since the last mini-batch with the correct variation ranges. The user can also decrease the chance of recomputation by setting a larger ε (at the cost of increasing the size of the uncertain sets). In practice, setting ε to the standard deviation of $\hat{u}$ achieves a good balance in controlling the probability of recomputation and reducing the size of the uncertain sets.

3.3 Lineage and Lazy Evaluation

In order to correctly compute the delta update in batch i, U_{i-1} cached in batch $i-1$ has to be updated with the latest values. For instance, consider the above example where tuple t_1 is classified in U_1. In batch 2, t_1 has to be updated with the latest `AVG(buffer_time)` in order to evaluate the predicate correctly.

To efficiently support this update and avoid regenerating the uncertain tuples from scratch, G-OLA keeps track of all the values used to compute the uncertain attributes in a tuple, i.e., its *lineage*, and propagates the lineage with each tuple. During delta maintenance, we update the uncertain set by re-evaluating the operation on the lineage carried with each uncertain tuple, and this evaluation is done lazily when the corresponding values are accessed.

However, since aggregates are computed from a set of values, propagating the lineage of aggregates will cause an explosion in the storage and networking overhead. We optimize the lineage propagation by dividing a query into multiple *lineage blocks*. A lineage block is a maximal subtree of the query plan that is an SPJA (Select-Project-Join-Aggregation) block, i.e., a subtree consisting of any combinations of select, project and join operators followed by an aggregation operator. It is maximal in the sense that extending the subtree with any node in the query plan will violate this requirement. As an example, the query plan shown in Figure 1(a) can be divided into two lineage blocks: $\{①, ②\}, \{③, ④, ⑤, ⑥\}$. G-OLA propagates lineage within each lineage block, while simply broadcasts the latest aggregate results between lineage blocks, thus bounding the overall cost of lineage propagation.

4. SYSTEM ARCHITECTURE

We have implemented G-OLA in FluoDB. FluoDB is a parallel online query execution framework that is build on top of SparkSQL and BlinkDB, and enables us to be backwards compatible with Apache Hive in terms of both storage and query language. Underneath, it runs on Spark, a distributed computing framework that supports efficient in-memory computation. Figure 2 depicts the system architecture of FluoDB. FluoDB extends the optimization framework of SparkSQL, poissonized resampling based error estimation techniques of BlinkDB and the computation infrastructure

[9]We refer interested readers to [3] for the implementation details of bootstrap.

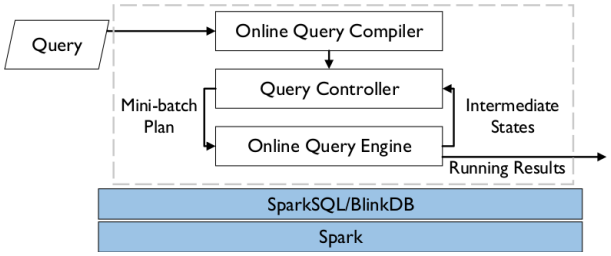


Figure 2: The system architecture of FluoDB.

of Spark, by adding three major components to empower online execution:

1. **Online Query Compiler**. The online query compiler compiles the query into a meta query plan, which when plugged with different mini-batches of data, turns into a series of mini-batch queries. In this series, each mini-batch query depends on the state computed in the previous iteration, and computes delta-updates on the results of its predecessor.

2. **Query Controller**. The query controller is in charge of partitioning the input data into mini-batches, and generating/scheduling the series of mini-batch queries given a meta query plan. It also manages, caches and distributes the intermediary uncertain sets during each iteration. Since our delta-maintenance algorithm (see Section 3) relies on the correctness of the running approximate variation ranges, the controller monitors the correctness of all the variation ranges as well, and schedules recomputing jobs to correct the query results when a failure is detected.

3. **Online Query Engine**. The online query engine implements a set of online relational algebra operators on Spark.

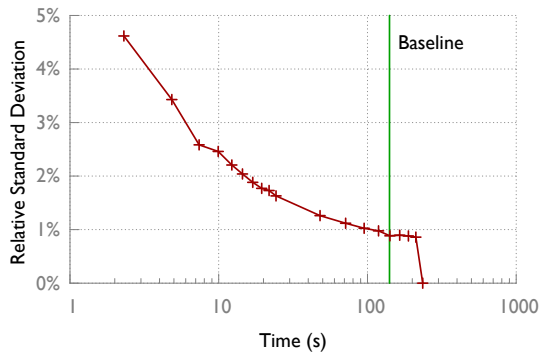
5. PERFORMANCE

In this section, we briefly discuss the performance of G-OLA. Our experiments are based on a synthetic 100GB dataset from the TPC-H benchmark[10], and a 100GB subset of a 10TB anonymized real-world video content distribution workload from Conviva Inc.[11] To simplify random partitioning during mini-batch execution, we denormalize the TPC-H data into a single fact table. The Conviva dataset contains a single de-normalized fact table. Our queries are based on a subset of the TPC-H benchmark queries and the original Conviva query trace.

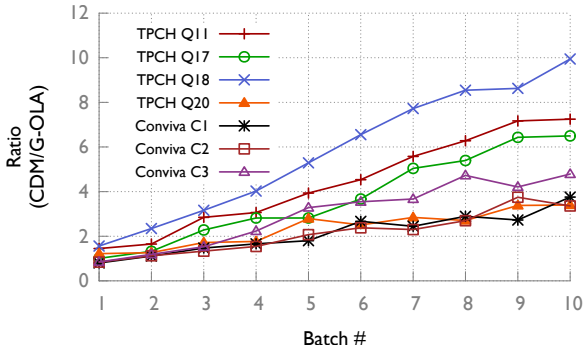
We compare the performance of G-OLA with existing online processing techniques in OLA [17] and incremental view maintenance work [5, 16, 19] by processing a set of illustrative queries in mini-batches of 1GB each. To clearly demonstrate the expressiveness and effectiveness of G-OLA, we focus only on complex OLAP queries with nested aggregates. Queries C1, C2, C3 are based on the Conviva query trace and compute statistics (such as histograms of `play_time` and `join_failure_rate`) of sessions with abnormal behaviors (e.g., those with a longer than average buffering time). Queries Q11, Q17, Q18, Q20 are from the TPC-H benchmark.[12]

Figure 3(a) demonstrates a typical query process in G-OLA using TPC-H Q17. As one can see, any traditional query engine will only

[10]http://www.tpc.org/tpch/
[11]http://www.conviva.com/
[12] Please note that while we retained the original structure of the TPC-H queries, we had to modify some very selective `WHERE` and `GROUP BY` clauses in the original queries to avoid undesirably sparse results for small samples of data.



(a)



(b)

Figure 3: (a) The relative standard deviation vs. query time curve delivered by G-OLA on TPC-H Q17. We plot the values for the first 10 mini-batches, and then for brevity, the 20th, 30th mini-batch and so on. (b) The ratio of query times of the first 10 batches of G-OLA and classical delta maintenance (CDM) techniques. Note that the ratio of query times for each iteration grows linearly with the number of iterations.

be able to deliver an answer after processing the entire dataset, which in this case, would incur 2.34 minutes latency (marked by the vertical bar). On the other hand, G-OLA can deliver an approximate answer in 2.3 secs (i.e., only in about 1.6% of the whole query time). Furthermore, G-OLA continuously refines the answer at a very user-friendly pace of roughly every 2.5 seconds. It is worth noting that while G-OLA incurs an additional 60% overhead in processing the whole dataset as compared to the baseline (primarily due to the error estimation overheads), it enables the user to make a smooth trade-off between error and latency by allowing her to stop the query execution at any time. For instance, if the user is satisfied with an accuracy of say, 2% relative standard deviation, she can stop the query at 16 seconds, which is almost 10× faster than a batched execution.

We also compare the delta maintenance techniques in G-OLA with the classical delta maintenance techniques. The results for the query response time for the first 10 mini-batches (of size 1GB each) are shown in Figure 3(b). We plot the ratio of the query times spent by classical delta maintenance algorithms (CDM) and G-OLA for each batch. While these queries and datasets cannot necessarily form a representative sample of the incredibly wide range of real-world OLAP queries and datasets, we observe that G-OLA significantly outperforms the classical delta-maintenance algorithms. In the classical algorithms, every update on an inner aggregate subquery causes the engine to recompute the query on the entire data that was previously processed, while G-OLA could effectively limit the size of the

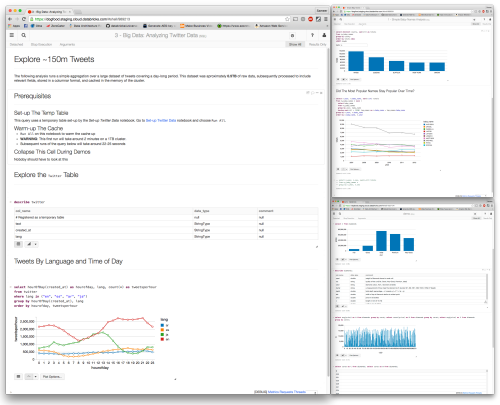


Figure 4: Sample screenshots of an interactive web-based console that will allow attendees to launch custom online SQL queries on the underlying data.

uncertain set of tuples in each iteration, achieving almost constant query time for each iteration.

6. DEMONSTRATION DETAILS

Our demonstration scenario places the attendees in the shoes of a Data Scientist at *MyTube Inc.*, a popular (and imaginary) video-sharing website that wants to adapt its policies and decisions in near real time to maximize ad revenue and improve user retention. *MyTube* collects a variety of user metrics (ranging from sessions IDs, content IDs, video start/stop times or geographical locations) that it aggregates across multiple dimensions to continuously derive insights about its viewers.

As part of this demo, the attendees will be able to interact with a web-based dashboard that will compute and plot a number of ad popularity and user retention metrics while cycling through various user groups and/or geographical regions. Since each of these metrics are computed by fairly complex queries on massive amounts of data, the dashboard will feature approximate answers with error bars that will get progressively refined with time. As shown in Fig. 4, we will also feature a powerful web-based console that would allow the attendees to launch arbitrary SQL aggregate queries on the underlying data. For a side-by-side comparison of performance and user experience, we will also demonstrate a traditional SparkSQL engine running in batched execution mode on the same dataset.

6.1 Data Characteristics

Our 10 TB workload trace would comprise of a de-normalized fact table of session logs from a video-sharing website. This table will consist of billions of rows and hundreds of columns (ranging from sessionIDs, contentIDs, adIDs, and video or ad start/end times), and will be stored in HDFS (or partially cached in memory) across 100 Amazon EC2 machines. Each tuple in the table will correspond to a session log entry, such as a video or an ad being played.

6.2 Demonstration Scenarios

To demonstrate the power and expressiveness of G-OLA in terms of the types of queries it can support, we will feature two real-time data analysis scenarios that are closely modeled after real-world use-cases:

Real-time Ad Optimization. In this scenario, *MyTube Inc.* wants to adapt its policies and decisions in near real time to maximize its ad revenue. This involves aggregating over a number of user metrics across multiple dimensions to understand how an ad performs for a particular group of users or content at a particular time of day.

Performing such analysis quickly is essential, especially when there is a change in the environment, *e.g.*, new ads, or a new page layout. The ability to re-optimize the ad placement every minute as opposed to every day or week often leads to a material difference in revenue.

A/B Testing. In this scenario, *MyTube Inc.* aims to optimize its business by improving user retention, or by increasing their user engagement. Often this is done by using A/B testing to experiment with anything from new content to slight changes in the number and/or the duration of ads that are shown. While the number of combinations and changes that one can test is daunting, the ability to quickly understand the impact of various tests and identify important trends is critical to rapidly improving the business.

In these and many other scenarios, the queries are unpredictable (because the exact problem, or the query is not known in advance) and quick response time is essential as data is changing quickly, and the potential profit is inversely proportional to the response time. In this demo, we will specifically focus on these scenarios.

7. RELATED WORK

Online Aggregation. Online aggregation [17] and its successors [13, 20] proposed the idea of allowing users to observe the progress of aggregation queries and control the execution on the fly. The users can trade accuracy for time in a smooth manner. However, online aggregation is limited to simple SPJA queries without any support for nested aggregation subqueries.

Sampling-based Approximate Query Processing. There has been substantial work on using sampling to provide approximate query answers, many of which [2, 4, 8, 11, 21] focus on constructing the optimal samples to improve query accuracy. STRAT [11], SciBORQ [21], Babcock et al. [8] and AQUA [2] construct and/or pick the optimal stratified samples given a fixed time budget, but do not allow users to specify an error bound for a query. BlinkDB [4] supports sample selection given user-specified time or accuracy constraints. Such selection relies on an error-latency profile, which is built for a query by repeatedly trying out the query on smaller sample sizes and extrapolating the points.

Incremental View Maintenance. Incremental view maintenance (IVM) is a very important topic in database view management, and has been studied for over three decades. IVM focuses on a similar problem—computing a *delta update* query when the input data is updated. Maintaining SQL query answers have been studied in both the *set* [9, 10] and *bag* [12, 16] semantics. Computing the delta update query has been studied for query with aggregates [16, 19] and temporal aggregates [24]. However, majority of work in this area only focuses on simple SPJA queries without nested and correlated aggregation subqueries. More recently, DBToaster [5] has investigated higher-order IVM and support for nested queries. However, for queries with nested aggregates, the *delta update* query obtained by higher-order IVM is often no simpler than the original query. `G-OLA`'s delta-maintenance technique doesn't have this limitation.

Data Stream Processing. Data stream processing [1, 15, 18, 22] combines (1) incremental processing (e.g., sliding windows) and (2) sublinear space algorithms for handling updates. These techniques mainly rely on manual programming and composing, and thus have limited adoption and generalization. There has also been work on one-pass streaming algorithms [6, 23] for single layer of nested aggregate queries that rely on building correlated aggregate summaries on streams of data. However, it is non-trivial to automatically build these summaries for queries with arbitrary levels of nesting and/or user defined aggregates. `G-OLA` on the other hand provides automatic incremental processing to general SQL queries, including those with multiple levels of nesting and arbitrary aggregates.

Acknowledgments

This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, C3Energy, Cisco, Cray, Cloudera, EMC, Ericsson, Facebook, Guavus, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Splunk, Virdata and VMware.

8. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[2] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The aqua approximate query answering system. In *SIGMOD Record*, volume 28, pages 574–576, 1999.

[3] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. I. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when you're wrong: building fast and reliable approximate query processing systems. In *SIGMOD*, pages 481–492, 2014.

[4] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42, 2013.

[5] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5(10):968–979, 2012.

[6] R. Ananthakrishna, A. Das, J. Gehrke, F. Korn, S. Muthukrishnan, and D. Srivastava. Efficient approximation of correlated sums on data streams. *IEEE Trans. Knowl. Data Eng.*, 15(3):569–572, 2003.

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

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

[9] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD*, volume 15, pages 61–71, 1986.

[10] O. P. Buneman and E. K. Clemons. Efficiently monitoring relational databases. *TODS*, 4(3):368–382, 1979.

[11] S. Chaudhuri, G. Das, and V. Narasayya. Optimized stratified sampling for approximate query processing. *TODS*, 32(2):9, 2007.

[12] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, pages 190–190, 1995.

[13] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears. Online aggregation and continuous query support in mapreduce. In *SIGMOD*, pages 1115–1118, 2010.

[14] B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall, New York, 1993.

[15] T. M. Ghanem, A. K. Elmagarmid, P.-Å. Larson, and W. G. Aref. Supporting views in data stream management systems. *TODS*, 35(1):1, 2010.

[16] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *ACM SIGMOD Record*, volume 24, pages 328–339, 1995.

[17] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, pages 171–182, 1997.

[18] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system, 2002.

[19] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *PVLDB*, pages 802–813, 2002.

[20] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. volume 4, pages 1135–1145, 2011.

[21] L. Sidirourgos, M. L. Kersten, and P. A. Boncz. Sciborq: Scientific data management with bounds on runtime and quality. In *CIDR*, volume 11, pages 296–301, 2011.

[22] H. Thakkar, N. Laptev, H. Mousavi, B. Mozafari, V. Russo, and C. Zaniolo. SMM: A data stream management system for knowledge discovery. In *ICDE*, pages 757–768, 2011.

[23] S. Tirthapura and D. P. Woodruff. A general method for estimating correlated aggregates over a data stream. In *ICDE*, pages 162–173, 2012.

[24] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In *ICDE*, pages 51–60, 2001.