iOLAP: Managing Uncertainty for Efficient Incremental OLAP

Kai Zeng
Cloud and Information Services Lab, Microsoft
kaizeng@microsoft.com

Sameer Agarwal
Databricks Inc.
sameer@databricks.com

Ion Stoica
University of California, Berkeley
istoica@cs.berkeley.edu

ABSTRACT

The size of data and the complexity of analytics continue to grow along with the need for timely and cost-effective analysis. However, the growth of computation power cannot keep up with the growth of data. This calls for a paradigm shift from traditional batch OLAP processing model to an *incremental OLAP processing model*. In this paper, we propose `iOLAP`, an incremental OLAP query engine that provides a smooth trade-off between query accuracy and latency, and fulfills a full spectrum of user requirements from approximate but timely query execution to a more traditional accurate query execution. `iOLAP` enables interactive incremental query processing using a novel mini-batch execution model—given an OLAP query, `iOLAP` first randomly partitions the input dataset into smaller sets (mini-batches) and then incrementally processes through these mini-batches by executing a *delta update query* on each mini-batch, where each subsequent delta update query computes an update based on the output of the previous one. The key idea behind `iOLAP` is a novel delta update algorithm that models delta processing as an uncertainty propagation problem, and minimizes the recomputation during each subsequent delta update by minimizing the uncertainties in the partial (including intermediate) query results. We implement `iOLAP` on top of Apache Spark and have successfully demonstrated it at scale on over 100 machines. Extensive experiments on a multitude of queries and datasets demonstrate that `iOLAP` can deliver approximate query answers for complex OLAP queries orders of magnitude faster than traditional OLAP engines, while continuously delivering updates every few seconds.

1. INTRODUCTION

Over the last few years, there has been a significant increase in the amount of data that is being collected and analyzed every day to support various data-driven decisions. A major challenge in processing these massive amounts of data comes from the fact that the underlying hardwares cannot get fast or cheap quickly enough to keep up with the data growth[1], which in turn means that cost of decision making will keep increasing. As OLAP queries usually touch a significant amount of data, this mismatch between data growth and computation power shows clearly that the batch processing model used by traditional OLAP processing is no longer suitable for many OLAP applications.

Furthermore, data-driven applications pose a wide variety of requirements on query latency and accuracy—on one hand, business-critical analysis usually requires perfect accuracy, while on the other, time-critical analysis (such as system diagnosis/log processing) value timeliness of query results over perfect accuracy. More recently there is an increasing need for interactive human-driven exploratory analysis, whose desired accuracy or the time-criticality cannot be known *a priori* and change dynamically based on unquantifiable human factors (such as insights gained during the analysis). However, existing OLAP engines either always deliver a perfectly accurate result which may incur long query latency, or intentionally trade off accuracy for latency by only delivering approximate answers on samples of data [7, 9, 12, 17, 34]. No existing OLAP engine can provide a single system that can fulfill the variety of requirements by giving a smooth trade-off between accuracy and latency.

In this paper, we propose a new system—`iOLAP` (for *incremental* OLAP)—to tackle this performance mismatch, by replacing the traditional batch processing model with a more interactive *incremental query processing model*. The core idea behind `iOLAP` is fairly intuitive—given an OLAP query, the system presents the user an *approximate* result with an associated error estimate (e.g., confidence intervals), as soon as it has processed a small portion of the whole dataset. At the same time, the system keeps crunching a larger and larger fraction of the whole dataset, refining the approximate query results and updating the user. This process continues until either the user is satisfied with the accuracy of the query results and stops the query, or the system has processed all the data (in which case it delivers accurate query results just as a traditional DBMS). This approach gives the user a smooth control over query execution and the flexibility to make the accuracy-latency trade-off on the fly, and provides a unified system that covers the whole spectrum, from approximate but timely query answers to perfectly accurate query answers with a larger latency. As an example, compared to a traditional OLAP engine, `iOLAP` can deliver an approximate answer with a 95% accuracy 15× faster, an answer with a 98% accuracy 7× faster, and a perfectly accurate answer by executing the query on the entire dataset at a comparable performance (see Section 8). Last but not least, this approach simplifies the design of existing sampling-based approximate query processing (S-AQP) systems [7, 9, 12, 17, 34] by removing the requirements of pre-generating and maintaining samples.

A limited form of incremental query processing for simple SPJA[2] queries was proposed in Online Aggregation (OLA) [26]. More gen-

[1] According to one recent report [3], data is expected to grow by 64% every year.

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

SIGMOD '16, June 26–July 1, 2016, San Francisco, CA, USA.

DOI: http://dx.doi.org/10.1145/2882903.2915240

erally, incremental query processing has also been extensively studied in the context of delta view maintenance [18, 25, 25, 31, 10]. An underlying limitation of these existing techniques is that they cannot efficiently support anything beyond simple (in most cases, flat SPJA) queries. This implies that for complex queries, such as those with nested comparison subqueries, none of these techniques can incrementally process the query efficiently. In a separate context, incremental processing has also been studied in the context of data stream systems. These systems either rely on manually programming the delta update logic [6, 22, 30, 35] (something that's nontrivial and error-prone for complex OLAP queries) or use progressive models [28, 15, 16] for query processing. While these models are very powerful for modeling delta updates, they often cause a lot of recomputation for complex queries, and hence share the same set of performance bottlenecks as other incremental view maintenance techniques.

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, it is very costly to incrementally process it. This is because as the query executes on larger and larger portions of the `Sessions` table, any refinement of the inner aggregate `AVG(buffer_time)` could result in recomputing the whole outer query on all previously-processed data from `Session`.

Example 1 (SBI: Slow Buffering Impact).

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

In this paper we propose `iOLAP`, a framework that addresses these set of challenges around performance and generality with a novel incremental query processing technique based on uncertainty propagation. `iOLAP` significantly generalizes incremental query processing to complex queries with arbitrary nested subqueries, user-defined functions (UDFs) and user-defined aggregate functions (UDAFs). `iOLAP` has been demonstrated [38] to scale to more than a 100 machines while crunching terabytes of data in parallel. This paper focuses on the theory and implementation details behind the system. `iOLAP` uses a novel *mini-batch* execution model for incremental query processing: Given an OLAP query, `iOLAP` automatically rewrites the query into an enhanced delta query, randomly partitions the dataset into smaller batches, and processes through them by repeatedly executing the enhanced delta query on each mini-batch of data one at a time. Each delta query in the sequence computes a fast delta update of the previous query.

The key idea behind `iOLAP` is a new delta update algorithm based on *uncertainty propagation*. In particular, we propose a novel perspective on the incremental processing problem by treating the changes that could happen in partial results of any operator during incremental processing—either tuple attributes or tuple multiplicities—as *uncertainties*. Note that these uncertainties need to be recomputed during delta updates. Therefore, the problem of efficient delta updates boils down to precisely figuring out uncertainties that *would* change from those that *would not* change at a very fine-grained level. Based on this, `iOLAP` focuses on minimizing the recomputation on uncertainties that would change.

Specifically, we introduce a formal uncertainty propagation theory that categorizes uncertainties into attribute uncertainty (changing attribute) and tuple uncertainty (changing multiplicity), annotate these uncertainties for each tuple, and track their propagation through a relational query plan. **For tuple uncertainties**, we quantify how likely each tuple uncertainty would change, and prune those that would not change. For instance, in Example 1, after executing the query on a certain portion of data, if we were to know that `AVG(buffer_time)` falls within the range $[21.1, 53.9]$, we know that with a very high probability, `buffer_time` = 58 will be always greater than the average, and can be selected in the inner filter. **For attribute uncertainty**, we carry lineage information for each uncertain attribute (i.e., the values used to compute the attribute) within the tuple itself, and compute the up-to-date values of the uncertain attributes in place by only referencing the carried lineage. This avoids generating tuples from scratch during recomputation, and thus maximizes the re-use of previous computation.

In summary this paper claims to make the following contributions:

- We present an uncertainty analysis framework for partial results produced in incremental query processing that provides a dichotomy of the uncertainties in partial results, and reveals how relational operators propagate uncertainties through a query plan. (Section 4)
- Based on the uncertainty propagation theory, we developed a novel delta update algorithm that estimates uncertainties and minimizes delta update recomputation at a tuple/attribute granularity using the lineage-based lazy evaluation.
- We implemented `iOLAP` using a mini-batch execution model that can be easily integrated with existing database engines, and fit well with distributed computing environment. We also conducted an extensive performance study of `iOLAP`.

In the rest of the paper, we firstly define the query model and semantics of `iOLAP` in Section 2. We then demonstrate the limitation of existing delta processing techniques in Section 3. In Section 4, we propose a novel uncertainty propagation theory to model the delta processing problem, develop a delta update algorithm based on that, and discuss the ideas for optimizing tuple uncertainty and attribute uncertainty. We detail the tuple uncertainty optimization and the attribute uncertainty optimization in Section 5 and 6 respectively. We cover the implementation details of `iOLAP` in Section 7, and evaluate the performance in Section 8.

2. QUERY MODEL

When the user submits a query Q on a dataset D, `iOLAP` randomly partitions D into p batches $\Delta D_1, \cdots, \Delta D_p$. It then iteratively crunches through these partitions by processing a single batch as input at a time, i.e., at the i-th ($1 \leq i \leq p$) batch, it processes ΔD_i and delivers to the user a partial query result. As these partial results are computed on samples and thus are approximations of the true query result that is computed on the whole dataset, `iOLAP` also presents an error estimate (e.g., a confidence interval) associated with the partial results. As more and more data is processed, the partial result gets refined periodically with better and better accuracy. Similar to the `POSTGRES-OLA` implementation [26], the users can stop the execution at any time when the result meets their desired accuracy criterion.

In order to provide statistically meaningful approximate answers to queries, `iOLAP` assumes that each batch of data contains a random subset of the entire dataset. By default, `iOLAP` supports block-wise randomness by randomly partitioning data blocks into batches. This works well when the attributes needed in the query are not correlated with the blocks. However, if this assumption does not hold, `iOLAP` also provides data pre-processing tools to randomly shuffle the entire input dataset. Furthermore, `iOLAP` also gives precise control to users in specifying which input relations need to be processed in an online fashion. For example, if the `SBI` query were to con-

tain 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.

Query Semantics. In this paper, we use the relational algebra with bag semantics, but generalize it to tuple multiplicities that are real numbers (see Appendix A for a formal definition). Semantically, given a query Q, the partial query result given by `iOLAP` at batch i is equivalent to computing Q on all the data seen in the first i batches (and scaled appropriately). In other words, let us denote the data processed up to batch i by $D_i = \bigcup_{j=1}^{i} \Delta D_j$, the partial query result delivered in batch i is equivalent to computing Q on D_i where each tuple is annotated with a multiplicity of $m_i = |D|/|D_i|$. This means that seeing a tuple in D_i is roughly equivalent to seeing it m_i times in D. We denote this partial query result by $Q(D_i, m_i)$. Clearly, $Q(D_i, m_i)$ is an approximation to the true query result $Q(D)$. For the sake of simplicity, we use $Q(D_i)$ and $Q(D_i, m_i)$ interchangeably throughout the paper.

Error Estimation. Since $Q(D_i, m_i)$ is an approximation to the true query result $Q(D)$, `iOLAP` also associates an error estimation with the partial answer. In our implementation, we use *bootstrap* [21] to estimate the error of $Q(D_i)$ with respect to $Q(D)$. Bootstrap is a simple Monte-Carlo procedure that repeatedly carries out a subroutine 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 of the query results $\{Q(\hat{D}_{i,j})\}$ obtained from all the bootstrap trials form an empirical distribution, based on which an error measure can be computed. Bootstrap can be efficiently piggybacked with the normal query execution [8, 39].

3. OVERVIEW

To achieve low latency in each batch, the guiding design principle behind `iOLAP` is to take full advantage of delta computation to minimize recomputation (and hence maximizes the reuse of prior work). In other words, instead of computing $Q(D_i)$ from scratch for each batch i, we utilize the fact that $D_i = D_{i-1} + \Delta D_i$, and update $Q(D_{i-1})$ from the previous batch by a delta query $\Delta Q(D_{i-1}, \Delta D_i)$, that is defined by $\Delta Q(D_{i-1}, \Delta D_i) = Q(D_i) - Q(D_{i-1})$. The intuition is that computing $\Delta Q(D_{i-1}, \Delta D_i)$ would be much faster than directly computing $Q(D_i)$. Similar intuition is shared by online aggregation (OLA) [26] and incremental view maintenance [10, 25, 31], with slight differences in the definition of ΔD: For OLA and `iOLAP`, ΔD is insertion of new sample tuples to the previous accumulated sample relation; while for delta view maintenance and streaming systems, ΔD can also include deletion of old tuples.

3.1 Limitations of Existing Approaches

Delta query processing is a well-studied area. Yet, previous approaches can only provide efficient delta update algorithms for simple SPJA queries, and thus fall short in generalizing to complex queries. In particular, we argue that previous delta update techniques are *static* and *coarse-grained*. Specifically, previous approaches generate delta update rules by only utilizing "static" information—the query structure. Such delta update rules (as shown in Figure 1) only exploit the strong compositionality of the relational algebra, but do not consider the relations which the query is evaluated on. As a result, these delta update rules are also "coarse-grained", in the sense that the delta queries in the rules are expressed as a function of the previous relations and the delta relations. Consequently, the rules cannot distinguish which subset of columns–or even more fine-grained, which columns of which subset of tuples–in the final and any intermediary query result are subject to change due to the update. Without fine grained updates, for complex OLAP queries (such as those with evolving nested aggregates in θ and Ψ), a large fraction of the query, sometimes even an entire subquery, needs to be re-evaluated from scratch every time, incurring a lot of overhead. Recently DBToaster [10] proposes to use higher-order delta queries to alleviate these limitations. But it still relies on the delta rules to generate the higher-order queries, and thus shares similar limitations.

- `SELECT`: $\Delta(\sigma_\theta R) = \sigma_\theta(\Delta R)$
- `PROJECT`: $\Delta(\pi_{\bar{A}} R) = \pi_{\bar{A}}(\Delta R)$
- `JOIN`: $\Delta(R_1 \bowtie R_2) = (\Delta R_1 \bowtie R_2) \cup (R_1 \bowtie \Delta R_2) \cup (\Delta R_1 \bowtie \Delta R_2)$
- `UNION`: $\Delta(R_1 \cup R_2) = (\Delta R_1) \cup (\Delta R_2)$
- `AGGREGATE`[3]: $\Delta\gamma_{\bar{A},\Psi=sum} R = \gamma_{\bar{A},\Psi=sum}(\Delta R)$

Figure 1: Delta update rules for simple SPJA queries

As an illustration, let us revisit the `SBI` query (Example 1). Figure 2(a) depicts its query plan. Assume we evaluate the `SBI` query incrementally on the dataset shown in Figure 2(b), where the `Sessions` relation is partitioned into p mini-batches $\{\Delta D_1, \cdots, \Delta D_p\}$, each of size n (e.g., for $n = 3$, $\Delta D_1 = \{t_1, t_2, t_3\}$ and $\Delta D_2 = \{t_4, t_5, t_6\}$ and so on). As we can see, operator ② produces a different `AVG(buffer_time)` with new data added in each batch. This in turn affects ⑤ (that filters tuples using the current value of `AVG(buffer_time)`), and ⑥ (that aggregates the filtered tuples), and requires them to be re-evaluated on all previously-processed tuples in every iteration. Based on the delta update rules in Fig. 1, the delta update query for `SBI` can be written as follows:

$$\Delta\overset{⑥}{\gamma}(\overset{⑤}{\sigma}(D_i \overset{④}{\bowtie} \overset{②}{\gamma}(D_i))) = \overset{⑥}{\gamma}(\overset{⑤}{\sigma}(D_i \overset{④}{\bowtie} (\overset{②}{\gamma}(D_{i-1}) \oplus \overset{②}{\gamma}(\Delta D_i)))) \quad [4]$$

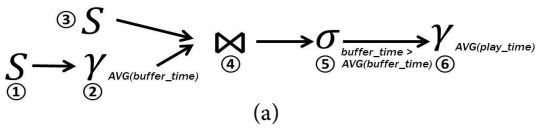
showing that we can only reuse subquery ② (i.e., $\gamma(D_{i-1})$), but have to recompute the subquery involving operators ③, ④, ⑤, ⑥ in Figure 2(a) every iteration. Note that since these static delta update algorithms do not keep track of the distributions of data that has already been *seen*, all recomputations on the same data are performed again and again regardless of whether it is really necessary. For example, let us look into the first two mini-batches. As `AVG(buffer_time)` at ② evaluates to 37 in batch 1 and 35.3 in batch 2, operator ⑤ makes conflicting decisions about tuple t_1: t_1 is filtered out in batch 1 and selected in batch 2. Therefore t_1 needs to be evaluated in batch 2. However, as the `buffer_time` of t_2 (t_3) is much less (greater) than the `AVG(buffer_time)` in both batches, operator ⑤ makes consistent decisions, and thus t_2 and t_3 do not need to be ideally re-evaluated. Due to the coarse-grained delta update rules, existing engines repeatedly re-evaluate ③ to ⑥ on t_2 and t_3 regardless of the above observation incurring significant costs in each batch. Quantitatively, the delta update cost increases linearly with batches, hindering continuous update. Roughly, processing through all the p mini-batches will process $n \cdot O(p^2)$ data in total, which could be much larger than the original dataset of size $p \cdot n$.

3.2 Our Approach

On the contrary, we propose a novel delta update technique that is both *dynamic* and *fine-grained*. Specifically, we treat the changes that could happen in the partial query results as uncertainties in the results, and explicitly track the uncertainty at a much fine-grained level—per column within specific tuples. Based on this tracking, we develop a delta update algorithm that is much more targeted and efficient.

[3] $\gamma_{\bar{A},\Psi}$ represents an `AGGREGATE` operator with group-by column $\bar{A}$ and aggregate function Ψ.

[4] We annotate the equation with their corresponding operators in Figure 2(a). $\oplus$ represents the delta update function for `AVG` aggregate based on the running sum and running count.



(a)

		session_id	buffer_time (s)	play_time (s)
ΔD_1	t_1	id_1	36	238
	t_2	id_2	58	135
	t_3	id_3	17	617
ΔD_2	t_4	id_4	56	194
	t_5	id_5	19	308
	t_6	id_6	26	319
...		...	...	...

(b) *Sessions*

Figure 2: (a) The query plan of the SBI query (Example 1), and (b) an example of the Sessions relation.

Let us revisit the SBI query in Figure 2(a) as an example. By tracking uncertainty, we can identify that—as a result of aggregating incomplete data—the column AVG(buffer_time) is uncertain at ②; despite this uncertain column, the columns outputted by ④ are deterministic as the (empty) join key is deterministic; the selection decisions at ⑤ are uncertain (as they're affected by AVG(buffer_time) which is uncertain); finally, ⑥ aggregates on a deterministic column of an uncertain set of tuples. Furthermore, such uncertainty can be tagged with a different confidence level on a per-tuple basis by exploiting the "dynamic" information collected at runtime—for instance, the data statistics of the uncertain columns in any intermediary and final query results. To be specific, if we could measure that AVG(buffer_time) obtained at ②, although *uncertain*, falls with a given range (say $[21.1, 53.9]$) with a very high probability, we can confidently say that tuples like t_2 (t_3) which have buffer_time far greater (less) than the range are always selected (filtered) with high probability across batches. And thus, we can avoid re-evaluating t_2 and t_3 in batch 2 (and all the following batches), leading to more efficient delta updates.

3.3 Supported Queries

In this paper, we limit the discussion of the iOLAP delta update algorithm to positive relational algebra queries, i.e., any query that can be composed using relational operators SELECT, PROJECT, JOIN[5], UNION, and AGGREGATE. Additionally we do not consider queries that have approximate join/group-by keys under sampling. Generalization to relational algebra including set differences, and queries with approximate join/group-by keys under sampling, are out of the scope of this paper, and would be ideal future directions. iOLAP delta update algorithm relies on efficient error estimation of uncertain columns. We use recent advances in bootstrap-based error estimation, which can be applied to arbitrary user-defined aggregate functions as long as they are Hadamard differentiable, or more intuitively *smooth*[6] under sampling-based approximation, that are common pre-conditions to apply sampling to approximate queries. For instance, MIN/MAX aggregates are not Hadamard differentiable, and thus are usually not approximated through sampling approach.

4. UNCERTAINTY AND DELTA UPDATE

As mentioned earlier, iOLAP treats the changes that could happen in the partial results of any operator in a query as *uncertainties*. In this section, we first categorize these uncertainties that exist in delta processing, and study how they propagate through a query plan in Sec. 4.1. Based on this uncertainty propagation theory, we then develop a new delta update algorithm to compute partial results efficiently in Sec. 4.2. Please note that the uncertainty propagation theory introduced in this section is a slightly conservative version that does not fully utilize statistics collected at runtime, but the delta update algorithm built on top of it already subsumes the existing delta update rules in Figure 1. Finally, in Sec. 4.3, we discuss the ideas for optimizing the uncertainty propagation and the delta update algorithm, by utilizing the statistics collected at runtime (Sec. 5) and leveraging the query lineage (Sec. 6).

4.1 Uncertainty Dichotomy and Propagation

Given an operator in a positive relational query, the difference between the partial result R_i produced by the operator at batch i and the true final query result R of the operator can be characterized by two basic properties: (1) their tuple multiplicities and (2) the attribute values of the tuples in the result. On one hand, some tuples that should be part of the true result (i.e., appear in R) are not included in R_i, while on the other, some tuples that are not part of R appear in the partial result R_i. We refer to this mismatch of tuple multiplicities in the partial result as *tuple uncertainty*. Furthermore, even for same tuples, some of them in R_i can have different attribute values than those in R. We refer to this mismatch in attributes as *attribute uncertainty*. To summarize, there are two types of uncertainties in partial results R_i: (1) the *tuple uncertainty*, and (2) the *attribute uncertainty*. Furthermore, please note that these uncertainties are not just limited to the query result and can exist in the output of all intermediate query operators, while propagating through the query plan.

Next we discuss how each relational operator propagates these two types of uncertainties in the query plan. For each relation R at a particular batch i, we define two types of tagging functions $u_\#$ (for tuple uncertainty) and u_A (for attribute uncertainty) , that map tuples in R to $\{T, F\}$:

- $u_\#$ tags each tuple $t \in R$ with whether the multiplicity of t is uncertain or not, where $u_\#(t) = T$ indicates that the multiplicity of t is uncertain, and $u_\#(t) = F$ when it is not.
- For each attribute A of R, u_A tags each tuple $t \in R$ with whether the value of attribute A in t is uncertain or not, where $u_A(t) = T$ indicates that $t.A$ is uncertain, and $u_A(t) = F$ when it is not.

For any input relation R at the leaf level of a query plan, all its attributes are deterministic, i.e., $u_A(t) = \text{T}$ for any attribute A in R. For the tuple uncertainty, if R is not streamed in, $u_\#(t) = F$. Otherwise, the multiplicity of each tuple t is defined by $s(t; i)$ where $s(\cdot; i)$ is the accumulated sampling function for the i-th batch. $s(\cdot; i) = 1$ indicates that a tuple has been *seen* (i.e., processed) in the first i batches while $s(\cdot; i) = 0$ indicates that it is not. Given that we always process a new batch of data in each iteration, $s(\cdot; \cdot)$ has an property that $s(t; j) \geq s(t; i)$ for $j \geq i$. Thus, if $s(t; i) = 1$ at batch i, we know that the multiplicity of t will not change in subsequent batches and can mark $u_\#(t) = F$. On the other hand, if $s(t; i) = 0$, we mark $u_\#(t) = T$. We denote the attributes referenced in f, which is either a tuple function or an aggregate function, by $attr(f)$. The following rules describe the uncertainty propagation in a query plan[7].

- SELECT: SELECT propagates the attribute uncertainty of its input relation, but could incur tuple uncertainty if the selection predicate is applied on uncertain attributes. That is, if u'_x

[5]By JOIN we mean natural join throughout the paper. Outer joins are not considered part of classical relational algebra, and require set difference to express them.

[6]See [33] by Pol and Jermaine for an intuitive definition of smooth queries.

[7]We use the SQL version of PROJECT and UNION which are without duplicate elimination. Duplicate elimination can be expressed using AGGREGATE, and thus not explicitly discussed here.

are the tagging functions of R, then the tagging functions u_x of $(\sigma_\theta R)$ are defined by

$$\begin{cases} u_A(t) = u'_A(t) \\ u_\#(t) = u'_\#(t) \vee \bigvee_{B \in attr(\theta)} u'_B(t) \end{cases}$$

- PROJECT: PROJECT propagates the tuple uncertainty of its input relation, but could generate uncertain attributes if the projection functions are applied on uncertain attributes. That is, if u'_x are the tagging functions of R, then the tagging functions u_x of $(\pi_{\overline{A_i=\psi_i}} R)$ are defined by

$$\begin{cases} u_A(t) = \bigvee_{B \in attr(\psi)} u'_B(t') \\ u_\#(t) = u'_\#(t') \end{cases}$$

where $t.A_i = \psi_i(t')$ for $\forall i$.

- JOIN: JOIN propagates the attribute uncertainty and tuple uncertainty of its input relations. Specifically, if u^1_x and u^2_x are the tagging functions of R_1 and R_2 respectively, then the tagging functions u_x of $(R_1 \bowtie R_2)$ are defined by

$$\begin{cases} u_A(t) = u^i_A(t_1) \\ u_\#(t) = u^1_\#(t_1) \vee u^2_\#(t_2) \end{cases}$$

where $A \in$ schema U_i of R_i, and $t_j = t$ on schema U_j for $j = 1, 2$.

- UNION: UNION propagates the attribute uncertainty and tuple uncertainty of its input relations. Specifically, if u^1_x and u^2_x are the tagging functions of R_1 and R_2 respectively, then the tagging functions u_x of $(R_1 \cup R_2)$ are defined by

$$\begin{cases} u_A(t) = u^i_A(t) \\ u_\#(t) = u^i_\#(t) \end{cases}$$

where $i = 1$ or 2 and $t \in R_i$.

- AGGREGATE: AGGREGATE is a bit complicated. Both of the attribute uncertainty and tuple uncertainty of its input relation could result in attribute uncertainty in its output; on the contrary, an output tuple has tuple uncertainty only if all input tuples within a group have tuple uncertain. Formally, if u'_x are the tagging functions of R, then the tagging functions u_x of $(\gamma_{\bar{A},\Psi} R)$ are defined by

$$\begin{cases} u_A(t) = \bigvee_{t'=t \text{ on } \bar{A}} u'_A(t') \text{ and} \\ \quad u_\Psi(t) = \bigvee_{t'=t \text{ on } \bar{A}} (\bigvee_{B \in attr(\Psi)} (u'_B(t') \vee u'_\#(t'))) \\ u_\#(t) = \bigwedge_{t'=t \text{ on } \bar{A}} u'_\#(t') \end{cases}$$

As we can see, the attribute uncertainty and the tuple uncertainty can *cause* each other, propagating and interleaving through a query plan. Take the SBI query (Example 1) and its query plan depicted in Figure 2(a) as an example. Figure 3 shows the uncertainty annotations on the outputs of selected operators in the plan. Since the input dataset to AVERAGE ② is streamed in and thus has tuple uncertainty, the AVG(buffer_time) attribute in ②'s output is uncertain. This attribute uncertainty further causes the tuple uncertainty in the output of SELECT ⑤, which subsequently causes attribute uncertainty and tuple uncertainty in AGGREGATE ⑥.

4.2 Delta Update Algorithm

The uncertainty propagation theory identifies subsets of partial results of the query operators that are subject to change across batches. At each batch, all the uncertain values from the previous batch, both attributes and tuple multiplicities, need to be recomputed and brought up-to-date with the new data. Therefore, delta updating a query simply boils down to the problem of updating values with uncertainties in the output relations of the query operators. Before diving into the detailed delta update rules, we first introduce Principle 1 to handle and update the tuple and attribute uncertainties.

	session_id	buffer_time	play_time	#
t_1	$id_1(F)$	$36(F)$	$238(F)$	$1(F)$
t_2	$id_2(F)$	$58(F)$	$135(F)$	$1(F)$
t_3	$id_3(F)$	$17(F)$	$617(F)$	$1(F)$
t_4	$id_4(F)$	$56(F)$	$194(F)$	$0(T)$
...	...	...	...	...

(a) Outputs of ① and ③/ Inputs for ② and ④

AVG(buffer_time)	#
$37(T)$	$1(F)$

(b) Output of ②/ Input for ④

	buffer_time	play_time	AVG(buffer_time)	#
t_1	$36(F)$	$238(F)$	$37(T)$	$1(F)$
t_2	$58(F)$	$135(F)$	$37(T)$	$1(F)$
t_3	$17(F)$	$617(F)$	$37(T)$	$1(F)$

(c) Output of ④/ Input for ⑤

	buffer_time	play_time	AVG(buffer_time)	#
t_1	$36(F)$	$238(F)$	$37(T)$	$0(T)$
t_2	$58(F)$	$135(F)$	$37(T)$	$1(T)$
t_3	$17(F)$	$617(F)$	$37(T)$	$0(T)$

(d) Output of ⑤/ Input for ⑥

AVG(play_time)	#
$135(T)$	$1(T)$

(e) Output of ⑥

Figure 3: Uncertainty annotations of the SBI query (shown in Figure 2(a)) on the outputs of selected operators in batch 1. Note that for ① we show tuples t_4 and so on conceptually (and thus are grayed out) to demonstrate the annotation. Physically these tuples have not been processed and thus have multiplicity = 0 at batch 1.

Principle 1 (Delta Update Principle).

- ***Tuple Uncertainty***. *Tuples with tuple uncertainty may change their participation in the outputs. Therefore, tuple uncertainty should be updated as early as possible—at the first operator that causes this tuple uncertainty—in order to minimize unnecessary computation on the corresponding tuple.*
- ***Attribute Uncertainty***. *Uncertain attributes should be updated as late as possible, only when its value is used. The core intuition is to avoid updating attributes that are never used, and to maximize the reuse of computation on the corresponding tuple.*

As some tuples need to be remembered and re-evaluated in the next batch, we name the collection of tuples that needs to be remembered (and preferably cached) by an operator, as the *state* of that operator. Next, we devise the delta update rules for each relational operator according to Principle 1.

- SELECT. SELECT could incur tuple uncertainty if the predicate is computed on uncertain attributes. Therefore, SELECT saves each input tuple t into its operator state if t has no tuple uncertainty at the input, but has tuple uncertainty in the output. For instance, ⑤ in Figure 2(a) saves all tuples in Table 3(c) in its state after batch 1. All the uncertain attributes in the state are then updated in the next batch.
- PROJECT and UNION. These operators do not change tuple uncertainty. In other words, the operator states for PROJECT and UNION are always $\varnothing$.
- AGGREGATE. AGGREGATE accumulates all the tuples seen so far in order to deliver the correct running results. Thus, AGGREGATE constructs its state by saving the state from the previous batch, along with all input tuples without tuple uncertainty in the current batch. In practice, all the tuples in the state can be compressed into a *sketch state*, which is much more

space-efficient. For instance, to compute the average, in batch 1, ② in Figure 2(a) saves the running sum and running count of all the tuples in Table 3(a) in its state. In general, any aggregate function that can be computed using sub-linear space can maintain the state of `AGGREGATE` space-efficiently using sketches. However, if the aggregated column is uncertain, the state cannot be compressed into a sketch, instead each uncertain attribute need to be updated in the next batch.

- `JOIN`. New incoming tuples from each input relation need to be joined with all the previous tuples in the other input relation to produce the correct join result. Thus, for each side of the join, if the other side have tuples with tuple uncertainty, `JOIN` constructs its state by augmenting its state from the previous batch with all its input tuples in the current batch without tuple uncertainty, which in turn can greatly reduce the intermediate operator state for `JOIN`. For instance, in batch 1, ④ in Figure 2(a) saves Table 3(b) in its state but does not need to save tuples in Table 3(a) in the state since the other side of the join—Table 3(b)—does not have tuple uncertainty. All the uncertain attributes in the state are then updated in the next batch. In practice, many workloads join a large *fact* table with smaller *dimension* table(s). Therefore, if we only sample and stream the larger *fact* table, we only need to keep the smaller *dimension* table in the `JOIN` operator's state.
- `SINK`. We add a virtual operator `SINK` at the end of every query plan. `SINK` reconstructs its from the state from the previous batch, along with all input tuples without tuple uncertainty in the current batch. All the uncertain attributes in the state are updated in the next batch.

One can easily verify that this delta update algorithm subsumes the delta update rules used in previous work as in Figure 1. Some viewlet transformation optimizations proposed in DBToaster [10] can also be applied to our delta update algorithm (as shown in Appendix B. In combination, this conservative version of our delta update algorithm can achieve the same higher-order delta update rules of DBToaster [10].

4.3 Optimizing Delta Update

The uncertainty propagation theory in Section 4.1 only utilizes static information like the query structure and the monotonicity of the sampling process, but does not exploit any dynamic information, e.g., statistics collected at runtime about uncertain attributes (as discussed in Section 3.2). Therefore, it always conservatively tags all the tuples in the same relation indifferently with the same uncertainty property, incurring unnecessary recomputation overhead.

The key idea to optimize the uncertainty propagation, and thus the delta update algorithm, is to introduce an even more fine-grained partitioning of both per-row and per-column uncertainty, where: (1) **per-row**, we partition tuple uncertainty based on how *confident* we are about whether a given tuple is uncertain; (2) **per-column**, we only recompute uncertain attributes by leveraging lineage-based lazy evaluation techniques.

Tuple Uncertainty. The techniques used by `iOLAP` to handle tuple uncertainty are based on an important observation: *The tuple uncertainties, although in the same relation, may not be equally uncertain. By exploiting dynamic information at query time, we can identify tuples that are not likely to change their multiplicities across batches.*

Specifically, we can use data statistics collected at runtime, to estimate the distribution of uncertain attributes and get insights into tuple uncertainty that is caused by attribute uncertainty. This can allow us to carefully partition the tuples with uncertain multiplicities into two parts—the *non-deterministic* and the *near-deterministic* sets. The key intuition is that although tuples in the near-deterministic set has tuple uncertainty, with very high probability they are unlikely to change their multiplicities in the following batches. In contrast, the non-deterministic set of tuples are likely to change their multiplicities. Therefore, we only need to save the tuples in the non-deterministic set in the operator state, that will be recomputed in the next batch. We will explain this optimization in detail in Section 5.

Attribute Uncertainty. As discussed in Section 4.2, uncertain attributes in operator states need to be updated with the latest values in the next batch. The previous delta processing techniques usually interpret value update as deleting the old tuple followed by inserting a tuple. However, this approach is inefficient, as generating a new tuple requires going through the entire plan. The intuition of `iOLAP` is that although the uncertain attributes in a tuple need to be recomputed, many computations involved in generating the tuple can be avoided. Such computations include I/O operation (reading the input tuples from disk), shuffling the tuple according to a deterministic key, and evaluating the deterministic attributes. `iOLAP` achieves this by propagating the lineage information along with each uncertain attribute, and thus enables fine-grained update targeting only the uncertain attributes. This avoids re-generating the whole tuple from scratch, and maximize the reuse of previous computations.

5. TUPLE UNCERTAINTY PARTITION

In this section, we describe the optimization technique to handle tuple uncertainty.

5.1 Discovering Certainty in Uncertainty

As discussed in Section 4.1, the attribute uncertainty is first brought into a query by evaluating an `AGGREGATE` on incomplete data. Due to the blocking nature of the `AGGREGATE` operators, the running results for aggregate functions (on samples of data) are approximate and uncertain. This makes complex OLAP queries with nested aggregates non-monotonic, making simple delta maintenance techniques inefficient. However, there is a key principle behind all S-AQP techniques—running aggregate results will eventually converge to the *true result* (i.e., the aggregate result computed on the full dataset) as the sample size increases. Therefore, as the query engine processes through these batches, the running aggregate results will concentrate in a relatively small range around the true results, and this range will shrink as more and more data is processed. In general, as the attribute uncertainty propagates through a query plan, any uncertain attribute in the outputs of operators in a query plan also shares this convergence property. `iOLAP` leverages this convergence property of uncertain attributes. Before we dive into details of the algorithm, let's first make an observation using an example.

Example 2. *Assume that all the intermediate results of* $AVG(buffer\_time)$ *throughout its online processing are within the range of* 37 ± 16.9 *(Example. 1/Table 2(b)). This implies that across all batches, tuple* t_2 *(with* $buffer\_time = 58$*) will be selected, while tuple* t_3*(with* $buffer\_time = 17$*) will be filtered out. For these tuples, the decisions made by the query engine will never change across all batches. Thus, if we know this fact a priori, we can prune these near-deterministic tuples during online processing.*

From the above example, we can see that by utilizing the convergence property of uncertain attributes, the tuple uncertainty of different tuples can be classified according to how confident we are of the filter decisions we made on the uncertain attributes. And for the decisions we are quite confident of, we can classify those tuples as "not having tuple uncertainty", and thus eliminate the recomputation on them.

Formally, for an uncertain 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 the delta-maintenance algorithm using fine-grained tuple uncertainty by first assuming that these variation ranges are given, and then explain how these variation ranges can be approximated in practice. In batch i, at any predicate $x\ \vartheta\ y$ involving uncertain values,[8] iOLAP classifies the input tuples into two sets: the *non-deterministic* set U_i in which tuples satisfy $\mathfrak{R}(x) \cap \mathfrak{R}(y) \neq \emptyset$, and the *near-deterministic* set C_i in which tuples satisfy $\mathfrak{R}(x) \cap \mathfrak{R}(y) = \emptyset$. For instance, if $\mathfrak{R}(\texttt{AVG(buffer\_time)}) = [21.1, 53.9]$, then $t_2, t_3 \in C_1$, 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 save U_{i-1} in the state; and in batch i, instead of evaluating $Q(D_i)$ from scratch by recomputing both U_{i-1} and C_{i-1}, iOLAP 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, iOLAP approximates the variation ranges using running estimates. Next we will explain how we can approximate the variation range using bootstrap. Note that bootstrap can be substituted with other error estimation methods.

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 $\hat{\mathfrak{R}}(u) = [min(\hat{u}) - \varepsilon \cdot stdev(\hat{u}), max(\hat{u}) + \varepsilon \cdot stdev(\hat{u})]$ to approximate $\mathfrak{R}(u)$, where ε is a slack variable that can be 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 batches, which will result in incorrect query answers. Thus, to check the integrity of $\hat{\mathfrak{R}}(u)$. we keep the history of $\hat{\mathfrak{R}}(u_i)$ at each batch i. At a new batch $(i+1)$, we check the integrity of $\hat{\mathfrak{R}}(u_i)$, or in other words, detect the *failure*, by checking $[min(\hat{u_{i+1}}), max(\hat{u_{i+1}})] \subseteq \hat{\mathfrak{R}}(u_i)$.
- If the check succeeds, $\hat{\mathfrak{R}}(u)$ is updated by setting $\hat{\mathfrak{R}}(u_{i+1}) = [min(\hat{u_{i+1}}) - \varepsilon \cdot stdev(\hat{u_{i+1}}), max(\hat{u_{i+1}}) + \varepsilon \cdot stdev(\hat{u_{i+1}})] \cap \hat{\mathfrak{R}}(u_i)$. If the check fails, we trace up the history of $\hat{\mathfrak{R}}(u_i)$, pick the last batch j where $\hat{\mathfrak{R}}(u_{i+1}) \subseteq \hat{\mathfrak{R}}(u_j)$, and recover the correct query result by recomputing the query on the data from batch $(j+1)$ to i.

Theorem 1. *At batch i, the above algorithm delivers the same query result as $Q(D_i)$, i.e., the result of evaluating a query Q on D_i.*

Proof. We give a brief sketch-proof by induction for Theorem 1. It holds obviously at batch 1. Assume that it holds up to batch i. If $[min(\hat{u_{i+1}}), max(\hat{u_{i+1}})] \subseteq \hat{\mathfrak{R}}(u_i)$ for any x and y in predicate $x\vartheta y$, then $\mathfrak{R}(x_{u_{i+1}}) \cap \mathfrak{R}(y_{u_{i+1}}) \subseteq \mathfrak{R}(x_{u_i}) \cap \mathfrak{R}(y_{u_i})$, which implies that the non-deterministic sets $U_{i+1} \subseteq U_i$. As we will re-evaluate U_i in batch $(i+1)$, we can guarantee to deliver the correct query result as $Q(D_{i+1})$. The proof follows similarly for the case where the integrity check fails and we recover the correct query result. Note that each bootstrap trial can be viewed as evaluating the query on the same set of input tuples with different multiplicities. Thus, the bootstrap estimation is also guaranteed to be correct. □

[8] ϑ is some comparison operator.

[9] We refer interested readers to [8] for the implementation details of bootstrap.

	buffer_time	play_time	AVG(buffer_time)	#
t_1	$36(F)$	$238(F)$	$37, \mathfrak{R} = [21.1, 53.9](T)$	$1(F)$
t_2	$58(F)$	$135(F)$	$37, \mathfrak{R} = [21.1, 53.9](T)$	$1(F)$
t_3	$17(F)$	$617(F)$	$37, \mathfrak{R} = [21.1, 53.9](T)$	$1(F)$

(a) Output of ④

	buffer_time	play_time	AVG(buffer_time)	#
t_1	$36(F)$	$238(F)$	$37, \mathfrak{R} = [21.1, 53.9](T)$	$0(T)$
t_2	$58(F)$	$135(F)$	$37, \mathfrak{R} = [21.1, 53.9](T)$	$1(F)$
t_3	$17(F)$	$617(F)$	$37, \mathfrak{R} = [21.1, 53.9](T)$	$0(F)$

(b) Output of ⑤

AVG(play_time)	#
$135, \mathfrak{R} = \cdots(T)$	$1(F)$

(c) Output of ⑥

Figure 4: Uncertainty annotation of the SBI query (shown in Figure 2(a)) at selected operators in batch 1, using the new uncertainty propagation rules.

The user can also decrease the chance of failure-recover by setting a larger ε (at the cost of increasing the size of the non-deterministic set). In practice, setting ε to $2\times$ the standard deviation of $\hat{u}$ achieves a good balance in controlling the probability of failure-recover and reducing the size of the non-deterministic sets (See Section 8).

5.2 Propagation of Non-Deterministic Sets

In this subsection, we summarize how the new tuple uncertainty propagates through the query plan, and elaborate the delta processing using non-deterministic sets. We modify the tagging functions by introducing per-row partitioning. In general, the new uncertainty propagation rules are the same as those discussed in Section 4.1, but are different for tuple uncertainty in SELECT.

- SELECT. A tuple t in $(\sigma_\theta R)$ does not have tuple uncertainty if tuple t does not have tuple uncertainty in R, and $\theta = true$ for all possible values of uncertain attributes in θ. That is, assuming $\theta = x\ \vartheta\ y$ where ϑ is some comparison operator, and the tuple uncertainty tagging function for relation R is $u'_\#$, the tuple uncertainty tagging function $u_\#$ of $(\sigma_\theta R)$ is defined by

$$u_\#(t) = u'_\#(t) \vee (\mathfrak{R}(x) \cap \mathfrak{R}(y)) \neq \emptyset)$$

Figure 4 demonstrates the uncertainty annotation of the SBI query in batch 1. Note that ④ now annotates AVG(buffer_time) with its variation range $\mathfrak{R}$, which in turn is used by ⑤ to prune the tuple uncertainty of t_2 and t_3, yielding $u_\#(t_2) = u_\#(t_3) = F$ by following our new SELECT rule. As a result, the output of ⑥ is marked without tuple uncertainty, as at least t_2 is contributing to the aggregate result.

Delta Update Rules. The new uncertainty propagation rules significantly optimize the states maintained by operators. By following the new uncertainty annotation, SELECT ⑤ in Figure 2(a) now only saves t_1 (marked in dark shade in Figure 4) in the state, while t_2 and t_3 are pruned from the state because they are not marked with tuple uncertainty; AGGREGATE ⑥ saves the running sum and running count of t_2 (marked in light shade in Figure 4) in its state.

6. LINEAGE AND LAZY EVALUATION

As discussed in Section 4.2, iOLAP requires all uncertain attributes in states of the previous batch to be updated with the latest values in the current batch, in order to correctly compute the delta update. Take the SBI query as an example. During batch 1, t_1 is classified in U_1 and thus saved in the state of ⑤ (as in Figure 5(a)). During batch 2, since ② updates the inner aggregate AVG(buffer_time) to 35.3 (as in Figure 5(b)), t_1 in Figure 5(a) has to be updated to 35.3 in order to evaluate the predicate correctly.

As discussed in Section 4.3, it is obviously a waste to regenerate a tuple from scratch just in order to update a couple of uncertain

attributes. Additionally, since the cached data is just a subset of the running result set, regenerating the tuples from scratch would require random I/O access to the input relations. However, the following example reveals an important observation that can help us avoid such wasteful recomputation.

Example 3. *Consider the SBI query plan shown in Figure 2(a). In the input relation of SELECT ⑤, although attribute AVG(buffer_time) is uncertain, attribute play_time is deterministic. Furthermore, the join relationship between tuples from ② and ③ is also deterministic. Therefore, we should avoid re-evaluating JOIN and re-generating the play_time attribute from scratch.*

The key idea here is to *locally* recompute and update the uncertain attributes, without touching other deterministic attributes. The recomputation is local in the sense that each operator can update its own saved state without referencing other operators.

To achieve this local update algorithm, iOLAP uses two techniques: (1) **Lineage Propagation**. iOLAP propagates with each tuple the information about how its uncertain attributes are computed, i.e., its *lineage*. (2) **Lazy Evaluation**. During delta update, iOLAP updates the saved state by re-evaluating the carried lineage information, without re-generating the whole tuple from scratch. This evaluation is done lazily only when the corresponding uncertain values are accessed. Next, we will discuss these two techniques in detail.

6.1 Lineage Propagation

A natural first step is to decide *the lineage information that needs to be propagated*. Intuitively, we can model the computation used to generate an uncertain attribute u as a lineage function, i.e., $u = f(\bar{x})$. The function definition f and the input parameter $\bar{x}$ are enough to recompute u. Note that f is static and shared by all tuples in the relation, and thus is extracted at compile time (i.e., it does not need to propagated with each tuple). Thus, iOLAP only propagates $\bar{x}$, defined as *Lineage*:

Definition 1 (Lineage). *We define the lineage for attribute A of tuple t output by a query plan P, denoted by $\mathcal{L}_{t.A}()$, inductively as:*

- *Base Relation. The lineage function of a base relation R is defined by $\mathcal{L}_A(t) = \{t.A\}$.*
- *SELECT. If $\mathcal{L}'$ is the lineage function of R, then the lineage function of $\sigma_\theta(R)$ is defined by $\mathcal{L}_A(t) = \mathcal{L}'_A(t)$.*
- *PROJECT. If $\mathcal{L}'$ is the lineage function of R, then the lineage function of $\pi_{\overline{A_i=\psi_i}}(R)$ is defined by $\mathcal{L}_A(t) = \bigcup_{B_i \in attr(\psi_i)} \mathcal{L}'_{B_i}(t')$ where $t.A_i = \psi_i(t')$ for $\forall i$.*
- *JOIN $R_1 \bowtie R_2$. If $\mathcal{L}^1$ and $\mathcal{L}^2$ are the lineage functions of R_1 and R_2 respectively, then the lineage function of $(R_1 \bowtie R_2)$ is defined by $\mathcal{L}_A(t) = \mathcal{L}^i_A(t_i)$ where $A \in$ schema U_i of R_i and $t_j = t$ on schema U_j for $j = 1, 2$.*
- *UNION. If $\mathcal{L}^1$ and $\mathcal{L}^2$ are the lineage functions of R_1 and R_2 respectively, then the lineage function of $(R_1 \cup R_2)$ is defined by $\mathcal{L}_A(t) = \mathcal{L}^i_A(t_i)$ where $i = 1$ or 2 and $t \in R_i$.*
- *AGGREGATE. If $\mathcal{L}'$ is the lineage function of R, then the lineage function of $\gamma_{\bar{A},\Psi}(R)$ is defined by $\mathcal{L}_A(t) = \bigcup_{t'=t \text{ on } \bar{A}} \mathcal{L}'_A(t')$ and $\mathcal{L}_\Psi(t) = \bigcup_{t'=t \text{ on } \bar{A}} \bigcup_{B \in attr(\Psi)} \mathcal{L}'_B(t')$.*

Block-wise Lineage. Note that since aggregates are computed from a large 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 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 2(a) can be divided into two lineage blocks: $\{①, ②\}$, $\{③, ④, ⑤, ⑥\}$. iOLAP propagates lineage within each lineage block, while simply propagating the aggregate results along with their corresponding group-by keys between lineage blocks, thus bounding the overall cost of lineage propagation. Formally, Definition 1 is modified as:

- AGGREGATE. If $\mathcal{L}'$ is the lineage function of R, then the lineage function of $\gamma_{\bar{A},\Psi}(R)$ is defined by $\mathcal{L}_A(t) = \{(rel(\gamma), t.key)\}$, where $rel(\gamma)$ is a unique reference to the output relation of $\gamma_{\bar{A},\Psi}(R)$ and $t.key$ is the group-by key of t.

	buffer_time	play_time	AVG(buffer_time)	#
t_1	$36(F)$	$238(F)$	$37, \mathfrak{R} = \cdots (T)$	$o(T)$

(a) State of ⑤ at Batch 1

AVG(buffer_time)	#
$35.3, \mathfrak{R}' = \cdots (T)$	$1(F)$

(b) Output of ② at Batch 2

	buffer_time	play_time	AVG(buffer_time)	#
t_1	$36(F)$	$238(F)$	$37, \mathfrak{R} = \cdots, \mathcal{L} = \{(②, \text{key=\_})\} (T)$	$o(T)$

(c) State of ⑤ at Batch 1 with Lineage

Figure 5: Lineage propagation and lazy evaluation of iOLAP.

Figure 5(c) shows the state of ⑤ with lineage. We can see that AVG(buffer_time) in t_1 is annotated with the lineage "computed from the aggregate of group (key=_) in the output of ②".

Folding deterministic value. We can fold deterministic part of the lineage function into a single attribute, and therefore reduce of the size of the lineage propagated with each tuple. Here *deterministic* means that the subexpressions do not involve uncertain attributes and thus remain unchanged across batches. For instance, in $sum(x) + y + 3$, we can fold $y + 3$ into a single expression z, and propagate z instead of $y + 3$.

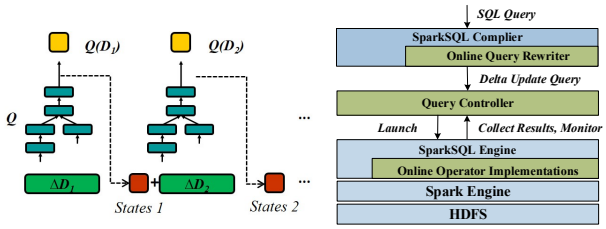
6.2 Lazy Evaluation

It is straightforward to lazily evaluate an uncertain attribute if its propagated lineage does not involve aggregates. However, it is less obvious when the lineage involves aggregates. As we only propagate a reference to the aggregate results, along with their group-by keys across lineage block boundaries, we need to join the up-to-date aggregate results with the propagated lineage information in order to update the corresponding uncertain values. For example, to update AVG(buffer_time) of t_1 (as in Figure 5(c)) at batch 2, one needs to join it with the output of ② in Figure 5(b) to retrieve the latest AVG(buffer_time). Formally, if the lineage of t is defined as $\mathcal{L}_A(t) = \{(rel, key)\}$, lazy evaluation of t requires natural joining t with the relation rel on column key and projecting onto column $rel.A$, i.e., $\Pi_{rel.A}(\{t\} \bowtie_{key} rel)$. Joining t and rel is non-trivial and often requires shipping data, because t and rel are usually distributed according to different shuffle keys, especially in a distributed SQL engine like SparkSQL. However, in practice the aggregate relation rel is usually very small, and it is often very efficient to broadcast-join t and rel by broadcasting rel to all the machines. Finally, we conclude this section with an important observation. Note that this update process is modeled as a join query, and is thus able to be optimized using the underlying SQL optimization framework by choosing the most cost-effective way to deliver the lineage information.

7. IMPLEMENTATION

In this section, we present the implementation details of iOLAP.

Mini-Batch Execution Model. We use a mini-batch execution model to implement iOLAP. As shown in Figure 6(a), the mini-



(a) (b)

Figure 6: (a) and (b) The mini-batch execution model and system architecture of iOLAP.

Workload	Batch Size	# of Tuples Per Batch
TPC-H (lineorder)	11.5GB	86, 000, 000
TPC-H (partsupp)	7.5GB	80, 000, 000
TPC-H (customer)	2.5GB	15, 000, 000
Conviva	25.6GB	10, 600, 000

Table 1: Batch sizes used for the relations that are streamed in

batch model treats the iterative query processing of iOLAP as a series of short and independent jobs. At each batch, the job takes the new incoming data and the states from the previous batch as input, and produces the query results and a set of states for the next batch. The mini-batch execution model can easily utilize the distributed computation resources, handle fault tolerance and stragglers, and better integrate with existing batch-oriented database systems. Last but not least, it is worth noting that iOLAP can be implemented using other execution models—such as the streaming operator model commonly used in existing streaming and online aggregation systems [26, 32, 6, 22, 30, 35].

System Architecture. We have implemented iOLAP in SparkSQL [4], a distributed OLAP engine built on top of Spark. iOLAP is open-sourced [2]. Figure 6(b) depicts the system architecture of iOLAP. iOLAP extends SparkSQL in three modules:

(1) *Online Query Rewriter*. The online query rewriter rewrites the query into a delta update query, which when plugged with different mini-batches of data, turns into a series of mini-batch queries. This rewriting includes (1) adding columns for bootstrap and lineage; (2) replacing operators with their online counterparts; (3) modifying plans to support lazy evaluation. We have a detailed discussion about query rewriting in Appendix C. The online query rewriter simply consists of a set of plan rewriting rules. It can be easily integrated into a relational query compiler.

(2) *Online Operator Implementations*. We modify the SparkSQL engine with implementations of online operator. Compared to the standard relational operators, these online operators can store and load states as in Section 4.2.

(3) *Query Controller*. The query controller partitions the input data into mini-batches, schedules the delta update query on each mini-batch and collects query results. The controller also monitors the correctness of all the variation ranges, and schedules recomputing jobs to recover the query result when a failure is detected. The query controller is implemented as a thin user application in Spark driver.

8. EXPERIMENTAL STUDY

In this section, we evaluate the effectiveness and efficiency of iOLAP. All the experiments are performed on a EC2 cluster of 20 r3.2xlarge machines (each with 8 vCPU[10], 61GB of RAM and 160GB SSD). All the data is stored on S3. Our experiments are conducted on both synthetic and real-world workloads:

- A synthetic 1TB dataset from the TPC-H benchmark [5]. As real-life large scale OLAP usually use denormalized relation schema to avoid expensive distributed joins, we project the TPC-H relation onto a schema similar to the SSB benchmark. Specifically, we join table lineitem and orders into a single relation lineorder, but keep other relations unchanged. We choose a subset of the TPC-H benchmark queries which include all the queries with nested subqueries structures (Q11, Q17, Q18, Q20, Q22), and a representative subset of the rest which are all simple SPJA queries. We used the same TPC-H queries as in [39].
- A 2TB subset of a 17TB anonymized real-world video content distribution workload from Conviva Inc. [1], comprising of a de-normalized fact table. We compose a query workload based on the real analysis used in [29, 20] on the same dataset, which involves simple SPJA queries (C3, C5, C11, C12), complex queries with nested subqueries and HAVING clauses (C1, C2, C4, C6, C7, C8, C9, C10), UDF (C6, C7) and UDAF (C8, C9, C10). The queries with nested subqueries are similar to those in the TPC-H benchmark.

During the experiments, we always stream in the fact table or the largest table (lineorder, partsupp or customer in TPC-H) used in the queries, and use bootstrap with 100 trials for error estimation. Unless specified, we use the batch sizes as shown in Table 1, and the default slack parameter = 2.0 for iOLAP throughout the experiments. We compare the performance of iOLAP with existing online processing techniques in OLA [26] and incremental view maintenance work [10, 25, 31], specifically DBToaster [10], which is the state-of-the-art delta processing algorithm. We implement the higher-order delta update algorithm of DBToaster (referred to as HDA) without code generation and indexes on SparkSQL, as code generation and indexes are outside the scope of this paper.

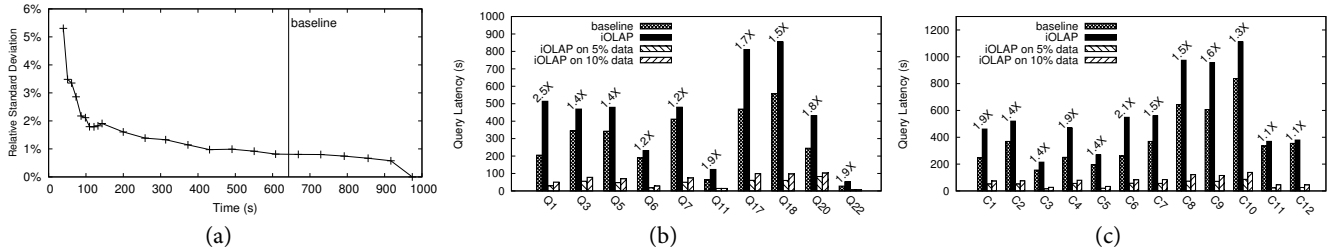
8.1 End-to-End Performance

In this section, we evaluate the effectiveness of incremental OLAP interface for interactive analysis by comparing iOLAP with the batch processing model of a traditional OLAP engine (named the *baseline*), i.e., answering the query on the original dataset using unmodified SparkSQL. The results are shown in Figure 7(a), 7(b) and 7(c).

Figure 7(a) demonstrates a typical query processing in iOLAP using C8 from the Conviva workload. As one can see, any traditional query engine will only be able to deliver an answer after processing the entire dataset, which in this case, would incur 10.7 minutes latency (marked by the vertical bar). On the other hand, iOLAP can deliver an approximate answer in 39.4 secs (i.e., only in about 6.1% of the whole query time). Furthermore, iOLAP continuously refines the answer at a very user-friendly pace of roughly every 10 seconds. It is worth noting that while iOLAP incurs an additional 50% overhead in processing the whole dataset as compared to the baseline (primarily due to the error estimation overheads and scheduling multiple mini-batch jobs), 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 1.6 minutes, which is almost 7× faster than a batched execution.

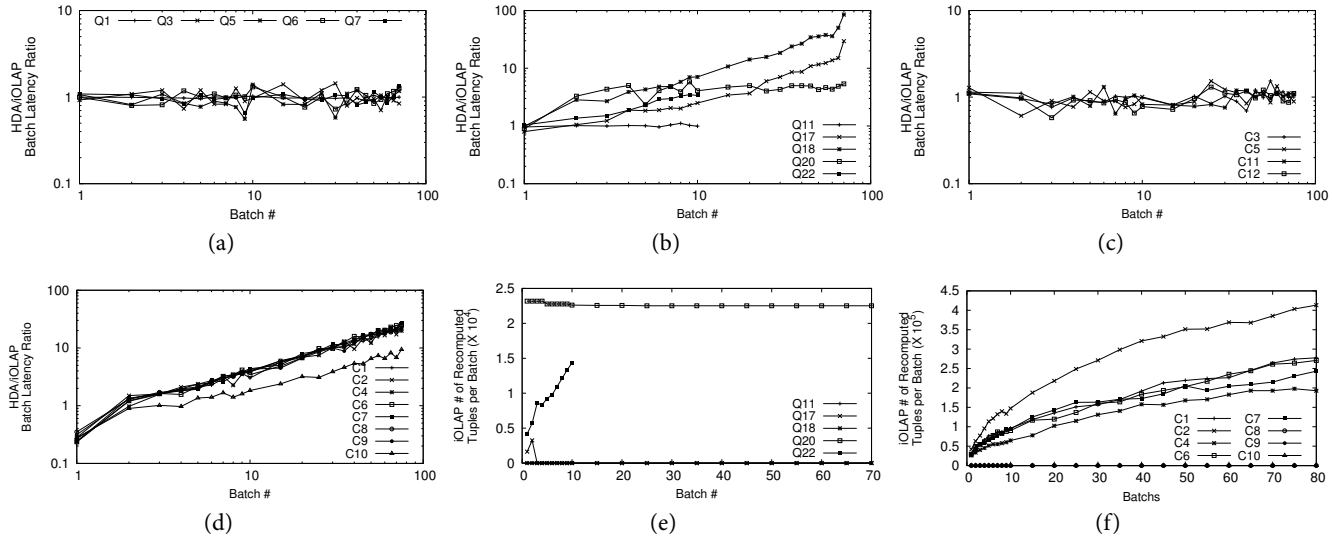
Figure 7(b) and 7(c) plot more results on all the TPC-H and Conviva queries. For the sake of presentation, we only plot (1) the query time of the baseline to process all the data (denoted by *baseline*), (2) the query time of iOLAP to process all the data (denoted by *iOLAP*), (3) the query time of iOLAP to process a 5% sample (denoted by *iOLAP on* 5% *data*), and (4) the query time of iOLAP to process a 10% sample (denoted by *iOLAP on* 10% *data*). For clarity, we also mark the relative ratio between *iOLAP* and *baseline* in the figures.

[10] Each vCPU is a hyperthread of an high frequency Intel Xeon E5-2670 Processor.



(a) (b) (c)

Figure 7: (a) The relative standard deviation vs. query time curve delivered by iOLAP on Conviva C8. We plot the values for the first 10 mini-batches, and then for brevity, every 5 mini-batches after that. (b) and (c) The query times of the baseline, iOLAP to deliver approximate results on 5% and 10% samples, and the iOLAP to process all the data for TPC-H and Conviva workloads respectively.



(a) (b) (c)

(d) (e) (f)

Figure 8: (a)-(d) The ratio of query times of iOLAP and HDA in the first 10 batches, and for brevity every 5 mini-batches after that, for TPC-H ((a) and (b)) and Conviva ((c) and (d)) workloads respectively. (e) and (f) The number of tuples recomputed by iOLAP in the first 10 batches, and for brevity every 5 mini-batches after that, for TPC-H and Conviva workloads respectively.

When processing the whole dataset, iOLAP incurs 60% overhead on average, and at most cases the overhead is below 100%. Again, these overheads are primarily due to the use of bootstrap for error estimation and the scheduling cost of multiple mini-batch jobs. Moreover, at query time, especially in a distributed environment, the majority of the query latency is spent on shuffling data, computing joins, etc., compared to which the relative overhead incurred by iOLAP is small. It is also worth noting that iOLAP is able to deliver approximate answers much faster than the baseline, e.g., if an approximate answer computed on a 10% sample is all that is needed, iOLAP usually takes only 10% to 20% of the baseline latency.

8.2 Delta Processing of iOLAP

In this subsection, we study the performance of the delta maintenance algorithms used by iOLAP for incremental query processing. We compare iOLAP with HDA. In this section, we focus on breakdown experiments that shed light on the performance benefits brought by iOLAP. We have more experiments showing end-to-end performance comparison between iOLAP and HDA in Appendix D.

Delta Update Latency. We compare the query latency per batch used by iOLAP and HDA. We plot the ratio of the query times spent by HDA and iOLAP for each batch in Figure 8(a)-8(d). For clarity, we plot both the X and Y axes using log scale. We can observe that the trends of the per-batch latency ratios in general fall into two classes: For simple SPJA queries (shown in Figure 8(a) and 8(c)), the performance of iOLAP and HDA is comparable, this clearly demonstrates that the delta processing techniques of iOLAP boil down to the classical delta processing techniques, as discussed at the beginning of Section 5. On the other hand, for complex queries with nested subqueries (shown in Figure 8(b) and 8(d)), iOLAP significantly outperforms the classical delta-maintenance algorithms. Specifically, iOLAP performs slightly slower than HDA in the first batch (because iOLAP need extra work on caching uncertainty sets, caching input relations for joins etc. as discussed in Section 5.2), but is much faster than HDA in the following batches. Furthermore, the per-batch query latency ratio grows linearly in general, which shows that the performance of HDA degrades with more data processed; on the contrary, iOLAP achieves almost constant query time for each batch. This is because that in the classical algorithms, every update on an inner aggregate subquery causes the engine to recompute the outer query on the entire data that was previously processed, while iOLAP could effectively limit the recomputation needed in each batch. Unlike other queries, the curves of TPC-H Q11 and Q20 flatten out. This is because their outer queries join two aggregate subqueries which are small in size, and thus are less expensive to be recomputed. Even though, iOLAP outperforms HDA in batch latency for Q20, as it avoids most of the recomputation of the outer query.

Number of Tuples Recomputed. We further look into the delta processing by studying the number of tuples recomputed in each mini-batch in iOLAP. Figure 8(e) and 8(f) depict the number of tuples recomputed in each batch. We only plot the curves for complex queries with nested subqueries as simple SPJA queries do not have recomputation. Recall that we use the default batch sizes in Table 1. The numbers of tuples recomputed per batch shown in the figures are almost negligible compared to the average number of incoming tuples per batch (some of which are even 0), which incurs very lit-

tle recomputation overhead. It is also worth noting that for almost all the queries, the number of tuples recomputed grows sub-linearly across batches, which demonstrates the efficiency of the tuple uncertainty partitioning technique in iOLAP.

Optimization Breakdown. To better understand the delta update algorithms used in iOLAP, we conducted an experiment to study the effectiveness of the two major optimizations: (1) tuple uncertainty partitioning, denoted by OPT1, and (2) lineage propagation and lazy evaluation, denoted by OPT2. We gradually turned off the optimizations until fall back to HDA. We show the breakdown results of C2 from Conviva as an example in Figure 9(a). As we can see, the tuple uncertainty partitioning limits the recomputation to the non-deterministic sets, which reduces the query latency of each mini-batch to almost 25% of that of HDA. However, we still need to recompute the non-deterministic set from scratch. The lineage propagation and lazy evaluation maximize reusing of the computation for the tuples that need to be recomputed, bringing down the query latency per batch by another 18%.

8.3 Memory Utilization of iOLAP

In this section, we study the memory utilization of iOLAP. Due to space limit, we only show the experiment results on the TPC-H workload. The results on the Conviva workload are similar and are available in Appendix D.

We plot the memory overhead caused by keeping states for operators in Figure 9(b). Since JOIN only save states in the first batch (due to the fact that all joins in the experiments are between a streamed fact table and dimension tables) but other operators save states every batch, we show the states of JOIN and other operators separately. As shown, iOLAP only need to keep states of a few hundreds MBs for most queries. Exceptions are Q3, Q5, Q7, Q11, Q18 and Q20, which have a large JOIN states (5-50GB) because they have many joins due to a snowflake schema. However, it is worth noting that these JOIN states are already optimized to only keep the small dimension tables (see Section 4.2), as shown by the fact that the JOIN states are much smaller than the total amount of data shipped by the baseline. Moreover, the JOIN states in memory can always be spilled to disk.

We also study the data footprint overhead of bootstrap and lineage propagation of iOLAP. These techniques require expanding the intermediate query results with extra columns, incurring a larger data footprints than the baseline. As iOLAP is built on SparkSQL, which uses a pipeline implementation, these data footprints are eventually reflected on the data shipped across network (shuffled or broadcasted) at query time. Figure 9(c) compare the data shipped by the baseline and iOLAP. As different operator has different data footprint overhead (e.g., the overhead can go up to 100× for AGGREGATE, but much smaller for other operators), we divide the queries into two categories: (1) Queries that only ship AGGREGATE results (e.g., Q1, Q6, Q20, Q22), which only ship a small amount (< 10GB) of data (2) Queries that ship results of a mixed operators (e.g., Q3, Q5, Q7, Q11, Q17, Q18), which ship > 10GB data in our experiments. This classification can be verified by the data shipped by the baseline. As shown, iOLAP-Total has a small overhead for both categories compared to the baseline (100MB-9GB for (1) and 23%-45% for (2)). Additionally the data footprint of iOLAP-Per-Batch is 1-2 orders of magnitude smaller than that of the baseline, which implies that the user would shuffle much less data if she stops the query early.

8.4 Parameter Tuning

We next study tuning the parameters of iOLAP, i.e., the slack parameter and batch sizes in practice. We show the experiment results on Conviva. The results on TPC-H are similar and can be found in Appendix D.

Slack Parameter. As discussed in Section 5, the approximate algorithm to estimate the variation ranges has a tunable knob—the slack parameter, that directly impacts (1) the probability of failure-recovery happening at query time, and (2) the size of the non-deterministic set. In short, a larger slack parameter will decrease the probability of failure-recovery, but increase the size of the non-deterministic set and thus increase the recomputation per batch, and vice versa. Therefore, the user needs to tune this parameter to make the optimal trade-off in practice. We study the impact of this slack parameter by varying it from 0 to 2.5, and measuring the probability of failure-recovery and the size of the non-deterministic set. The results are shown Figure 9(d). Interestingly, we find out that setting a slightly bigger slack can significantly reduce the probability of failure-recovery. With the slack increasing, the probability of failure-recovery quickly goes to 0. For instance, setting slack = 0.5 reduces the failure probability by $7 - 8\times$ on average compared to the extreme case (slack = 0); when slack = 2.0, all queries have no failure-recovery. On the other hand, increasing the slack parameter can increase the size of the non-deterministic set. But in practice, even a very small sample contains enough data to make the variation range concentrate in a very small range. Thus, the non-deterministic set does not increase much. The results are shown in Figure 9(e). All in all, we find out that slack = 2.0 leads to a good trade-off in practice.

Batch Size. Another tunable knob of iOLAP is the batch size. In practice, the batch size depends on two factors: (1) how interactive the user wants to get updates on the query results, and (2) how much latency of the whole query processing the user can tolerate. Smaller batch sizes will reduce the query latency spent in each mini-batch, but increase the number of mini-batches and thus increase the overhead of scheduling more mini-batch jobs. Figure 9(f) and 9(g) show how per-batch latency and overall latency change with varying the batch size. Clearly, with a larger batch size, the per-batch latency increases linearly, but the overall latency decreases linearly.

9. RELATED WORK

Online Aggregation. Online aggregation [26] and its successors [19, 32] 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 [7, 9, 12, 17, 34] focus on constructing the optimal samples to improve query accuracy. STRAT [17], SciBORQ [34], Babcock et al. [12] and AQUA [7] 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 [9] 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. iOLAP relies on simulation-based bootstrap techniques used by BlinkDB to estimate the variation ranges of uncertainty attributes. Therefore, modulo the stratified sampling building phase, a first iteration of iOLAP is the same as BlinkDB. Different from BlinkDB-like AQP systems that provides near-instantaneous approximate result output, iOLAP can provide progressively approximate results over time. iOLAP does not require preparing a fixed sized sample beforehand, which simplifies the design of AQP systems. Additionally, given a query and a user-defined error bound, BlinkDB relies on an estimation module to pick a sample size. If the answer on this sample isnÕt accurate, or it doesnÕt

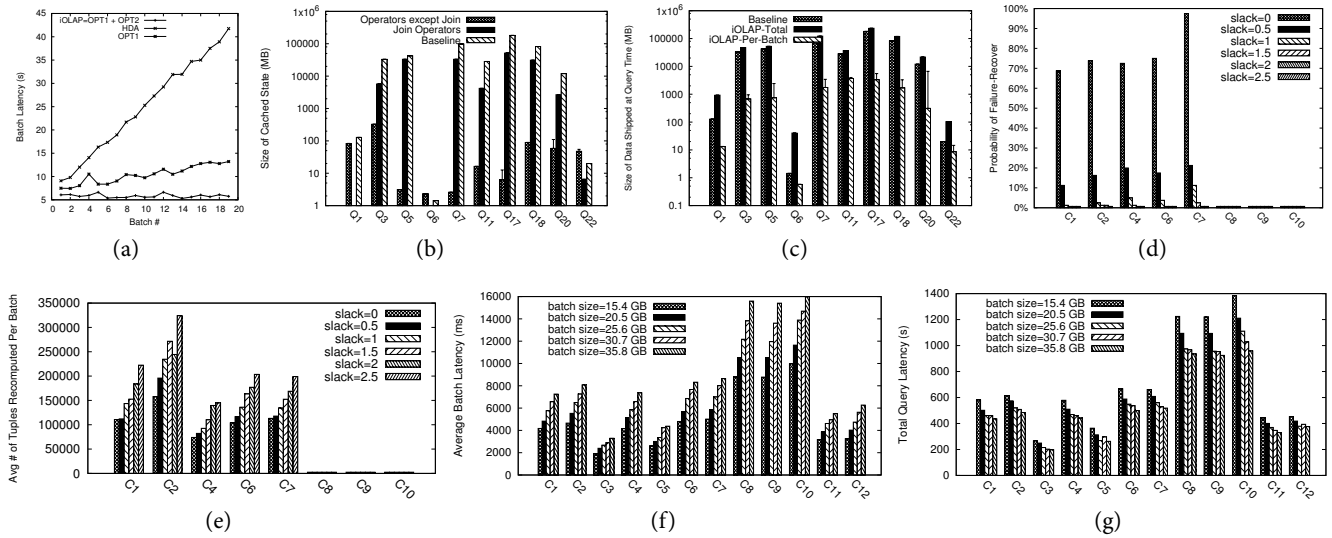


Figure 9: (a) Breakdown results of the delta processing optimization techniques used in iOLAP for C2. (b) The state sizes saved by iOLAP for TPC-H. We plot the all-batch total state sizes across all batches for JOIN, and the average and max (as error bars) per-batch state sizes for other operators. We also plot the data size shuffled/broadcasted by the baseline as a reference. (c) The size of data shipped by the baseline and iOLAP for TPC-H. We plot the all-batch total data size (iOLAP-Total), and the average and max (as error bars) per-batch data size (iOLAP-Per-Batch). (d)-(g) The relationship of the slack parameter vs. the probability of failure-recovery, the slack parameter vs. the size of non-deterministic set, the batch size vs. the query latency per mini-batch, and the batch size vs. the total query latency on Conviva workload respectively.

satisfies the userÕs requirements, BlinkDB then has to rerun the full query from scratch on a larger sample size. On the contrary, iOLAP just needs to compute a delta update query on the new delta input. ABM [39] is an analytical bootstrap method which is much faster than simulation-based bootstrap. The analytical bootstrap work is orthogonal to iOLAP. iOLAP can use the analytical bootstrap proposed in analytical bootstrap instead of simulation-based bootstrap to estimate the variation ranges, achieving better performance.

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* [13, 14] and *bag* [18, 25] semantics. [23, 24] proposed a $\mathbb{Z}$-relation model for annotating relations for incremental view maintenance. iOLAP utilizes this relational model to model the propagation of uncertainties in a relational query. Computing the delta update query has been studied for query with aggregates [25, 31] and temporal aggregates [37]. However, majority of work in this area only focuses on simple SPJA queries without nested and correlated aggregation subqueries. More recently, DBToaster [10] 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. iOLAP's delta-maintenance technique doesn't have this limitation. [27] models incremental view maintenance as a sample clean problem, and use the view on a cleaned sample to infer the view on the whole dataset. While [27] focuses more on how to maintain a statistically sound sample for a view definition, and how to maintain the view approximately, iOLAP focus on how to exactly maintain the result of a query given uniform sample updates.

Data Stream Processing. Data stream processing [6, 22, 30, 35] 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. More recent works on stream processing [28, 15, 16] use a progressive model to model incremental processing. Tuples are augmented with progressive intervals, and operators are aware of progressive intervals. This model can automatically achieve the delta update rules in Figure 1, but also share the same performance limitation on complex queries beyond SPJA queries. In particular, [16] also uses the progressive model to model different sampling schemes, but still relies on the user to code the data flow query in order to deliver meaningful result under sampling, e.g., scaling a SUM aggregate according to the sampling rate. In contrast, iOLAP provides a meaningful semantics for uniform sampling by default without user intervention, and can be extended to incorporate stratified sampling. There has also been work on one-pass streaming algorithms [11, 36] 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. iOLAP on the other hand provides automatic incremental processing to general SQL queries, including those with multiple levels of nesting and arbitrary aggregates.

10. CONCLUSION

This paper presented iOLAP, an incremental OLAP query engine that uses a mini-batch execution for interactive incremental query processing. iOLAP uses a novel delta update algorithm that is built on top of an uncertainty propagation theory. We experimentally validated the effectiveness and efficiency of iOLAP. The delta processing techniques of iOLAP could benefit other related fields in general, e.g., delta view maintenance and streaming systems.

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.

11. REFERENCES

[1] Conviva Inc. http://www.conviva.com/.

[2] iOLAP Github repository. https://github.com/amplab/iolap.

[3] Knowledge Management. http://www.globalgraphics.com/technology/knowledge-management/.

[4] Spark and SparkSQL. http://spark.apache.org/.

[5] TPC-H Benchmark. http://www.tpc.org/tpch/.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

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

[15] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB*, 8(4):401–412, 2014.

[16] B. Chandramouli, J. Goldstein, and A. Quamar. Scalable progressive analytics on big data in the cloud. *PVLDB*, 6(14):1726–1737, 2013.

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

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

[19] 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.

[20] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. A. Joseph, A. Ganjam, J. Zhan, and H. Zhang. Understanding the impact of video quality on user engagement. In *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*, pages 362–373, 2011.

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

[22] 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.

[23] T. J. Green, Z. G. Ives, and V. Tannen. Reconcilable differences. *Theory Comput. Syst.*, 49(2):460–488, 2011.

[24] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, pages 31–40, 2007.

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

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

[27] S. Krishnan, J. Wang, M. J. Franklin, K. Goldberg, and T. Kraska. Stale view cleaning: Getting fresh answers from stale materialized views. *PVLDB*, 8(12):1370–1381, 2015.

[28] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order processing: a new architecture for high-performance stream systems. *PVLDB*, 1(1):274–288, 2008.

[29] X. Liu, F. Dobrian, H. Milner, J. Jiang, V. Sekar, I. Stoica, and H. Zhang. A case for a coordinated internet video control plane. In *ACM SIGCOMM 2012 Conference, SIGCOMM '12, Helsinki, Finland - August 13 - 17, 2012*, pages 359–370, 2012.

[30] 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.

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

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

[33] A. Pol and C. Jermaine. Relational confidence bounds are easy with the bootstrap. In *SIGMOD*, pages 587–598, 2005.

[34] 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.

[35] 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.

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

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

[38] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica. G-OLA: generalized on-line aggregation for interactive analysis on big data. In *SIGMOD*, pages 913–918, 2015.

[39] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo. The analytical bootstrap: A new method for fast error estimation in approximate query processing. In *SIGMOD*, pages 277–288, 2014.

APPENDIX

A. RELATIONAL ALGEBRA WITH BAG SEMANTICS

In this paper, we generalize multiset relations to tuple multiplicities that are real numbers. Formally, a multiset relation R maps all tuples with schema U (we denote this set of tuples as U-Tup) to tuple multiplicities, i.e., $R : U\text{-Tup} \to \mathbb{R}$. That is, $R(t)$ represents the multiplicity of tuple t in relation R. We focus on positive relational algebra, i.e., queries that can be composed using operators `SELECT`, `PROJECT`, `JOIN`, `UNION` and `AGGREGATE`. Note that we use the SQL version of `PROJECT` and `UNION` which are without duplicate elimination. Duplicate elimination can be expressed using `AGGREGATE`, and thus not explicitly discussed here.

- `SELECT`: If $R : U\text{-Tup} \to \mathbb{R}$ and the selection predicate θ maps each tuple to either 0 or 1, then $\sigma_\theta R : U\text{-Tup} \to \mathbb{R}$ is defined by
$$(\sigma_\theta R)(t) = R(t) \cdot \theta(t)$$
- `PROJECT`: If $R : U\text{-Tup} \to \mathbb{R}$ and ψ_i is a tuple function, then $\pi_{\overline{A_i=\psi_i}} R : U\text{-Tup} \to \mathbb{R}$ is defined by
$$(\pi_{\bar{A}} R)(t) = R(t')$$
where $t.A_i = \psi_i(t')$ for $\forall i$.
- `JOIN`: If $R_i : U_i\text{-Tup} \to \mathbb{R}$ for $i = 1, 2$, then $R_1 \bowtie R_2 : U\text{-Tup} \to \mathbb{R}$ is defined by
$$(R_1 \bowtie R_2)(t) = R_1(t_1) \cdot R_2(t_2)$$
where $t_i = t$ on schema U_i for $i = 1, 2$.
- `UNION`: If $R_1, R_2 : U\text{-Tup} \to \mathbb{R}$, then $R_1 \cup R_2 : U\text{-Tup} \to \mathbb{R}$ is

defined by

$$(R_1 \cup R_2)(t) = R_i(t)$$

where $i = 1$ or 2 and $t \in R_i$.

- AGGREGATE: If $R : U\text{-Tup} \to \mathbb{R}$ and Ψ is an aggregate function, then $\gamma_{\bar{A},\Psi} R : U\text{-Tup} \to \mathbb{R}$ is defined by

$$(\gamma_{\bar{A},\Psi} R)(t) = \sum_{t'=t \text{ on } \bar{A}} R(t')$$

B. OPTIMIZING DELTA UPDATE STATES

The optimizations of viewlet transformation proposed in DBToaster [10] can be translated into a set of query rewriting rules, By applying these query rewriting rules, we can rewrite a query into an equivalent query, which can reduce the size of the states maintained by our delta update algorithm proposed in Section 4.2. Using the query rewriting rules in combination with our delta update rules, we can achieve the same higher-order view maintenance ideas as in DBToaster. Below are the equivalent query rewriting rules of the viewlet transformation rules as listed in Figure 2 of [10].

- **Query Decomposition**

$$\gamma_{\bar{A}\bar{B},\Psi=sum(f_1\times f_2)}(Q_1 \bowtie Q_2) = \gamma_{\bar{A}\bar{B},\Psi_3=sum(\Psi_1\times\Psi_2)}(\gamma_{\bar{A},\Psi_1=sum(f_1)}(Q_1) \bowtie \gamma_{\bar{B},\Psi_2=sum(f_2)}(Q_2)) \quad (1)$$

where Q_1 and Q_2 have no common columns; $\bar{A}$ and $\bar{B}$ are the group-by terms of each.
This optimization rule pushes group-by AGGREGATE below JOIN, which can help reduce the state of JOIN. For instance, assuming Q_2 has tuple uncertainties, before rewriting, we need to save all tuples from Q_1 without tuple uncertainty in the state, which is of size $|Q_1|$; in contrast, after rewriting, the state is reduced to all tuples from $\gamma_{\bar{A},\Psi_1=sum(f_1)}(Q_1)$ without tuple uncertainty, whose size is the number of distinct $\bar{A}$ in Q_1.

- **Factorization and Polynomial Expansion**

$$(Q \bowtie Q_1) \cup (Q \bowtie Q_2) \cup \cdots = Q \bowtie (Q_1 \cup Q_2 \cup \cdots) \quad (2)$$

This optimization rule can pull common subexpressions out of UNION. When $Q_1, Q_2, \cdots$ have tuple uncertainties, instead of keep Q as the state for each JOIN in the left hand size of Equation 2, we only need to save Q as the state for the only JOIN in the right hand side.

- **Input Variables**

$$\gamma_{\bar{A},\Psi=sum(f(\overline{\bar{B}\bar{C}}))}(\sigma_{\theta(\overline{\bar{B}\bar{C}})}(Q)) = \gamma_{\bar{A},\Psi_2=sum(f(\bar{B}\bar{C}))}(\sigma_{\theta(\bar{B}\bar{C})}(\gamma_{\bar{A}\bar{B},\Psi_1=sum(1)}(Q))) \quad (3)$$

where f, θ are functions over columns; $\bar{B}$ is the columns in Q used by f, θ; $\bar{C}$ is a column that do not appear in Q.
This optimization rule push AGGREGATE below SELECT, which can reduce the state of SELECT. If the predicate θ causes tuple uncertainty, this rewriting rules can compress multiple tuples in the state that are from Q and with the same $\bar{A}\bar{B}$ columns into a single tuple from $\gamma_{\bar{A}\bar{B},\Psi_1=sum(1)}(Q)$.

- **Nested Aggregates and Decorrelation**

$$\gamma_{\bar{A},\Psi=sum(f)}(\sigma_{\theta(\bar{B},\bar{C})}(Q_O \bowtie Q_N)) = \gamma_{\bar{A},\Psi_2=sum(1)}(\sigma_{\theta(\bar{B},\bar{C})}(\gamma_{\bar{A}\bar{B},\Psi_1=sum(f)}(Q_O) \bowtie Q_N)) \quad (4)$$

where Q_N is a nested non-grouping aggregate subquery; f, θ are functions over columns; $\bar{A}$ is columns from Q_O; $\bar{B}$ is the columns in Q_O used by f, θ; $\bar{C}$ is the columns in Q_N used by θ; Q_O and Q_N have no common columns.
This optimization rule works similarly to the input variable rule above, by pushing AGGREGATE below SELECT. This rule reduces the state of SELECT by compressing multiple tuples in the state which are from $Q_O \bowtie Q_N$ and with the same $\bar{A}\bar{B}$ columns into a single tuple from $\gamma_{\bar{A}\bar{B},\Psi_1=sum(f)}(Q_O) \bowtie Q_N$.

Next we show an example to demonstrate how these rewriting rules work.

Example 4. *Given two relations R (of two columns AB) and S (of two columns CD), and query*

```
SELECT SUM(A * D)
FROM R, S
WHERE B = C
```

Assume that both R and S are streamed in, and thus we need to keep all the tuples seen so far from both R and S as the state of JOIN. By applying Query Decomposition rule, we can rewrite the above query into $\gamma_{AB,\Psi_3(\Psi_1\times\Psi_2)}(\gamma_{B,\Psi_1=sum(A)}(R) \bowtie_{B=C} \gamma_{C,\Psi_2=sum(D)}(S))$. After the rewriting, for JOIN, we only need to save $\gamma_{B,\Psi_1=sum(A)}(R)$ and $\gamma_{C,\Psi_2=sum(D)}(S))$ as the state, which is equivalent to the higher order view $\Delta_R(S)$ and $\Delta_S(R)$ in [10]'s terminology.

C. IMPLEMENTATION OF IOLAP

Given a query, iOLAP automatically rewrites it into a delta query. The delta query is a normal SQL query, but enhancing the original query with error estimation, uncertainty tagging, lineage propagation, and lazy evaluation. The rewriting consists of 4 steps:

1. At compile time, the compiler analyzes the query plan, tagging each attribute (including the multiplicity column) with its corresponding uncertainty by following the propagation rules in Section 4.1.
2. After that, we add bootstrap into the query plan to support error estimation. We use a poissonized bootstrap implementation proposed in [8]. Specifically, we insert columns representing bootstrap-generated multiplicities after scanning the streamed relations. These multiplicity columns take random values from a $Poisson(1)$ distribution, and are propagated downstream. Consequently, all the downstream AGGREGATE operators are modified to use these multiplicity columns, and thus all the uncertain attributes throughout the plan are duplicated to multiple instances, each one corresponding to one bootstrap trial.
3. After modifying the plan for bootstrap, we further add uncertainty propagation and lineage propagation support into the plan. For each operator that may produce tuple uncertainty, we insert a new column in its output to keep the tuple uncertainty, and propagate it to downstream operators. For each operator whose output has uncertain attributes, we insert the lineages of the uncertain attributes as columns into its output, and propagate them through the plan.
4. The compiler then replaces the operators in the plan with their online counterparts, and inserts join and project operators to implement lazy evaluation as described in Section 6.

D. MORE EXPERIMENTS

D.1 End-to-End Performance

We study the performance of iOLAP and HDA by comparing the query latency used by them to process (1) all the data, (2) a 5% sample, and (3) a 10% sample. The results on TPC-H and Conviva are de-

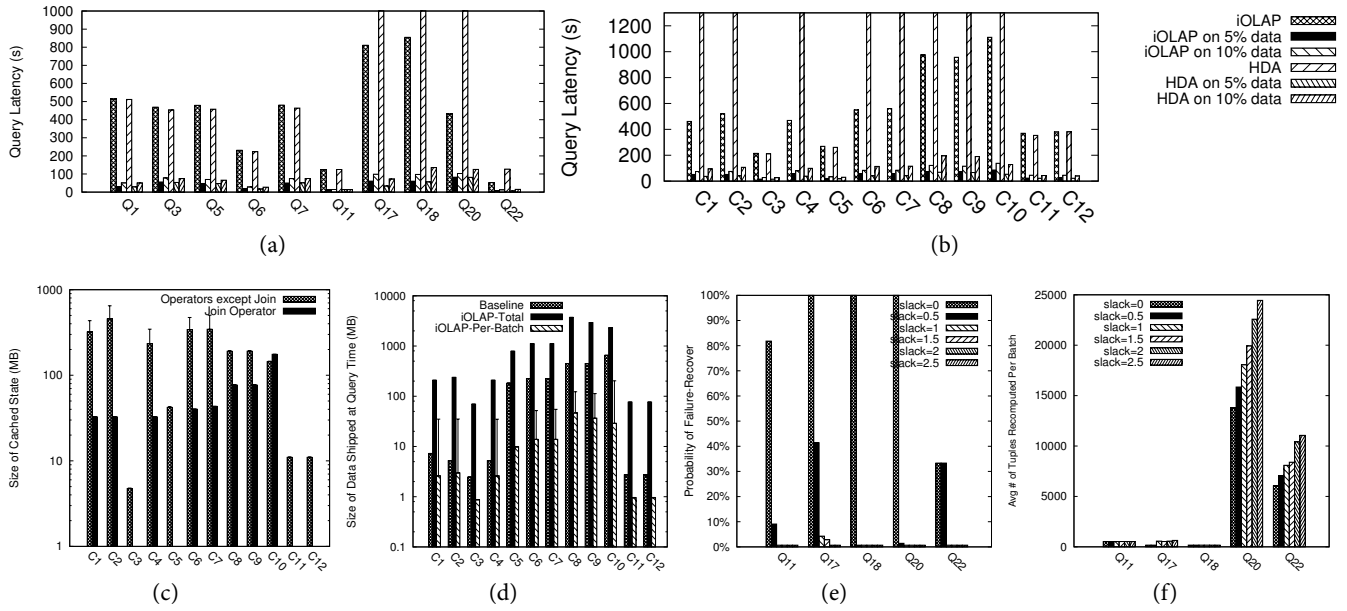


Figure 10: (a) and (b) The query times of iOLAP and HDA to deliver approximate results on all the data, as well as 5% and 10% samples for TPC-H and Conviva workloads respectively. (c) The state sizes saved by iOLAP for Conviva. We plot the all-batch total state sizes across all batches for JOIN, and the average and max (as error bars) per-batch state sizes for other operators. (d) The size of data shipped by the baseline and iOLAP for Conviva. We plot the all-batch total data size (iOLAP-Total), and the average and max (as error bars) per-batch data size (iOLAP-Per-Batch). (e) and (f) The relationship of the slack parameter vs. the probability of failure-recover, and the slack parameter vs. the size of non-deterministic set on TPC-H respectively.

picted in Figure 10(a) and 10(b) respectively. As we can see, for simple SPJA queries, such as TPC-H Q1, Q3, Q5, Q6, Q7, Q11, Q22, and Conviva C3, C5, C11, C12, iOLAP and HDA have comparable performance. For complex queries with nested subqueries, HDA's overhead quickly accumulate up: The query latencies of HDA on 10% samples have already exceeded those of iOLAP. When processing all the data, HDA took much longer time than iOLAP. Actually most of the complex queries did not finish in a reasonable time, and the bars get cut off in the figures.

D.2 Memory Utilization of iOLAP

We study the memory overhead caused by keeping states for various operators on Conviva. The results are plotted in Figure 10(c), which shows that all the operators, including JOIN, keep only a few hundreds of MBs states across all iterations.

We study the data footprint overhead of the bootstrap and lineage propagation of iOLAP on Conviva, as shown in Figure 10(d). Because the Conviva dataset has a single fact table, all Conviva queries fall into the category of queries that only ship AGGREGATE results . It is clear that iOLAP-Total has a < 5GB overhead compared to the baseline. And the data footprint of iOLAP-Per-Batch is 1-2 orders of magnitude smaller than that of the baseline.

D.3 Parameter Tuning

We experiment with different slack parameter settings to see how it impact the probability of failure-recovery and the size of non-deterministic set on the TPC-H workload. We plot the results in Figure 10(e) and 10(f). Similar to the results on Conviva, with the slack increasing, the probability of failure-recovery quickly decrease to 0; in contrast, the size of non-deterministic sets increases slowly.