A Data Cleaning Approach for Approximately Up-To-Date Results on Stale Materialized Views

ABSTRACT

Materialized views, stored pre-computed query results, are widely used to optimize queries on large datasets. When base tables are updated, however, materialized views can become out-of-date and any queries issued to these views will give *stale* results. While various techniques have been proposed to make view maintenance run faster, they may still be very costly, especially in the Big Data era when new data is coming at an increasingly fast rate. In this work, we look at this problem from a different perspective. We treat staleness as data errors and model the view-maintenance problem as a data-cleaning problem. Instead of maintaining a materialized view, we aim to correct the query result on the stale view directly. To achieve this goal, we define the concept of an *update pattern* and propose efficient techniques to maintain a sample of update patterns for three types of materialized views. Using the sample update patterns, we infer how the updates affect a query result on the stale view. For common aggregate queries (`sum`, `count`, and `avg`), we can derive a correction factor from the sample to correct the stale query result. We bound our corrections in analytic confidence intervals, and prove that our technique is optimal with respect to estimate variance. As sampling can be sensitive to long-tailed distributions, we further explore an outlier indexing technique to give increased accuracy when the data distributions are skewed. We evaluate our method on real and synthetic datasets and in both a single node (MySQL) and distributed environment (Apache Spark). In one large scale experiment using 20-node Apache Spark cluster, we derived a 700 million row view from a 1TB activity log base dataset. We found that maintaining a 5% sample was 16.3x faster than full maintenance yet only with mean query error of 1.2%.

1. INTRODUCTION

Materialized views, stored pre-computed query results, are used to optimize query processing on large datasets [6,15, 16]. Materialization and related concepts such as selecting which queries to materialize have been well studied in recent research [4,20,25,32]. This research has further expanded beyond the SQL setting [22] and has shown promising results when applied to numerical linear algebra and machine learning. However, when derived from frequently changing tables, materialized views face the obvious challenge of *staleness* where the pre-computed results need to be updated to reflect the changes in the base tables. To avoid expensive recalculation, incrementally updating materialized views, also called incremental maintenance, has been well studied [6,15].

Unfortunately, in many desired applications, incremental maintaining the view for every incoming update can be very costly. When the base tables change rapidly, the database system has to propagate each update to all of the materialized views. Consequently, there are many techniques that defer updating the views to a later time or trigger the updates only when necessary [6,34]. Deferral allows for many advantages such as batching updates together to amortize overheads and scheduling updates at times when there are more system resources available eg. at night. Deferral strategies provide increased flexibility to meet the resource constraints of the system, but may not guarantee that the views will always be up-to-date. When a materialized view is out-of-date during a maintenance gap, queries issued to the view can have *stale* results.

In this work, we address the incremental maintenance problem from a different perspective. We explore whether refreshing stale rows in a materialized view can be modeled as data cleaning problem and whether stale query results can be "cleaned". While much of the data cleaning literature focuses on improving query accuracy on dirty datasets, there has been an increasing trend of considering the costs of data cleaning [31]. In particular, recent results show that to answer aggregate queries, such as SUM, COUNT, and AVG, on dirty datasets, it suffices to clean a small, representative sample of records. This model raises a new possibility for materialized views, namely, that a sample of up-to-date rows can be used to answer aggregate queries without having the maintain the entire materialized view. In our proposed approach, given a stale query result, we sample updates to the materialized view, and we use the sample to estimate a correction to stale query results thus "cleaning" the result. While approximate, the corrected query result can be bounded within confidence intervals. Existing techniques allow the user to control the freshness of queries by choosing maintenance parameters (e.g. nightly maintenance vs. hourly maintenance) based on prior experience, however, without bounds on the results, a burst of updates can lead to unexpected changes in query accuracy. On the other hand, our approach gives results that are, in expectation, fresh and the user controls the tightness of the bound with the sampling ratio.

Our approach has three components: (1) sampling, (2) correction, and (3) outlier indexing. In (1), we sample the updates to the view in a way that ensures that the sample is representative of the up-to-date data. Updates can affect different classes of views differently, and we define a logical unit called an "update pattern" which represents how an update affects the derived view. Our system maintains a sample of the "update patterns" and uses this sample to derive a correction. (2) we use the sampled update pattern to derive a correct aggregate queries on stale materialized views. From the sample, we estimate how much the updates affect the query in question and we use this estimate to adjust the

query result when applied to a stale view. Finally, in (3) sampling has the potential to mask outliers, and in fact, it is known heavy-tailed distributions are poorly approximated from samples [5]. In this framework, we utilize a technique called outlier indexing [5], which guarantees that rows in the materialized view derived from an "outlier" record (one that has abnormal attribute values) is contained in the sample. Coupling outlier detection with sampling has an interesting implication; not only can we answer exact selection queries on the outliers, but the information from the outliers can potentially improve query accuracy or likewise reduce the number of needed samples.

Our approach can be implemented with a relatively small overhead: at maintenance time the generation of random numbers to build the sample, and at query execution time single pass over a small sample of data to estimate a correction for the query. Consequently, sampling can significantly save on maintenance costs and give a flexible tradeoff between accuracy and performance. To summarize, our contributions are as follows:

- We model the incremental maintenance problem as a data cleaning problem and propose a technique to approximately clean stale results; providing a flexible tradeoff between accuracy and performance.
- We use an outlier index to increase the accuracy of the approach for power-law, long-tailed, and skewed distributions.
- We analyze the costs of our approach and show analytically and empirically that it can result in dramatic reductions in view maintenance time.
- We evaluate our approach on real and synthetic datasets and in both the single node and distributed environments.

The paper is organized in the following way. In Section 2, we introduce materialized views and discuss the current maintenance challenges. Next, in Section 3, we give a brief overview of our overall system architecture. In Section 4 and 5, we describe the mathematics and query processing of our technique. In Section 6, we describe the outlier indexing framework. Finally in Section 7, we evaluate our approach.

2. BACKGROUND

2.1 Incremental Maintenance Main Ideas

Incremental maintenance of materialized views is well studied, see [6] for a survey of the approaches. Most incremental maintenance algorithms consists of two steps: calculating a "delta" view, and "refreshing" the materialized view with the delta. More formally, given a base relation T, a set of updates U, and a view $\mathbf{V}_T$:

Calculate the Delta View- In this step, we apply the view definition to the updates and we call the intermediate result a "delta" view:

$$\Delta\mathbf{V}$$

This is also called a *change propagation formula* in some literature, especially on algebraic representations of incremental view maintenance.

Refresh View- Given the "delta" view, we merge the results with the existing view:

$$\mathbf{V}_T^{'} = refresh(\mathbf{V}_T, \Delta\mathbf{V})$$

The details of the refresh operation depend on the view definition. Refer to [?] for details.

2.2 Maintenance Strategies

There are two principle types of maintenance strategies: immediate and deferred. In immediate maintenance, as soon as the base table is updated, the change is propagated to any derived materialized view. Immediate maintenance has an advantage that materialized view is always up-to-date, however it can be very expensive. This scheduling strategy places a bottleneck when records are written reducing the available write-throughput for the database. Such an approach lacks flexibility since system resources need to be provisioned for the worst-case influx of updates. Furthermore, especially in a distributed setting, record-by-record maintenance cannot take advantage of the benefits of consolidating communication overheads by batching.

To address these challenges, deferred maintenance is alternative solution. The main idea of deferral is to avoid maintaining the view immediately and schedule an update at a more convenient time either pre-set or adaptively. In deferred maintenance, the user often accepts some degree of staleness in the materialized view for additional flexibility for scheduling refresh operations. For example, a user can update the materialized view nightly during times of low-activity in the system. During these times, the system can take use more resources to process the updates without worrying about the affect on the throughput. However, this also means that during the day the materialized view becomes increasingly stale as it was computed the night before.

More sophisticated deferred scheduling schemes are also possible. In particular, we highlight a technique called lazy maintenance which defers maintenance until such time a query on the view requires a row to be up-to-date [34]. While it ensures that query results are never stale, lazy maintenance potentially shifts much of the computational cost to query execution time.

Immediate maintenance introduces a bottleneck on updates to the base table and lazy maintenance introduces a bottleneck during query execution, and rapid updates can make these approaches impractical. As a consequence, periodic maintenance or recalculation is often the most feasible solution. While the user can set the maintenance period to control the expected staleness of the data, in general, the relative error between a stale query result and the up-to-date result is unbounded.

2.3 Data Cleaning

Much of data cleaning research focuses on improving query accuracy on dirty datasets. For example, designing rules or algorithms to remove or correct erroneous records [28]. However, recent work has considered the costs of data cleaning and how to budget effort. The SampleClean project presents a query processing framework that cleans on a sample of data, and then bounds the results of aggregate queries on dirty datasets [31].

This process allows the user to control the accuracy of the query with the sampling ratio as opposed to querying a dirty dataset with an unknown level of data error. One of the algorithms in SampleClean, NormalizedSC, takes a dirty dataset and query and using a sample of clean data learns how to correct a query so the result is "clean". Since the "correction" is derived from a sample, the correction is approximate but it turns out it bounded in error for `sum`, `count`, and `avg` queries. Inspired by this solution, we apply a data cleaning cleaning approach to materialized views where we model staleness as a type of data error and budget our cleaning effort to get an approximate correction for aggregate queries (`sum`, `count`, and `avg`).

2.4 SAQP

Estimating the results of aggregate queries from samples has been well studied in a field called Sample-based Approximate Query Processing (SAQP). Our approach differs from SAQP as we look to approximately correct a query rather than directly estimating the query result. In other words, we use a sample of up-to-date data to understand how to compensate for the staleness. The SAQP approach to this problem, would be to estimate the result directly from the maintained sample; a sort of SAQP scheme on the sample of the view [18]. We found that estimating a correction and leveraging an existing deterministic result lead to lower variance results on real datasets (see Section 7).

3. SYSTEM OVERVIEW

Frequent incremental maintenance of materialized views can be challenging given system resource constraints. Existing materialized view maintenance techniques lie on a spectrum of accuracy and performance. Immediate maintenance guarantees that query results on the view will always be accurate, while batch maintenance allows for a higher throughput of updates to the base table. These two techniques sit at the extremes of the spectrum since they still require that all updates are processed and propagated to the view.

Modeling this problem as a data cleaning problem gives us a new perspective for addressing staleness. When updates arrive, rows in the materialized view may be either out-of-date or missing altogether; making the view "dirty". The challenge is to clean the materialized view by updating the out-of-date rows and inserting the missing rows, however this can be very expensive if there are a large number of updates. Suppose, we cleaned one dirty row at a time processing only as much of the updates as necessary to clean the row. Then, for queries on the view, we get progressively less stale results for a lesser cost. With this intuition in mind, to ensure that our result is unbiased, we clean a random sample of dirty rows. For aggregate queries, such as `sum`, `count`, and `avg`, we can then estimate from this sample how the updates change the query results. From this estimate, we can derive a correction to stale query results.

In implementation, our proposed system will work in conjunction with existing maintenance or re-calculation approaches. We envision the scenario where materialized views are being refreshed periodically eg. nightly. While maintaining the entire view throughout the day may be infeasible, sampling allows the database to scale maintenance with the performance and resource constraints during the day. Then, between maintenance periods, we can provide approximately up-to-date query results for aggregation queries.

An additional challenge is that sampling has the potential mask outliers in the updates. We address this problem by using an "outlier index" which has been applied in the SAQP setting [5].

3.1 System Architecture

The architecture of our proposed solution is shown in Figure 3.1. The top of the diagram resembles a typical materialized view maintenance architecture. However, there are a couple of key additions: (1) as updates arrive we sample the "update pattern". (2) we maintain an outlier index, and (3) we combine the sample and the outlier index to process queries on the stale view. In this section, we will introduce these three components, which are also explained in much more detail in the following sections.

3.1.1 Supported Materialized Views

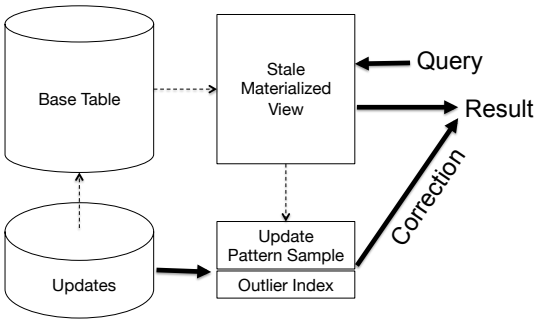

Figure 1: We clean a stale query result by deriving a correction from a sample of up-to-date data. To make this process robust to skewed datasets, we couple sampling with an outlier indexing approach. For aggregation queries, we can guarantee that our results are unbiased and bounded.{{What's the difference between dotted lines and solid lines? Add stale result and update-to-data result into the figure?}}

We will first introduce the taxonomy of materialized views that can benefit from our approach. In this paper, we analyze and experiment with three classes of views: Select-Project Views, Foreign-key Join Views, and Aggregation Views. Our technique can be applied to a broader class of views, please refer to [?] for a full description.

Select-Project Views (SPView): One type of view that we consider are views generated from Select-Project expressions of the following form:

```
SELECT [col1,col2,...]
FROM table
WHERE condition([col1,col2,...])
```

Foreign-Key Join Views (FJView): As an extension to the Select-Project Views, we can support views derived from a Foreign-Key join:

```
SELECT table1.[col1,col2,...],
table2.[col1,col2,...]
FROM table1, table2
WHERE table1.fk = table2.fk
AND condition([col1,col2,...])
```

Aggregation Views (AggView): We also consider views defined by group-by aggregation queries of the following form:

```
SELECT [f1(col1),f2(col2),...]
FROM table
WHERE condition([col1,col2,...])
GROUP BY [col1,col2,...]
```

3.1.2 Sampling the Update Pattern

Given these materialized views, the first challenge is sampling. We have to sample the updates in a particular way so the sample accurately represents how updates affect queries on the view. The three classes of views require different sampling techniques. For example, insertions to the base database only result in insertions to Select-Project views. But, for Aggregation Views, insertions to the base table can also result to updates to existing stale rows. In Section 4, we describe the sampling algorithm and a cost analysis of how much sampling can reduce maintenance costs.

3.1.3 Correcting a Query

Once we have an appropriate sample, we can use information from the sample to correct stale query results. Suppose, we issue an aggregate query to the stale view. Then, we scan our sample, and calculate an approximate correction. The corrected result is in expectation up-to-date and is probabilistically bounded. Like the sampling, the algorithm to calculate the correction varies between the types of views. We detail query correction in Section 5.

3.1.4 Outlier Indexing

We are often interested in records that outliers, which we define in this work as records with abnormally large attribute values. Outliers and power-law distributions are a common property in web-scale datasets. Often the queries of interest involve the outlier records, however sampling does have the potential to mask outliers in the updates. If we have a small sampling ratio, more likely than not, outliers will be missed.

Therefore, we propose coupling sampling with outlier indexing. That is, we guarantee that records (or rows in the view derived from those records) with abnormally large attribute values are included in the sample. What is particularly interesting is that these records give information about the distribution and can be used to reduce variance in our estimates. See Section for details on this component.

3.1.5 Example Application: Log Analysis

To illustrate our approach, we use the following running example which is a simplified schema of one of our experimental datasets (Figure ?). Imagine, we are querying logs from a video streaming company. These logs record visits from users as they happen and grow over time. We have two tables: Log and Video, with the following schema:

```
Log(sessionId, videoId, responseTime, userAgent)
Video(videoId, title, duration)
```

These tables are related with a foreign-key relationship between Log and Video, and there is an integrity constraint that every log record must link to one video in the video table.

Log

sessionId	videoId	responseTime	userAgent
1	21242	23	Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0)
1	8799	44	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_6_8)
2	191	303	ROBOT (bingbot/2.0; +http://www.bing.com/bingbot.htm)

Video

videoId	title	duration
21242	NBA Finals 2012	2:30:04
8799	Bill Gates Interview	0:30:02
191	Cooking Channel	0:22:45

Figure 2: A simplified log analysis example dataset. In this dataset, there are two tables: a fact table representing video views and a dimension table representing the videos.

Consider the following example materialized view AggView, which stores a result for each video and the maximum time it took for the server to load that video:

```
SELECT videoId,
max(responseTime) AS maxResponseTime
FROM Log
GROUP BY videoId;
```

Suppose, the user creates and materializes this view. The user wants to know how many videos had a max response time of greater than 100ms.

```
SELECT COUNT(1)
FROM AggView
WHERE maxResponseTime > 100
```

Let us suppose the query result is 15. {{The example made the point, but it may need to be polished.}} Since the user queried the table, there have been new logs inserted into to the Log table. So materialized aggregation view and the old result of 15 are now stale. For example, if our sampling ratio 5%, our system maintains a sample of this view. That means for 5% of the videos (distinct videoID's) we refresh stale maxResponseTime if necessary. From this sample, we calculate how many new videos changed from a maxResponseTime of less than 100ms to times greater than 100ms; let us suppose this answer is 2. Since our sampling ratio is 5%, we extrapolate that 10 new videos throughout the view should now be included in the count, resulting in the estimate of 25. In contrast, if we had applied SAQP, we would have counted how many videos in the sample had a max response time of greater than 100ms.

(a-1) An example of SPView

```
CREATE VIEW SPView AS
SELECT sessionId, videoId, responseTime
FROM Log
WHERE userAgent LIKE "%Mozilla%"
```

(a-2) Update Patterns for the SPView

	sessionId	videoId	responseTime
Insert r1:	1	214	18 ms
Insert r2:	2	2056	15 ms

(b-1) An example of FJView

```
CREATE VIEW FJView AS
SELECT sessionId, videoId, responseTime,
       duration
FROM Log, Video
WHERE Log.videoId = Video.videoId
```

(b-2) Update Patterns for the FJView

	sessionId	videoId	response Time	duration
Insert r1:	45	567	12 ms	0:34:23
Insert r2:	5566	20	20 ms	1:00:08

(c-1) An example of AggView

```
CREATE VIEW AggView AS
SELECT videoId, max(responseTime) as
       maxResponseTime
FROM Log
GROUP BY videoId
```

(c-2) Update Patterns for the AggView

	videoId	maxResponse Time (old)	maxResponse Time (new)
Insert r1:	5	NULL	15690 ms
Update r2:	1	1298 ms	6298 ms
Keep r3:	65	798 ms	798 ms

Figure 3: Illustration of the update patterns for SPView and FJView and AggView.

4. SAMPLING UPDATE PATTERNS

Updates can affect the three classes of materialized views in different ways, and our sample should represent how the update will affect the materialized view. We describe this process as sampling the "update pattern". In this section, we discuss how to efficiently sample update patterns for different types of views. We first formalize the update patterns in Section 4.1, and then present scalable sampling techniques for them in Section 4.2. Furthermore, we also analyze the cost of our methods, and show how sampling can lead to significant costs savings in incremental view maintenance.

4.1 Update Patterns

An *update pattern* represents how an update to a base table affects a materialized view derived from that table. We define this information to be such that if we took a 100% sample the information would be sufficient to transform the stale view to the corresponding up-to-date view exactly. In

our model, there are three possible update patterns. (1) Insert a new record into a view; (2) Update attribute values of an existing record; (3) Keep an existing record unchanged. Figure 3 illustrates the update patterns of the three types of views supported by our system.

For SPView, the only possible updates to the view are inserting new records. For instance, Figure 3(a-1) shows an example of SPView. When the base table Log is updated, there might be some new records whose userAgent field contains "Mozilla", which should be added into the SPView accordingly. Figure 3(a-2) illustrates the update patterns for the SPView. The figure means that in order to obtain the up-to-date SPView, we should insert r_1 and r_2 into the original SPView.

For FJView, similar to SPView, its update patterns only involve insertion. For instance, Figure 3(b-1) shows an example of FJView. When the base tables, Log or Video are updated, there might be some new records that satisfy the join condition, which should be added into the FJView accordingly. Figure 3(b-2) illustrates the update patterns for the FJView. The figure means that we should insert r_1 and r_2 into the original FJView to obtain the up-to-date FJView.

For AggView, the changes of the view could be inserting a new group, updating the aggregation attribute values of an existing group, or keep an existing group unchanged, thus its update patterns involve insertion, updating or no changing. For instance, Figure 3(c-1) shows an example of AggView and Figure 3(c-2) illustrates the update patterns for the AggView. In the figure, "Insert r_1" indicates that the group-by key, "videoId = 5", does not exist in the original view, and we need to insert this new group into the view; "Update r_2" indicates that the group-by key, "videoId = 1", is already in the original view, and we should update the aggregation attribute value (i.e., maxResponseTime) of the group from 1298 ms to 6298 ms; "Keep r_3" indicates that the group-by key, "videoId = 65", is already in the original view, and we do not need to change the aggregation attribute value of the group.

4.2 Sampling Techniques for Update Patterns

Modeling updates to the base table in terms of update patterns does not change the expense of incremental view maintenance. It does, however, give us a logical unit to sample and in this section, we describe how our system only maintains a random sample of the update patterns. As our earlier analysis of the update patterns suggests, our sampling techniques for each type of materialized view are slightly different.

4.2.1 Select-Project and Foreign-Key Join Views

Consider a base table T and the corresponding insertion table $\triangle T$. Given a SPView defined on the table, its full update patterns involve the inserted records of $\triangle T$ that satisfy the predicate of the SPView. To sample the update patterns (e.g., at a sampling ratio of $\rho = 5\%$), we randomly select 5% records from $\triangle T$, and then keep the records that satisfy the view's predicate as a sample.

For a given FJView, let T be the fact table w.r.t the FJView, and $T_1, T_2, \cdots, T_k$ be the dimension tables w.r.t the FJView. The full update patterns of the FJView involve the inserted records of $\triangle T \bowtie (T_1 + \triangle T_1) \bowtie (T_1 + \triangle T_1) \cdots \bowtie (T_k + \triangle T_k)$ that satisfy the predicate of the FJView[1]. Similar to SPView, to sample the update patterns (e.g., $\rho = 5\%$), we first randomly select 5% records from $\triangle T$, and then join the selected records with the up-to-date dimension tables (i.e., $T_i + \triangle T_i$ $(1 \le i \le k)$), and finally return as a sample the joined records that satisfy the FJView's predicate.

[1]Due to foreign-key integrity constraints, it is guaranteed that the join result of $T \bowtie \triangle T_i$ $(1 \le i \le k)$ is empty.

Cost Analysis. Let $n = |\triangle T|$ be the number of inserted records, v be the cardinality of the stale view, δ_v be the cardinality of the delta view, and ρ be the sampling ratio. Table 1 compares the cost of our update-pattern sampling techniques (denoted by Sampling) with the corresponding cost of incremental view maintenance (denoted by Maintenance). The table does not include the I/O cost of loading the updates into memory since in both Sampling and Maintenance we only need to load the updates once and amortize that I/O cost over all views.

- Delta View. For SPView and FJView, incremental view maintenance has the processing cost of $\text{cost}_{pred}(n)$ and $\text{cost}_{join}(n)$, respectively, since they have to evaluate the predicate or the join for all n inserted records. Our sampling approach can reduce this number to $\rho \cdot n$ as we only need to evaluate this on our sample. The additional overhead of sampling is to generate n random numbers, i.e., $\text{cost}_{rand}(n)$.
- Refresh. For SPView and FJView, incremental view maintenance has to insert δ_v rows into the original view while we have to insert only $\rho \cdot \delta_v$ records into the sample of update patterns.

From the above analysis, we can see that for SPView and FJView, when the predicate or the join is expensive to evaluate, our sampling techniques require much less cost than incremental view maintenance. For example, if the time of generating n random numbers is negligible, we can save the cost of incremental view maintenance by 20 times for the sampling ratio of $\rho = 5\%$.

4.2.2 Aggregation View

The update patterns for an AggView contains the update information for each existing group in the view as well as newly inserted groups. To sample the update patterns, we seek to maintain the update patterns for a sample of groups. That is, for the existing group keys in the AggView, we keep track of the update patterns for a sample of them; for the newly inserted group keys, we choose a sample of them to maintain.

To implement this idea, we incorporate hashing into our sampling techniques. The basic idea is similar to hash partitioning, where a sample can be thought as a data partition. Specifically, we define a hash function, hashfunc$(\cdot)$, which takes a single group key as an input and outputs a uniform random number in the range of [0, 1]. We only maintain the update patterns for the group keys such that

$$\text{hashfunc}(\text{"group key"}) \le \rho. \quad (1)$$

Obviously, the maintained group keys are a random sample of all the group keys with the sampling ratio of ρ. With this hash-based sampling technique, we can easily devise an efficient update-pattern sampling approach for an AggView.

To better understand the approach, we first introduce how to obtain the full update patterns of an AggView. The idea is the same as incremental view maintenance, which first computes a delta AggView by executing the AggView definition on an insertion table and then obtains the full update patterns by refreshing the AggView with the delta AggView. Similarly, to sample the update patterns, we first compute a delta *sample* AggView by executing the AggView definition on the records of an insertion table that satisfy hashfunc("group key") $\le$ ρ, and then obtains the *sample*

Table 1: Cost comparison between incremental view maintenance and update-pattern sampling.

	SPView		FJView		AggView	
	Maintenance	**Sampling**	**Maintenance**	**Sampling**	**Maintenance**	**Sampling**
Delta View	$\text{cost}_{pred}(n)$	$\text{cost}_{rand}(n)$ + $\text{cost}_{pred}(\rho \cdot n)$	$\text{cost}_{join}(n)$	$\text{cost}_{rand}(n)$ + $\text{cost}_{join}(\rho \cdot n)$	$\text{cost}_{group}(n)$ + $\text{cost}_{agg}(\delta_v)$	$\text{cost}_{hash}(n)$ + $\text{cost}_{group}(\rho \cdot n)$ + $\text{cost}_{agg}(\rho \cdot \delta_v)$
Refresh	$\text{cost}_{write}(\delta_v)$	$\text{cost}_{write}(\rho \cdot \delta_v)$	$\text{cost}_{write}(\delta_v)$	$\text{cost}_{write}(\rho \cdot \delta_v)$	$\text{cost}_{apply}(\delta_v)$	$\text{cost}_{apply}(\rho \cdot \delta_v)$

update patterns by refreshing the *sample* AggView with the delta *sample* AggView.

Cost Analysis: Table 1 shows the cost comparison between our update-pattern sampling and incremental view maintenance.

- Delta View. For AggView, incremental view maintenance has a grouping cost of $\text{cost}_{group}(n)$ as well as an aggregation cost of $\text{cost}_{agg}(\delta_v)$ where aggregates for each of the groups have to be maintained. In contrast, since we only maintain the update patterns for a sample of groups, we can reduce the grouping cost to $\text{cost}_{group}(\rho \cdot n)$ and the aggregation cost to $\text{cost}_{group}(\rho \cdot \delta_v)$. The additional overhead of sampling is to evaluate hashfunc$(\cdot)$ for each inserted record, i.e., $\text{cost}_{hash}(n)$.
- Refresh. Compared to SPView and FJView, the refresh cost of AggView is a little bit more complicated as we have a combination of insertions into the view and updates to the view. Incremental view maintenance has to apply δ_v records to the entire view while our sampling approach only needs to apply $\rho \cdot \delta_v$ records to the sample view. If there is an index in the view, we can determine which records are new insertions and which correspond to existing records in constant time. Therefore, our sampling technique can reduce the refresh cost from $\text{cost}_{apply}(\delta_v)$ to $\text{cost}_{apply}(\rho \cdot \delta_v)$.

Similar to the analysis results of SPView and FJView, our sampling techniques can save the cost by a factor of $1/\rho$ in almost every stage of incremental view maintenance. The only additional overhead is to compute a hash value for each inserted record (i.e., $\text{cost}_{hash}(n)$). However, as shown in our experiments (Section ??), even with the additional overhead, our sampling techniques can still run orders of magnitude faster than incremental view maintenance.

5. CORRECTING STALE QUERY RESULTS

In this section, we discuss how to correct stale query results using sample update patterns. We first present our correcting query processing in Section 5.1 and then discuss some practical considerations in Section 5.2.

5.1 Correcting query processing

Given a stale view and a query on the view, let ans denote the stale query result. The goal of correcting query processing is to use the sample update patterns w.r.t the view to correct ans. To achieve this goal, our basic idea is to first obtain a *delta query result* (denoted by Δans), by rewriting the query against the sample update patterns, and then combine ans and Δans into a corrected query result.

In the following, we will discuss how to apply this idea to different types of views and analyze the cost of our correcting query processing. For ease of presentation, we assume the query does not have a group-by clause, i.e.,

```
SELECT sum(a)/count(a)/avg(a) FROM View
WHERE Condition(A);
```

Table 2: Correcting a stale query result

	SPView & FJView	AggView
sum	$\text{ans}_{sum} + \Delta\text{ans}_{sum}/\rho$	$\text{ans}_{sum} + \Delta\text{ans}_{sum}/\rho$
count	$\text{ans}_{cnt} + \Delta\text{ans}_{cnt}/\rho$	$\text{ans}_{cnt} + \Delta\text{ans}_{cnt}/\rho$
avg	$\frac{\text{ans}_{cnt} \cdot \text{ans}_{avg} + (\Delta\text{ans}_{cnt}/\rho) \cdot \Delta\text{ans}_{avg}}{\text{ans}_{cnt} + (\Delta\text{ans}_{cnt}/\rho)}$	$\text{ans}_{avg} + \Delta\text{ans}_{avg}$

But note that our methods can be easily extended to a group-by query by treating each group key as an addition condition of the query.

5.1.1 Select-Project and Foreign-Key Join Views

Since both of the update patterns of SPView and FJView only consist of inserted records, their correcting query processing phases are the same. Given the above query, to get its delta query result, we rewrite the query and run it on the sample update patterns. The following show the rewritten queries for different aggregation functions.

```
Δans_sum = SELECT sum(a) FROM sample_update_patterns
           WHERE Condition(A);

Δans_cnt = SELECT count(a) FROM sample_update_patterns
           WHERE Condition(A);

Δans_avg, Δans_cnt = SELECT avg(a), count(a)
                     FROM sample_update_patterns
                     WHERE Condition(A);
```

For the sum and count queries, we only need to compute Δans_{sum} and Δans_{count}, respectively. But for the avg query, in addition to Δans_{avg}, we also need to compute Δans_{cnt} for query correction.

After obtaining delta query results, we next discuss how to use them to correct stale query results. Table 2 shows the corrected results of the sum, count, and avg queries.

To correct a sum query result, we cannot simply add it by the delta query result (i.e., $\text{ans}_{sum} + \Delta\text{ans}_{sum}$) since the delta query result is computed on the sample update patterns. Thus, we have to scale the delta query result according to the sampling ratio and correct the stale query result by adding the rescaled value, i.e., $\text{ans}_{sum} + \Delta\text{ans}_{sum}/\rho$.

To correct a count query result, we can use the same idea as above since a count query can be thought as the special case of a sum query when aggregate attribute values are all equal to one. Thus, the corrected count query result is $\text{ans}_{count} + \Delta\text{ans}_{count}/\rho$.

To correct an avg query result, we can imagine it as computing a weighted average value between a stale view and update patterns. In a stale view, ans_{avg} is the average value over ans_{cnt} records[2], thus its weight is ans_{cnt}; In the sample update patterns, ans_{avg} is the average value over Δans_{cnt} records. Since the value is computed on a sample, its weight is $\Delta\text{ans}_{cnt}/\rho$. Therefore, the corrected avg query result is computed as $\frac{\text{ans}_{cnt} \cdot \text{ans}_{avg} + (\Delta\text{ans}_{cnt}/\rho) \cdot \Delta\text{ans}_{avg}}{\text{ans}_{cnt} + (\Delta\text{ans}_{cnt}/\rho)}$.

[2] ans_{cnt} denotes the number of records that satisfy the avg query's condition, which can be easily obtained from computing ans_{avg}

5.1.2 Aggregation View

The update patterns of AggView contain the information that how every old record should be changed to a new one. Given the query as shown in Section 5.1, let a_{old} (a_{new}) denote the old (new) attribute in the update patterns corresponding to a; let A_{old} (A_{new}) denote the old (new) attribute set in the update patterns corresponding to A. For a single update pattern, we use $\text{Cond}(A_{old}) = 1$ ($\text{Cond}(A_{new}) = 1$) to denote that the old (new) record satisfies the query condition; otherwise, $\text{Cond}(A_{old}) = 0$ ($\text{Cond}(A_{new}) = 0$). The following show the rewritten queries of getting the delta query result for each aggregation function.

$$\Delta\mathsf{ans}_{sum} = \text{SELECT } \texttt{sum}(a_{new} * \text{Cond}(A_{new}) - a_{old} * \text{Cond}(A_{old})) \text{ FROM sample\_update\_patterns;}$$

$$\Delta\mathsf{ans}_{cnt} = \text{SELECT } \texttt{sum}(\text{Cond}(A_{new}) - \text{Cond}(A_{old})) \text{ FROM sample\_update\_patterns;}$$

$$\Delta\mathsf{ans}_{avg} = \text{SELECT } \frac{\texttt{sum}(a_{new} * \text{Cond}(A_{new}))}{\texttt{sum}(\text{Cond}(A_{new}))} - \frac{\texttt{sum}(a_{old} * \text{Cond}(A_{old}))}{\texttt{sum}(\text{Cond}(A_{old}))} \text{ FROM sample\_update\_patterns;}$$

Intuitively, a delta query result tells us how data updates will affect the query result on an old (i.e., a stale) sample AggView. Thus, it is computed as the difference between the query results on a stale sample AggView and an updated sample AggView. To use it to correct a stale query result (Table 2), for the `sum` query, since the `sum` difference is computed on a sample, we need to scale it by the sampling ratio and add the stale query result by the scaled value, i.e., $\mathsf{ans}_{sum} + \Delta\mathsf{ans}_{sum}/\rho$; for the `count` query, since a `count` query can be taken a special case of a `sum` query, similar to the `sum` query, we have $\mathsf{ans}_{count} + \Delta\mathsf{ans}_{count}/\rho$; for the `avg` query, since the `avg` difference on a sample is an unbiased estimation of the `avg` difference on the full data, we add the stale query result by the delta query result directly, i.e., $\mathsf{ans}_{avg} + \Delta\mathsf{ans}_{avg}$.

5.2 Error Bars

The only term in the correction that is not deterministic is $\Delta\mathsf{ans}$. For the SUM, COUNT, and AVG queries, we can bound this term in error bars thus giving us error bars for the entire expression. For these queries, we can rewrite the term $\Delta\mathsf{ans}$ as a mean value of uniformly randomly sampled records, refer to [31] on how to do this. By the Central Limit Theorem, the mean value of numbers drawn by uniform random sampling $\bar{X}$ approaches a normal distribution with:

$$\bar{X} \sim N(\mu, \frac{\sigma^2}{k})$$

Where μ is the true mean, σ^2 is the variance of the numbers, and k is the sample size. We can use this to bound the term with its 95% confidence interval (or any other user specified probability) $\bar{X} \pm 1.96\frac{\sigma}{\sqrt{k}}$. Since the estimate is unbiased the confidence interval tells the user that the true value lies in the confidence interval with the specified probability.

We can prove that for the SUM, COUNT, and AVG queries this estimate is optimal with respect to the variance. We use the following statistical property of random sampling which holds for i.i.d and exchangable sequences of random variables (ie. sampling with and without replacement respectively).

PROPOSITION 1. *The sample mean $\bar{X} = \frac{1}{K}\sum_i X_i$ is the Minimum Variance Unbiased Estimator of the population mean $\mathbb{E}(X)$ over the class of linear estimators when the distribution of X is unknown a priori.*

The concept of a Minimum Variance Unbiased Estimator (MVUE) comes from statistical decision theory [**?**]. Unbiased estimators are ones that, in expectation, give correct estimates. However, on its own, the concept of an unbiased estimate is not useful as we can construct bad unbiased estimates. For example, if we simply pick a random element from a set it is still an unbiased estimate of the mean of the set. Variance is used to compare different unbiased estimates, and the MVUE is the unbiased estimate which has the least variance. Under random sampling, the sample mean is the MVUE over the class of linear estimators when no other information about the distribution is known.

THEOREM 1. *For SUM, COUNT, and AVG queries, our estimate of the correction is optimal over the class of linear estimators when no other information is known about the distribution.*

PROOF. Using the proposition above, reformulating our correction estimates as sample means and the true query results as population means, we can prove that our approach gives the lowest variance estimate over the class of linear estimators. □

6. OUTLIER INDEXING

The presented approach may be sensitive to long-tailed distributions and power-laws which are common in large datasets [**?**]. Sampling may also hide any outliers, records with abnormally large or small attribute values. Since oultiers may occur very rarely, they are unlikely to be represented in a small sample. We address this problem using a technique called outlier indexing which has been applied in SAQP [5]. The user specifies a threshold t and an attribute on one of the base tables. If a record has an attribute that exceeds t it is added to the index. We ensure that when we sample the update patterns for views that these records are represented in the sample. The user can additionally place a size limit on the outlier index k. The same approach can be extended to attributes that have tails in both directions by making the threshold t a range, which takes the highest and the lowest values. However, in this section, we present the technique as a threshold for clarity.

6.1 Building The Outlier Index

The first step is that the user selects an attribute of the base table to index and specifies a threshold t and a size limit k. In a single pass of updates, the index is built storing references to the records with attributes greater than t. If the size limit is reached, the incoming record is compared to the smallest indexed record and if it is greater then we evict the smallest record.

To select the threshold, we can use prior information from the base table. This approach is, of course, a heuristic which assumes that the updates are distributionally similar to the base table. This can be done in the background during the periodic maintenance cycles. If our size limit is k, we can find the records with the top k attributes in the base table as to set a threshold to maximally fill up our index. Then, the attribute value of the lowest record becomes the threshold t.

6.2 Adding Outliers to the Sample

Since the outlier indices index the base table, all of the materialized views in the system share a common set of these indices. We discuss how to ensure that these records are added to the sample and what overhead this introduces. For Select-Project and Foreign-Key Join views, we sample updates to view. In the same pass as the sample, we can test

each record against the outlier indices. If the record exists in any of the indices it is added to the sample with a flag indicating that it is an outlier.

For aggregation views, we sample the view by taking a hash of the group by keys. If a record is in the outlier index, we must ensure that all records with its group by key are also added to the sample. To achieve this, we have to select all of the distinct group by keys in the outlier index and add those aggregates to the sample. As before, these records are marked with a flag denoting they are outliers.

Outlier indexing adds additional overhead since for Select-Project and Join views it adds the overhead of a hash table lookup for each record, and for Aggregation views it further requires a scan of the entire index. However, we envision that in most datasets the outlier index will be very small making this overhead negligible. In our experiments, we show that even a very small outlier index (less than .01% of records) is sufficient to greatly improve the accuracy of correction estimates.

6.3 Query Processing with the Outlier Index

The outlier index has two uses: (1) we can answer *selection* queries on the outliers, and (2) we can improve the accuracy of our *aggregation* queries. For selection queries, we must simply check to see if the record satisfying the query is in the outlier index. While the outlier index is small, outliers (and their derived materialized view rows) are often the records that we want to query.

We can also incorporate the outliers into our correction estimates. By guaranteeing that certain rows are in the index, we have to merge a deterministic result (set of outlier rows) with the estimate. One way to think of this is that we have $\Delta\mathsf{ans}$ is calculated from the set of records that are not outliers. Let $\Delta\mathsf{ans}_{cnt} + \mathsf{ans}_{cnt}$ be the size of the updated view, and l be the number of rows in the outlier index. Furthermore, let $\Delta\mathsf{ans}(o)$ be the delta answer calculated on only the outliers. We can update $\Delta\mathsf{ans}$ with the outlier information by:

$$\frac{\Delta\mathsf{ans}_{cnt} + \mathsf{ans}_{cnt} - l}{\Delta\mathsf{ans}_{cnt} + \mathsf{ans}_{cnt}}\mathsf{ans} + \frac{l}{\Delta\mathsf{ans}_{cnt} + \mathsf{ans}_{cnt}}\Delta\mathsf{ans}(o)$$

7. RESULTS

Our experiments are organized in the following way. We first evaluate our approach on a synthetic benchmark dataset where we can control the data distribution and the update rate. Our first experiment will discuss the practical overheads and tradeoffs to using our approach. Then, for three specific materialized views, we evaluate accuracy, performance, and outlier indexing. After evaluation on the benchmark, we present an end-to-end application of log analysis with a dataset from a video streaming company.

We define the following metrics for evaluation. For accuracy, we measure the relative error between estimated query result and the true query result. A relative error of 5% means that the difference between the estimated query result and the true result is 5% of the true result. Whenever we discuss error, we discuss it in terms of average query results. That is for a set of queries on the view what is the mean relative error. For accuracy, we compare against two alternative approaches, no maintenance where the materialized view is left to be stale and SAQP where instead of calculating an approximate query correction we estimate directly from the up-to-date view. We fix the sampling ratio of SAQP and our approach to be the same so they will have the same maintenance time. For performance, we measure the maintenance time as the time required to calculate the delta view and then refresh the materialized view for a batch of updates. As a baseline, we compare the performance to full incremental maintenance.

7.1 Experimental Setting

7.1.1 TPCD-Skew

The first dataset, TPCD-Skew dataset, is based on the Transaction Processing Council's benchmark schema but is modified so that it generates a dataset with values drawn from a Zipfian powerlaw distribution instead of uniformly. The Zipfian distribution [**?**] is a long-tailed distribution with a single parameter $z = \{0, 1, 2, 3, 4\}$ which a larger value means a more extreme tail. This database has been applied to benchmark other sampling based approaches, approximate queries, and outlier performance [**?**,5]. The dataset is provided in the form of a generation program which can generate both the base tables and a set of updates.

For this dataset, we applied our approach to three materialized views, each of a different type:

Select-Project View

```
SELECT *, if(lcase(l_shipinstruct) LIKE '%
    deliver%' AND lcase(l_shipmode) LIKE '%air
    %', 'priority', 'slow')
FROM lineitem_s
```

Aggregation View

```
SELECT l_orderkey, l_shipdate, sum(l_quantity)
    as quantity_sum, sum(l_extendedprice) as
    extendedprice_sum, max(l_receiptdate) as
    receiptdate_max, count(*) as group_count
FROM LINEITEM GROUP BY l_orderkey, l_shipdate
```

Foreign-Key Join View

```
SELECT supplier.*, customer.*
FROM   customer, orders, lineitem, supplier,
    partsupp
WHERE  c_custkey = o_custkey  AND o_orderkey =
    l_orderkey  AND l_suppkey = ps_suppkey AND
    l_partkey = ps_partkey AND ps_suppkey =
    s_suppkey AND s_nationkey <> c_nationkey
```

The base dataset is 10GB corresponding to 60,000,000 records, and the above three views are derived from this base dataset. Then, we randomly generated aggregation queries and evaluated them for different sample sizes, different update rates, and different parameter settings for the Zipfian distribution. All of our experiments for the TPCD-Skew dataset are run on a single r3.large Amazon EC2 node with a MySQL database.

7.1.2 Conviva

Conviva is a video streaming company and we evaluated our approach on user activity logs. We experimented with a 1TB dataset of these logs and a workload corresponding analyst queries on the log. We used this dataset to evaluate the end-to-end accuracy and performance of the system on a real dataset and query workload. We evaluated performance on Apache Spark with a 20 node r3.large Amazon EC2 cluster.

7.2 Update Rate and Accuracy

In the first experiment, for each of the three views listed above, we evaluate the accuracy of our approach. We measure accuracy by using the average relative query error for a set of 10000 randomly generated aggregation queries on

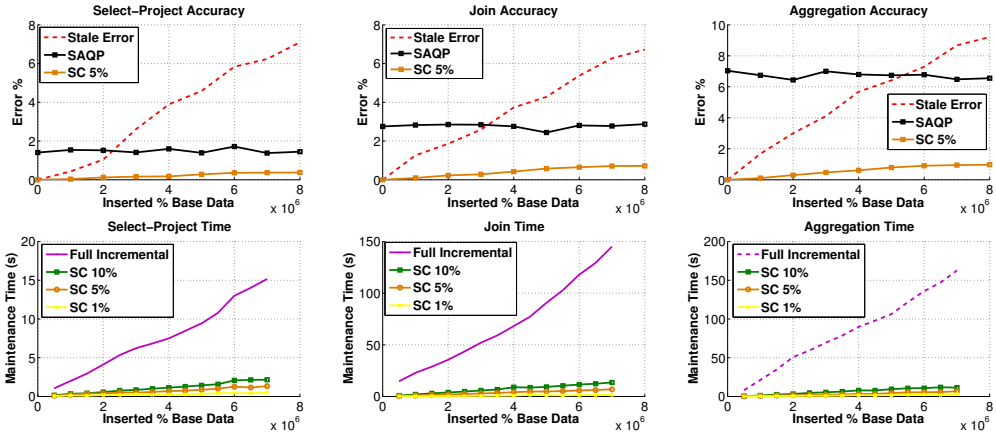

Figure 4: For each of the three views listed above, we plot the accuracy for 5% samples compared to SAQP and a stale baseline. We also plot the maintenance time for each of the views in comparison to full incremental maintenance for a 1%, 5% and 10% sample.

each view. We set the sample size to 5% and then vary the number of inserted records by increments of 500000 records to a final count of 8000000 records (1.3GB). In Figure **??**, we show the results of the experiment.

The results highlight an important point comparing the accuracy of SAQP to our approach. The accuracy our approach is proportional to the amount of correction needed, while SAQP keeps a roughly constant accuracy. As more records are inserted the approximation error in our approach increases. However, we find that even for very large amounts of inserted records (>10% of dataset size), our approach gives significantly more accurate results than SAQP. The gain is most pronounced in aggregation views where there are a mixture of updated and inserted rows into the view. Correcting an update to an existing row is often much smaller than doing so for a new inserted record.

7.3 Computational Efficiency

In the next experiment (Figure **??**), we evaluate how sampling can reduce maintenance time. For a batch of updates, we evaluate how long it takes to incrementally maintain a view compared to maintaining a sample. For, the join view, we build an index on the foreign key of the view. As before, we inserted records in increments of 500000. We find that for the Aggregation and Join view, which are more expensive to maintain, sampling leads to more pronounced savings. Furthermore, there are diminishing returns with small sample sizes as overheads start becoming significant.

7.4 Overhead of Query Correction

In Section 4, we ran cost analyses to argue that sampling can greatly reduce delta view and refresh costs. In practice, these gains will only be meaningful if the overheads for the query correction are small. Since, we are correcting a stale query rather than the materialized view, we are shifting some of the computational cost from maintenance time to query execution time. If our updates only contain insertions, then for Select-Project and Foreign-Key join views, since we derive a correction from a sample of the updates, we are guaranteed to have a reduced query time. However, for aggregation views, query correction requires a scan of the old view and a sample of the maintained view; potentially increasing the time to answer the query. In our first experiment, we set the sample size to 5% and evaluate the aggregation view listed earlier for 5000000 inserted records. In (Figure 7.4), we show that average query time is small compared to maintenance time, and furthermore, the time to calculate a correction is a just fraction of the query time.

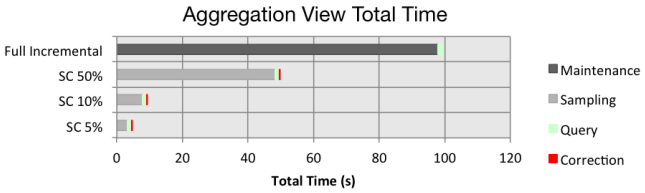

Figure 5: The overhead introduced by correcting a query is small compared to the overall maintenance time.

7.5 View Complexity

In our maintenance time experiments, we showed that more expensive views tended to benefit from our approach. We evaluated this tradeoff by taking the simplest possible view, a SELECT of the base table, and then progressively adding clauses to the predicate. For example:

```
WHERE (condition1)
WHERE (condition1 || condition2)
WHERE (condition1 || condition2 || condition 3)
```

We set the sample size to 5% and measure the maintenance time. For a such a view, the only cost for maintenance is a scan of the data and evaluating the predicate. Sampling saves on predicate evaluation but introduces the overhead of random number generation. Figure 7.5 illustrates how as the view becomes for complex the performance improvement given by our approach increases. Initially, the random number generation adds about 5% overhead, but as the cost of evaluating the benefits increase. We repeated the same experiment for the aggregation view, but instead we added terms to the group by clause to increase the cardinality of the view. We found that for a highly selective group by clause (thus a large view) the savings were nearly perfect (20x). However, when the view was small the cost savings were

smaller. This experiment emphasizes that our approach

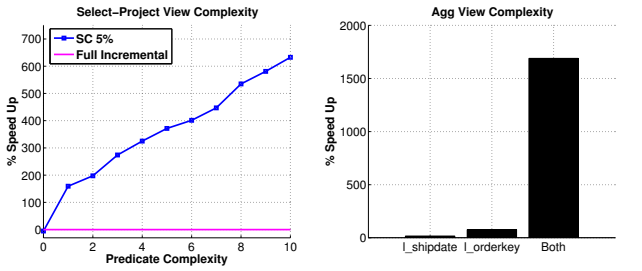

Figure 6: As the views definitions become more complex our approach gives increasing gains. For simplest view (selection no predicate), the overhead is small (5%).

rarely makes performance worse and at worst the improvements are not linear with the sampling ratio. Furthermore, we emphasize that when views are large and complex, as often the case, we can have significant improvements.

7.6 Outlier Indexing

We evaluated the accuracy for each of the views as a function of the size of the outlier index. We use a 5% sample size and 5000000 records inserted, and we evaluate the accuracy of our approach with and without outlier indexing. We apply the index to the Select-Project view, where we index the attributes l_extendedprice and l_quantity. The results for the other two views were similar, and in this paper, we only present results for the Select-Project views.

In the first experiment, we varied the total number of indexed records and plot accuracy as a function of index size. As a baseline, we compare if the same number of records were randomly sampled. In Figure 7.6, we find that outlier indexing can improve accuracy by a factor of 2.5 for this view. Next, we evaluated how the approach works under different levels of skew in the dataset. We varied the Zipfian parameter from (0,1,2,3,4) and measured the average improvement in accuracy. We found that as the dataset become more skewed, outlier indexing is increasingly important. Finally, we evaluated the overhead for the approach. We found that small indicies were sufficient for good accuracy and thus the additional overhead was marginal and orders of magnitude smaller than sample maintenance time.

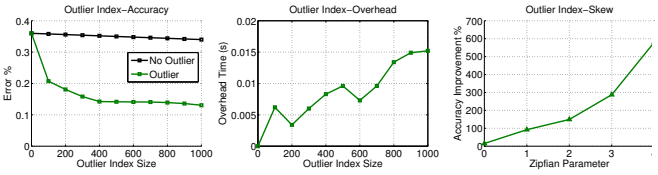

Figure 7: Outlier indexing greatly improves accuracy of our approach especially in skewed datasets for a small overhead of building the index and ensuring those rows are in the view.

7.6.1 *Distribution of Query Error*

In our earlier experiments, we presented the average error for the queries on the views. In this experiments, we looked at the distribution of query errors compared to their staleness. For each of the three views, we derive the views from a 10GB dataset. Then, as before, we simulate 5000000 inserted records and a 5% sample. We evaluated the relative error for each query.

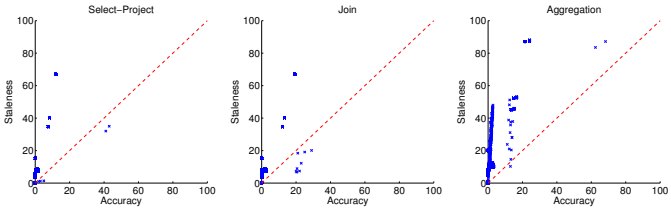

Figure 8: On the x-axis is the accuracy of our approach and on the y-axis is the error if the query was not maintained. We find that there are many outlier queries results–motivating the need for outlier indexing.

In Figure 7.6.1, we present staleness vs. estimation accuracy as a scatter plot. For all three of the views, the median error is greater than 10 times less than the staleness.

However, this distribution motivates our outlier indexing as there are clearly outlier queries. Outliers can occur for a variety of reasons. First, some of the generated queries are highly selective and a uniform sampling approach may not sample enough records to answer it accurately. Next, outliers can be caused by large updates to the data that are missed by our sampling. In Section ?, we show how outlier indexing can solve the latter problem and, in fact, after outlier indexing all of our queries on these three views are estimates more accurately than the pre-existing staleness error.

7.7 Sample Size and Accuracy

In this experiment, we illustrate the tradeoff between sample size and accuracy. We set a fixed 5000000 inserted records, and then vary the sampling ratio for the three types of views and show how much sampling is needed to achieve a given query accuracy. In Figure 7.7, we show the accuracy as a function of sampling ratio for each of the views. For both SAQP and our approach, there is a break-even point at which the approximation error is less than the staleness error. At sampling ratios beyond this point, our query approximated queries are on average more accurate than queries on the stale view. For all three of the views, the average query error is significantly less than SAQP as the break-even point happens earlier in our approach.

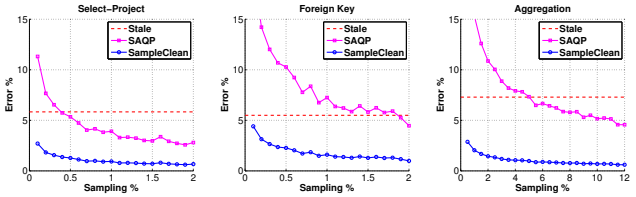

Figure 9: As we increase the sample size, the relative error goes down following a $\frac{1}{\sqrt{k}}$ rate. SAQP also follows the same rate so the lines will never cross by only changing the sample size.

7.8 Application: Log Analysis For Conviva

We used a dataset of queries given by Conviva to generate three materialized views. All three of these views are aggregation views, and for confidentiality, we exclude the actual

definitions of the views. View 1 has the most selective group by clause and was chosen to a be a large view where most of the maintenance is in the form of new rows to insert. View 1 aggregates session times for each visitor and video pair. View 2 is a medium size view where maintenance is a mix of both updates and insertions. View 2 computes buffering time statistics for each visitor over all videos they watched. View 3 is a small view where most of the maintenance is updates to existing rows. View 3 computes statistics for log events that triggered error states.

We selected these views to be representative of the dataset and query workload. We first present an experiment illustrating where these materialized views lie in the space of all the generated views from the conviva query log. We derived each view from a 7GB base table and inserted 1.5GB of records to the table. On a single node, we evaluated the average query accuracy for a 1% sample and the runtime of our approach. In Figure 7.8, we plot the three views on the distribution of error and runtime and we see that they are from different parts of the distribution.

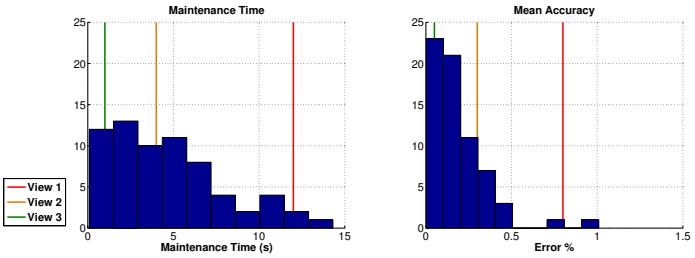

Figure 10: We chose three views that represented the distribution of all views well in terms of both accuracy and maintenance time.

7.8.1 Accuracy in Conviva

We evaluated the average query accuracy for different sample sizes and numbers of records inserted. As before, we derived each view from a 7GB base table and inserted 1.5GB of records to the table. We built an outlier index on all of the attributes that represent time intervals or rates. Figure 7.8.1, compares the accuracy to the staleness of the query. We find that even a 0.1% sample gives significantly more accurate results for View 1 and View 2. Even in the situation where the view is small, sampling can still have benefits as seen in View 3.

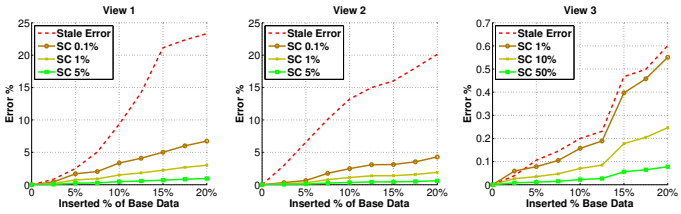

Figure 11: We compare average query staleness to our approach for a variety of sample sizes. For View 1 and View 2, we find that small sample sizes can give accurate results. Even in the extreme case of View 3, we find that sampling can still provide reasonably accurate results.

7.8.2 Performance in Conviva

We evaluated performance on Apache Spark, on a 20 node r3.large Amazon EC2 cluster. This cluster had enough memory to hold all of the materialized views in memory. Spark supports materialized views through a distributed data structure called an RDD [?]. The RDD's are immutable, thus requiring significant overhead to maintain. We compare maintenance to recalculation of the view when the data is in memory and when only the updates are in memory. We derived the views from a 700GB base table and inserted records in 15GB increments. We scaled these experiments up significantly from the accuracy experiments to illustrate the performance benefits of sampling at a large scale, however, as insertions as a percentage of the base data are the same. As Spark does not have support for indices, we rely on partitioned joins for incremental maintenance of the aggregation views. Each view is partitioned by the group by key, and thus only the sampled delta view has to be shuffled.

In Figure 7.8.2, we illustrate the maintenance time as a function of the number of inserted records. We find that we achieve good scalability on view 1 and view 2, and still have some performance gains on view 3. In view 1 and view 2, our gains are more pronounced due to the savings on communication in a distributed environment. Aggregation views with a larger cardinality create larger delta views. These delta views necessitate a shuffle operation that communications them during the refresh step. As view 3 is smaller, the communication gains are less and the only savings are in computation.

8. RELATED WORK

Addressing the cost of materialized view maintenance is the subject of many recent works with techniques including reducing coordination [4], using additional information about the view (eg. if the view is derived from linear algebraic operations) [22], and notification-based systems [24]. The increased research focus parallels an major concern in industrial systems for incrementally updating pre-computed results and indices such as Google Percolator [24], LinkedIn [26], and Twitter's Rainbird. The steaming community has also studied the view maintenance problem [1,9,12,13,17], in Spark Streaming they studied how they could exploit in-memory materialization [33], and in MonetDB they studied how ideas from columnar storage can be applied to enable real-time analytics [21].

Sampling has also been well studied in the context of query processing [2,8,23]. However, we argue, that our application of sampling in this work has a fundementally different goal. Prior work emphasizes sampling as a technique to reduce query execution time. We, on the other hand, use sampling to reduce maintenance costs. This is similar to the goals of load shedding studied in streaming databases [27,30]. Babcok et al. studied load shedding in the context of predefined aggregate queries, however, did not support ad hoc queries on the streaming data [3].

Sampling has also been studied in the context of materialized views. Joshi and Jermaine [18] proposed using sampled materialized views. This technique mirrors what we called SAQP in our evaluation. Their focus, however, was not addressing incremental maintenance costs but rather operations on materialized views. They proposed a tree-like data structure that could support queries on multiple materialized views and operations on these views. Gibbons et al. studied the maintenace of approximate histograms [10,11], which closely resemble aggregation materialized views. They, however, did not consider queries on these histograms but took a holistic approach to analyze the error on the entire histogram. We contrast our approach

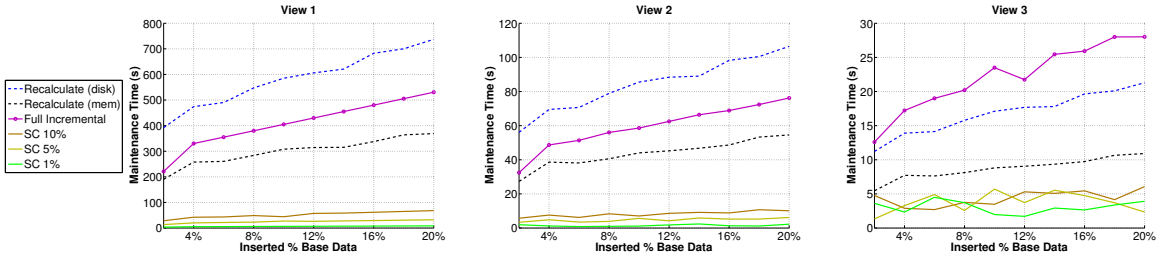

Figure 12: As before, we compare maintenance times for a variety of sample sizes and full incremental maintenance. We further include a comparision with view recalculation as our experimental platform, Spark, does not support indices or selective writes.

from those proposed in Joshi and Jermaine and Gibbons et al. since we do not estimate our query results directly from a sample. We use the sample to learn how the updates affect the query results and then compensate for those changes. Our experiments suggest that prior approaches (SAQP) are inefficient when updates are sparse and the maintenance batch is small compared to the base data.

Building off Gibbon et al. there are a variety of other works proposing storage efficent processing of aggregate queries on streams [7,14] which are similar to materialized views. Furthermore, there is a close relationship between sampling and probabilistic databases, and view maintenance and selection in the context of probabilistic databases has also been studied [29].

9. CONCLUSION

In this paper, we proposed a new approach to the staleness problem in materialized views. We demonstrated how recent results from data cleaning, namely sampling, query correction, and outlier detection, can allow for accurate query processing on stale views for a fraction of the cost of incremental maintenace. We evaluated this approach on a single node and in a distributed environment and found that sampling can provide a flexible tradeoff between accuracy and performance. In our end-to-end experiments with a 1TB log dataset from Conviva, we found that maintaining a 1% sample was 87x faster yet only with mean query error of 4%

Our results are promising and suggest many avenues for future work. We are interested in extending this approach to support a more general class of queries. To support general selection queries, we will have to extend our query correction to predict and attribute value via a regression. This is an ideal application for non-parametric regression techniques such as Gaussian Process Regression which do not make strong assumptions about the distribution of the data. This work also has applications in machine learning. We believe there is a strong link between pre-computed machine learning models and materialized views, and the principles of our approach could be applied to build fast, approximate streaming machine learning applications.

10. REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[2] 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 *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[3] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 350–361. IEEE, 2004.

[4] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Scalable atomic visibility with ramp transactions. In *ACM SIGMOD Conference*, 2014.

[5] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. Narasayya. Overcoming limitations of sampling for aggregation queries. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 534–542. IEEE, 2001.

[6] R. Chirkova and J. Yang. Materialized views. *Databases*, 4(4):295–405, 2011.

[7] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 61–72. ACM, 2002.

[8] M. N. Garofalakis. Approximate query processing: Taming the terabytes. In *VLDB*, 2001.

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

[10] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *ACM SIGMOD Record*, volume 27, pages 331–342. ACM, 1998.

[11] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *VLDB*, volume 97, pages 466–475, 1997.

[12] L. Golab and T. Johnson. Consistency in a stream warehouse. In *CIDR*, volume 11, pages 114–122, 2011.

[13] L. Golab, T. Johnson, and V. Shkapenyuk. Scalable scheduling of updates in streaming data warehouses. *Knowledge and Data Engineering, IEEE Transactions on*, 24(6):1092–1105, 2012.

[14] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *ACM SIGMOD Record*, volume 30, pages 58–66. ACM, 2001.

[15] A. Gupta, I. S. Mumick, et al. Maintenance of materialized views: Problems, techniques, and applications.

[16] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[17] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: batched stream processing for data intensive distributed computing. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 63–74. ACM, 2010.

[18] S. Joshi and C. Jermaine. Materialized sample views for database approximation. *Knowledge and Data Engineering, IEEE Transactions on*, 20(3):337–351, 2008.

[19] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3363–3372. ACM, 2011.

[20] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey. Opportunistic physical design for big data analytics. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 851–862. ACM, 2014.

[21] E. Liarou, S. Idreos, S. Manegold, and M. Kersten. Monetdb/datacell: online analytics in a streaming column-store. *Proceedings of the VLDB Endowment*, 5(12):1910–1913, 2012.

[22] M. Nikolic, M. ElSeidy, and C. Koch. Linview: Incremental view maintenance for complex analytical queries. *arXiv preprint arXiv:1403.6968*, 2014.

[23] F. Olken. *Random sampling from databases*. PhD thesis, University of California, 1993.

[24] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In

Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, 2010.

[25] L. L. Perez and C. M. Jermaine. History-aware query optimization with materialized intermediate views. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 520–531. IEEE, 2014.

[26] L. Qiao, K. Surlaker, S. Das, T. Quiggle, B. Schulman, B. Ghosh, A. Curtis, O. Seeliger, Z. Zhang, A. Auradar, et al. On brewing fresh espresso: Linkedin's distributed data serving platform. In *Proceedings of the 2013 international conference on Management of data*, pages 1135–1146. ACM, 2013.

[27] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman. Aggregation and degradation in jetstream: Streaming analytics in the wide area. *NSDI, April*, 2014.

[28] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[29] C. Ré and D. Suciu. Materialized views in probabilistic databases: for information exchange and query optimization. In *Proceedings of the 33rd international conference on Very large data bases*, pages 51–62. VLDB Endowment, 2007.

[30] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 309–320. VLDB Endowment, 2003.

[31] J. Wang, S. Krishnan, M. J. Franklin, K. Goldberg, T. Milo, and T. Kraska. A sample-and-clean framework for fast and accurate query processing on dirty data. *Aqua*, page t4, 1999.

[32] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[33] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 10–10. USENIX Association, 2012.

[34] J. Zhou, P.-A. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *Proceedings of the 33rd international conference on Very large data bases*, pages 231–242. VLDB Endowment, 2007.