

A Data Cleaning Approach to Materialized View Management

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 When base tables are updated, however, materialized views can become out-of-date and any queries issued to these views will give stale results.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 a type of data error and model the view-maintenance problem as a data-cleaning problem. Instead of maintaining a materialized view, we aim to correct a query result on the stale view using a sample. 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. For common aggregate queries (SUM, COUNT, and AVG), 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 found that we are able to provide more up-to-date query results with a smaller mean error and at a lower maintenance cost than other methods.

1. INTRODUCTION

Materialized views, stored pre-computed query results, are used to optimize query processing on large datasets [7, 18, 19]. Materialization and related concepts such as selecting which queries to materialize have been well studied in recent research [5, 22, 29, 37]. This research has further expanded beyond the SQL setting [25] 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 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 [7, 18].

Unfortunately, in many desired applications, incrementally maintaining the view for every incoming update can be very costly. When the base tables change rapidly, the database system has to propagate updates to all of the materialized views. Consequently, there are many techniques that defer maintenance to a later time [7, 39]. 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 e.g. 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, recent work has considered the costs of data cleaning [35]. 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 dirty 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 samples the “update patterns” and uses this sample to derive a correction. (2) we use the sampled update patterns to 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 clean the stale query result. Finally, in (3) sampling has the potential to mask outliers, and in fact, it is known long-tailed distributions are poorly approximated from samples [6]. In this framework, we utilize a technique called outlier indexing [6], which guarantees that rows in the materialized view derived from an “outlier” record (one that has abnormal attribute values) is contained

in the sample, which can be used to increase correction accuracy.

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 with staleness as a type of data error.
- We define the concept of “update patterns”, a logical unit of up-to-date data that represents how an update affects a materialized view, and show how to sample the update patterns. We show analytically and empirically that sampling update patterns can be far more efficient than full incremental maintenance.
- Using a sample of update patterns, we can correct stale aggregation queries on materialized views. We bound these corrections in confidence intervals and prove optimality of our approach.
- We use an outlier index to increase the accuracy of the approach for power-law, long-tailed, and skewed distributions.
- We evaluate our approach on real and synthetic datasets in both 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. Then, in Section 7, we evaluate our approach. Finally, we end with our Related Work in Section 8 and our Conclusions and Future Work in Section 9.

2. BACKGROUND

2.1 Incremental Maintenance Main Ideas

Incremental maintenance of materialized views is well studied, see [7] 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 table 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 U 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:

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

The details of the refresh operation depend on the view definition. Refer to [7] 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, any derived materialized view is also updated. Immediate maintenance has an advantage that queries on the materialized view are 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. 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 in a pre-set way or adaptively. In deferred maintenance approaches, the user often accepts some degree of staleness for additional flexibility for in scheduling. 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 [39]. 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.

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 [31]. 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 a sample of data, and then bounds the results of aggregate queries on dirty datasets [35] with respect to the clean data.

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 Approx-

imate 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 correct for the staleness. The SAQP approach to this problem, would be to estimate the result directly from the maintained sample [21]. We found that estimating a correction and leveraging an existing deterministic result led 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 freshness and performance. Immediate maintenance guarantees that query results on the view will always be fresh, 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 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 the more rows that we clean. 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 this paper, we present our approach with modeling only insertions to the base tables, refer to the extended version of the paper **{{Extended Paper}}** for details on how to model deletions and updates.

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 e.g. nightly. While maintaining the entire view throughout the day may be infeasible, sampling allows the database to scale the cost 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.

3.1 System Architecture

The architecture of our proposed solution is shown in Figure 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

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

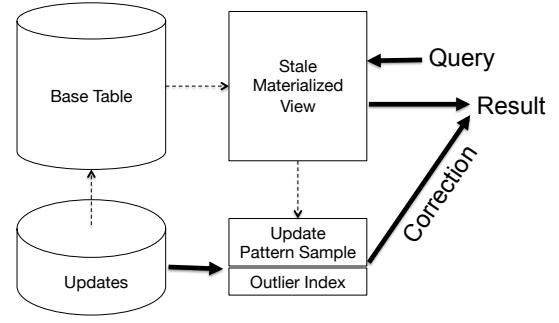


Figure 1: We clean a stale query result by deriving a correction from a sample of up-to-date data. To make this process robust to skewed datasets, we couple sampling with an outlier indexing approach. For aggregation queries, we can guarantee that our results are unbiased and bounded. **{{What’s the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

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, ...]
```

There is a taxonomy of aggregation views and we discuss different types of aggregate functions (distributive, algebraic, holistic) can be applied in our framework **{{Extended Paper}}**.

3.1.2 Sampling the Update Pattern

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 since they are affected by updates differently. 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.

| 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.

3.1.3 Correcting a Query

Suppose, we issue an aggregate query to the stale view. We can use information from the sample to correct stale query results. 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 are 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. We guarantee that records (or rows in the view derived from those records) in the outlier index 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.

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
```

(a-1) An example of SPView

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

(b-1) An example of FJView

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

(c-1) An example of AggView

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

(a-2) Update Patterns for the SPView

| | sessionId | videoid | responseTime |
|------------|-----------|---------|--------------|
| Insert r1: | 1 | 214 | 18 ms |
| Insert r2: | 2 | 2056 | 15 ms |

(b-2) Update Patterns for the FJView

| | sessionId | videoid | responseTime | duration |
|------------|-----------|---------|--------------|----------|
| Insert r1: | 45 | 567 | 12 ms | 0:34:23 |
| Insert r2: | 5566 | 20 | 20 ms | 1:00:08 |

(c-2) Update Patterns for the AggView

| | videoid | maxResponseTime (old) | maxResponseTime (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.

```
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.

4. SAMPLING UPDATE PATTERNS

Updates can affect the three classes of materialized views in different ways, making sampling challenging. 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 in Section 4.2. We also analyze the costs and overheads of our methods.

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 row into a view; (2) Update attribute values of

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 | $\text{cost}_{write}(\delta_v)$ | $\text{cost}_{write}(\rho \cdot \delta_v)$ | $\text{cost}_{write}(\delta_v)$ | $\text{cost}_{write}(\rho \cdot \delta_v)$ | $\text{cost}_{apply}(\delta_v)$ | $\text{cost}_{apply}(\rho \cdot \delta_v)$ |

an existing row in the view; (3) Keep the view 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 rows. For instance, Figure 3(a-1) shows an example of SPView. When the base table *Log* is updated, there might be some new rows 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 rows 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 updates could be inserting a new group, updating the aggregation attribute values of an existing group, or keep the view unchanged. 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 ΔT . Given a SPView defined on the table, its full update patterns involve the inserted records of ΔT 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 ΔT , and then keep the records that satisfy the view's predicate as a sample.

For a given FJView, let T be the fact table, and T_1, T_2, \dots, T_k be the dimension tables. The update pat-

terns of the FJView involve the inserted records of $\Delta T \bowtie (T_1 + \Delta T_1) \bowtie (T_1 + \Delta T_1) \dots \bowtie (T_k + \Delta T_k)$ that satisfy the predicate of the FJView¹. Similar to SPView, to sample the update patterns (e.g., $\rho = 5\%$), we first randomly select 5% records from ΔT , and then join the selected records with the up-to-date dimension tables (i.e., $T_i + \Delta T_i$ ($1 \leq i \leq k$)), and finally return as a sample the joined records that satisfy the FJView's predicate.

Cost Analysis. Let $n = |\Delta T|$ 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 existing groups in the view as well as groups to be inserted. Since, we may have to update existing groups, unlike SPView and FJView, the first step is to construct a sample of all of the existing group keys in the AggView. To sample the group keys, we apply a hash function to the key. We define a hash function, $\text{hashfunc}(\cdot)$, which takes a single group key as an input and outputs a number in the range of $[0, 1]$, and filter

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

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

Thus, if the hash function is uniform the maintained group keys are a random sample of all the group keys with the sampling ratio of ρ

Once, we have a random sample of the existing keys, we can use this sample to construct a sample of the update patterns. When a records are inserted into the base table, we first calculate a delta *sample* **AggView**; which only keeps those group by keys that satisfy the condition on the hash above (i.e lep). This ensures that if a key is already in the sample, our delta *sample* **AggView** contains this key. For keys that are not already in the sample, the hash condition filters them to select a random sample of them with sampling ratio of ρ . This process ensures that a group is either fully represented in the sample or not at all, thus its aggregate is an exact result.

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 $\text{hashfunc}(\cdot)$ 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 7), even with the additional overhead, our sampling techniques can save maintenance cost compared to full incremental 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 correction query processing in Section 5.1 and then discuss the error bars and optimality of query corrections 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**. We first obtain a *delta query result* (denoted by Δans), by applying the query to the sample update patterns, and then combine **ans** and Δans into a corrected query result.

We process aggregation queries of the following form,

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 | $\frac{\text{ans}_{cnt} \cdot \text{ans}_{avg} + (\Delta\text{ans}_{cnt}/\rho) \cdot \Delta\text{ans}_{avg}}{\text{ans}_{cnt} + (\Delta\text{ans}_{cnt}/\rho)}$ | $\text{ans}_{avg} + \Delta\text{ans}_{avg}$ |

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

For ease of presentation, we assume the query does not have a group-by clause. But our methods can be easily extended to a group-by query by treating each group key as an additional 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 we can obtain corrections in the same way. Given the above query, to obtain its delta query result, we rewrite the query and run it on the sample update patterns as follows.

```
 $\Delta\text{ans}_{sum}$  = SELECT sum(a) FROM sample_update_patterns
WHERE Condition(A);
```

```
 $\Delta\text{ans}_{cnt}$  = SELECT count(a) FROM sample_update_patterns
WHERE Condition(A);
```

```
 $\Delta\text{ans}_{avg}, \Delta\text{ans}_{cnt}$  = 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. In Table 2, we show how to use the delta query result to correct a stale query result.

To correct a **sum** query result, 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 treat it as computing a weighted average between a stale view and update patterns. In a stale view, ans_{avg} represents the average value of ans_{cnt} rows², thus its weight is ans_{cnt} ; In the sample update patterns, ans_{avg} represents the average value of Δans_{cnt} rows. Since this is computed on a sample, the weight w.r.t the full update patterns is $\Delta\text{ans}_{cnt}/\rho$. Therefore, the corrected **avg** query result is computed as a weighted average between ans_{avg} and Δans_{avg} , i.e., $\frac{\text{ans}_{cnt} \cdot \text{ans}_{avg} + (\Delta\text{ans}_{cnt}/\rho) \cdot \Delta\text{ans}_{avg}}{\text{ans}_{cnt} + (\Delta\text{ans}_{cnt}/\rho)}$.

5.1.2 Aggregation View

The update patterns of **AggView** contain the information that how every old record in the view 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 set of old (new) attributes in the update patterns corresponding to A . In an update pattern, we use $\text{Cond}(A_{old}) = 1$

² ans_{cnt} denotes the number of rows that satisfy the **avg** query's condition, which can be easily obtained when computing ans_{avg}

($\text{Cond}(A_{\text{new}}) = 1$) to denote that its old (new) record satisfies the query condition; otherwise, $\text{Cond}(A_{\text{old}}) = 0$ ($\text{Cond}(A_{\text{new}}) = 0$). To obtain the delta query result, we apply the query to the sample update patterns as follows:

$\Delta\text{ans}_{\text{sum}} = \text{SELECT } \text{sum}(a_{\text{new}} * \text{Cond}(A_{\text{new}}) - a_{\text{old}} * \text{Cond}(A_{\text{old}}))$
FROM sample_update_patterns;

$\Delta\text{ans}_{\text{cnt}} = \text{SELECT } \text{sum}(\text{Cond}(A_{\text{new}}) - \text{Cond}(A_{\text{old}}))$
FROM sample_update_patterns;

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

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 (as shown in Table 2), for the **sum** query, since the **sum** difference is computed on a sample, we need to scale it to the full data, and add the rescaled value to the old query result, i.e., $\text{ans}_{\text{sum}} + \Delta\text{ans}_{\text{sum}}/p$; 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 $\text{ans}_{\text{count}} + \Delta\text{ans}_{\text{count}}/p$; 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 delta query result to the old query result directly, i.e., $\text{ans}_{\text{avg}} + \Delta\text{ans}_{\text{avg}}$.

5.2 Error Bars and Optimality

The only terms in the correction that are not deterministic is Δans and $\Delta\text{ans}_{\text{cnt}}$. For the **sum**, **count**, and **avg** queries, we can bound these terms in error bars thus giving us error bars for the entire expression (refer to [35] on how to do this). The intuition is that we can rewrite the term Δans as a mean value of uniformly randomly sampled records. 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, 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), e.g., $\bar{X} \pm 1.96 \frac{\sigma}{\sqrt{k}}$. Since the estimate is unbiased, the confidence interval indicates 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 exchangeable sequences of random variables (i.e., 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. In other words, there exists no other linear function of the set $\{X_i\}$ that gives a lower variance estimate of the population mean than the sample mean.*

The concept of a Minimum Variance Unbiased Estimator (MVUE) comes from statistical decision theory [9]. Unbiased estimators are ones that, in expectation, are correct. 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. \square

6. OUTLIER INDEXING

Sampling update patterns may be sensitive to power-laws and other long-tailed distributions which are common in large datasets [8]. Sampling may also hide any outliers, records with abnormally large or small attribute values. Since outliers 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 [6]. The basic idea is that we create an index of outlier records and ensure that these records are included in the sample.

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. 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.

To select the threshold, there are many heuristics that we can use. For example, we can use our knowledge about the dataset to set a threshold. Or we can use prior information from the base table, a calculation which 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 we index the base table, all of the materialized views in the system share a common set of these outlier indices. We discuss how to ensure that these records are added to the sample and what overhead this introduces. For **SPView** and **FJView**, we sample the records that are inserted into the 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 update pattern 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 single initial 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 query all the rows that correspond to outlier rows, and (2) we can improve the accuracy of our *aggregation* queries. To query the outlier rows, we can select all of the rows in the materialized view that are flagged as outliers, and these rows are guaranteed to be up-to-date.

We can also incorporate the outliers into our correction estimates. We can use our outlier flag to ensure that we do not double count them in our results, thus avoiding bias. By guaranteeing that certain rows are in the index, we have to merge a deterministic result (set of outlier rows) with the estimate. This process truncates the distribution and it can be modeled as computing a query result on regular records and on the outliers separately; and then combining them together with a weighted average. If our dataset size is N , and we have l outliers. Let v_{reg} be the query result for the regular records, and v_{out} is the query result for outliers, then:

$$v = \frac{N-l}{N}v_{reg} + \frac{l}{N}v_{out}$$

We can use this method improve the accuracy of our correction estimates, by calculating Δ ans on the outliers and the regular records separately then averaging them together. See [6] for additional query processing details.

7. RESULTS

Our experiments are organized in the following way. We first evaluate our approach, which we denote as **Sample View Clean (SVC)**, on a synthetic benchmark dataset where we can control the data distribution and the update rate. For three specific materialized views, we evaluate accuracy and performance. Next, we evaluate overheads and tradeoffs to using our approach. Then, we evaluate the benefits and tradeoffs of 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.

7.1 Experimental Setting

7.1.1 TPCD-Skew

The first dataset, TPCD-Skew dataset [1], 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 distribution instead of uniformly. The Zipfian distribution [24] 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 dataset has been applied to benchmark other sampling based approaches, approximate queries, and outlier performance [4,6]. The dataset is provided in the

form of a generation program which can generate both the base tables and a set of updates. Unless otherwise noted, we experiment with $z = 2$.

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

{{format the three sqls}}

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. Along with the TPCD specification are templates to generate a query workload to benchmark database systems. We used these templates to generate queries and filtered the query workload to the aggregation queries and the ones whose predicates could be applied on our views. In all, for each view, we had 10,000 aggregate queries with different randomly generated predicates from the template. 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 corresponding dataset of analyst queries on the log. Using the query dataset, we constructed 67 views and queries to run on the views. 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 Accuracy

Update Rate and Accuracy : In the first experiment, for each of the three views listed above, we evaluate the accuracy of our approach. We set the sample size to 5% and then vary the number of inserted records by increments of 500000 (0.8% base data) records to a final count of 8000000 records (13.3% base data). For the 10,000 generated queries on the view, we calculate the mean error as a relative percentage of the true value. As a baseline, we compare against SAQP and no maintenance (ie. the stale error). In Figure 4, we show the results of the experiment.

For all three views a 5% sample sufficed to achieve a mean relative error of less than 1%, even though when 8000000 records were inserted the staleness was 7%. Furthermore, in

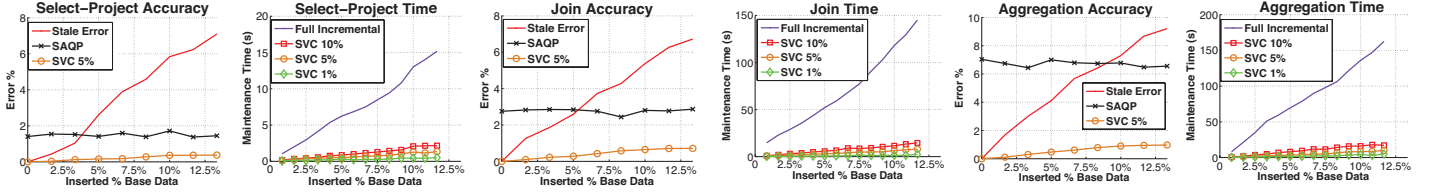


Figure 4: For each of the three views listed above, we plot the accuracy for 5% samples compared to SAQP and a stale baseline. We also plot the maintenance time for each of the views in comparison to full incremental maintenance for a 1%, 5% and 10% sample.

comparison to SAQP, 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.

Sample Size and Accuracy : The next question is how much sampling is necessary to achieve accurate results. We inserted 5000000 (8% base data) 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 5, we show the accuracy as a function of sampling ratio for each of the views. In our experiment, we found that a 0.1% sample was sufficient to ensure the approximation error due to sampling was less than the baseline staleness. SAQP also has this break even point, but we found for this number of inserted records the point SAQP required a much larger sample size.

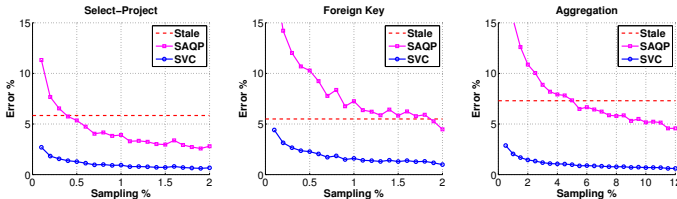


Figure 5: As we increase the sample size, the relative error goes down following a $\frac{1}{\sqrt{\text{sample size}}}$ rate. SAQP also follows the same rate so the lines will never cross by only changing the sample size.

Distribution of Query Error : In our earlier experiments, we presented the average error for the queries on the views. We found that for inserted 5000000 records (8% base data) on average queries on the views were 7-9% stale. However, that some queries were much more stale than others. In this experiment, we looked at the distribution of staleness for the aggregation view. We used a sampling ratio of 5% and evaluate the accuracy of our approach. In Figure 6, we show a CDF of the staleness and a scatter plot comparing the staleness of the query to the accuracy using our correction method.

The long tail of query errors can occur for a variety of reasons. First, some of the generated queries are highly selective and a uniform sampling approach may not sample

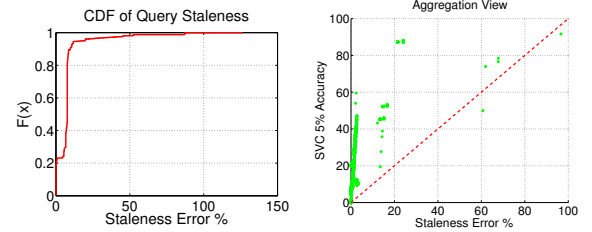


Figure 6: While the mean error is 9%, 5% of queries have more than 20% relative error. However, in the second plot, we show a scatter plot of staleness and a query correction accuracy. Points above the dotted line show that we improved query accuracy.

enough records to answer it accurately. Next, outliers can be caused by large updates to the data that are missed by our sampling, which we will address in subsequent experiments.

7.3 Efficiency

Sampling Efficiency: In Figure 4, we evaluate how sampling can reduce maintenance time. For a batch of updates, we evaluate how long it takes to sample the update patterns as opposed to maintaining the view. As a baseline, we compare to full incremental maintenance. For the join view, we build an index on the foreign key of the view. For the aggregation view, we build an index on the group by key of the view. We used the same range of increments as the accuracy experiment. We find that for the AggView and FJView, which are more expensive to maintain, sampling leads to more pronounced savings. For 10% of data inserted and a 10% sampling ratio, the maintenance time for FJView is 9.35x faster and for AggView it is 8.4x faster. However, for the SPView the gains are smaller with a 6.2x speedup. There are diminishing returns with smaller sample sizes as overheads start becoming significant.

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 SPView and FJView, 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 evalu-

ate the aggregation view listed earlier for 5000000 inserted records. In Figure 7, 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.

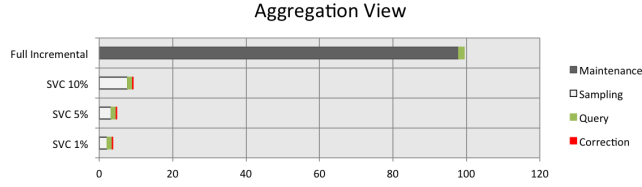


Figure 7: While, there is an additional overhead to correcting a query, this is small compared to the overall maintenance time.

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 || condition3)
```

We set the sample size to 5%, insert 5000000 records (8% of Base Data), and measure the maintenance time. For the selection 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 (see Section 4.1). Figure 8 illustrates how as the view becomes for complex the performance improvement given by our approach increases. Initially, there is about 5% overhead, but as the cost of evaluating the benefits increase to a 7.9x speedup over incremental maintenance. 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 16.9x. However, when the view was small with the Lshipdate key, the cost savings were smaller 5.5x. This experiment em-

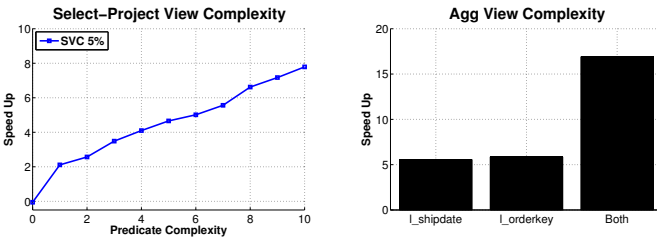


Figure 8: As the views definitions become more complex our approach gives increasing gains. For simplest view (selection no predicate), the overhead is small (5%).

phasizes that our approach 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.

7.4 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 (8% of Base Data) 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 Lextendedprice and Lquantity. 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 set the outlier index size to the top 500 records in the base table, and looked at the change in accuracy of the top 10 most in accurate query corrections with SVC 5%. We find that for these queries, we can improve the accuracy by a factor of 8.2. We further evaluated the overhead of the approach compared to the base SVC time of 1.44s. We found that the overhead was only 1% for an outlier index of size 1000. We varied the Zipfian parameter from (0,1,2,3,4), which makes the distribution more skewed, and measured the average improvement in accuracy over all queries. We found that as the dataset become more skewed, outlier indexing is increasingly important leading to almost a 6x more accurate estimate for $z = 4$.

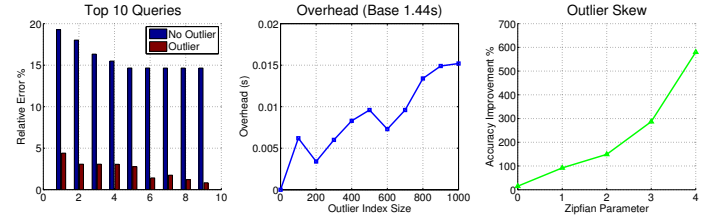


Figure 9: Outlier indexing greatly improves accuracy of our approach especially in skewed datasets for a small overhead of building the index and ensuring those rows are in the view.

7.5 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. LargeView has the most selective group by clause and was chosen to be a large view where most of the maintenance is in the form of new rows to insert. LargeView aggregates session times for each visitor and video pair. MediumView is a medium size view where maintenance is a mix of both updates and insertions. MediumView computes buffering time statistics for each visitor over all videos they watched. SmallView is a small view where most of the maintenance is updates to existing rows. SmallView 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. In Figure 10, we plot the average staleness of each of the views and mark the three views.

7.5.1 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

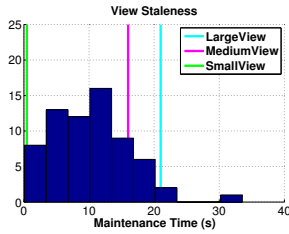


Figure 10: We chose three views that represented the distribution of all views well in terms of both accuracy. We plot the staleness of each of the 67 views and mark the three views in our experiment.

of records to the table. We built an outlier index on all of the attributes that represent time intervals or rates. Figure 11, compares the accuracy to the staleness of the query. We find that even a 0.1% sample gives significantly more accurate results for LargeView and MediumView. Even in the situation where the view is small, sampling can still have benefits as seen in SmallView.

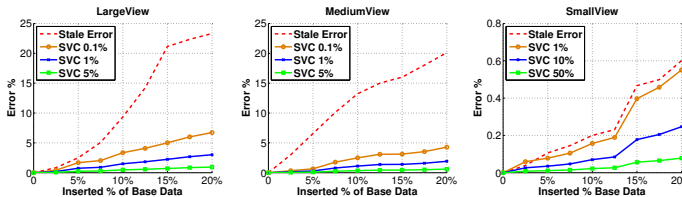


Figure 11: We compare average query staleness to our approach for a variety of sample sizes. For the larger views, we find that small sample sizes can give accurate results. Larger sample sizes are needed for accurate estimates in SmallView.

7.5.2 Performance in Conviva

We evaluated performance on Apache Spark, on a 20 node r3.large Amazon EC2 cluster. 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 approximately 14GB 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 12, we illustrate the maintenance time as a function of the number of inserted records. As Spark does not support selective writes, we include view recalculation as a baseline for comparison. In many Spark applications, recalculation is often faster than incremental maintenance. For LargeView a 10% sample with 105GB of inserted records (15%) achieves a 7.5x speedup over full incremental maintenance and a 5.6x speedup over recalculation. On the other hand, for SmallView, there is a 5.1x speedup over full incremental maintenance and a 1.7x improvement over recal-

culation. 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 SmallView 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 focus on various perspectives including complex analytical queries [25], transaction [5], and physical design [22]. The increased research focus parallels a major concern in industrial systems for incrementally updating pre-computed results and indices such as Google Percolator [28] and Twitter’s Rainbird [36]. The streaming community has also studied the view maintenance problem [2,13,15,16,20]. In Spark Streaming they studied how they could exploit in-memory materialization [38], and in MonetDB they studied how ideas from columnar storage can be applied to enable real-time analytics [23].

Sampling has been well studied in the context of query processing [3,12,26]. Particularly, a similar idea of sampling from updates has also been applied in stream processing [11,30,34]. But none of these works studied how to sample update patterns w.r.t materialized views and how to use the sample update patterns to correct stale query results.

Sampling has also been studied in the context of materialized views [21,27]. These techniques mirror what we called SAQP in our evaluation. Gibbons et al. studied the maintenance of approximate histograms [14], 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 from those proposed 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.

There are a variety of other works proposing storage efficient processing of aggregate queries on streams [10,17] 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 [32]. Srinivasan and Carey studied compensation for concurrency errors that happen during query processing [33] **{{Differentiate with our work}}**.

9. CONCLUSION AND FUTURE WORK

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 maintenance. 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 5% sample was 16.3x faster yet with only a mean query error of 1.4%.

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

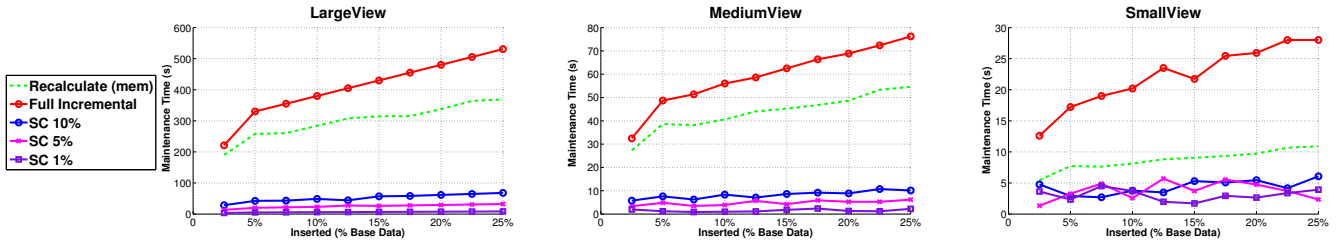


Figure 12: As before, we compare maintenance times for a variety of sample sizes and full incremental maintenance. We further include a comparison with view recalculation as our experimental platform, Spark, does not support indices or selective writes.

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] S. Chaudhuri and V. Narasayya. tpc-d data generation with skew. <ftp.research.microsoft.com/users/viveknar/tpcdskew>.
- [2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.
- [3] 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.
- [4] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database tuning advisor for microsoft sql server 2005. In *VLDB*, pages 1110–1121, 2004.
- [5] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Scalable atomic visibility with ramp transactions. In *SIGMOD Conference*, pages 27–38, 2014.
- [6] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. R. Narasayya. Overcoming limitations of sampling for aggregation queries. In *ICDE*, pages 534–542, 2001.
- [7] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.
- [8] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.
- [9] D. R. Cox and D. V. Hinkley. *Theoretical statistics*. CRC Press, 1979.
- [10] A. Dobra, M. N. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *SIGMOD Conference*, pages 61–72, 2002.
- [11] M. Garofalakis, J. Gehrke, and R. Rastogi. *Data Stream Management: Processing High-Speed Data Streams*. Springer, 2011.
- [12] M. N. Garofalakis and P. B. Gibbons. Approximate query processing: Taming the terabytes. In *VLDB*, 2001.
- [13] 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.
- [14] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. *ACM Trans. Database Syst.*, 27(3):261–298, 2002.
- [15] L. Golab and T. Johnson. Consistency in a stream warehouse. In *CIDR*, pages 114–122, 2011.
- [16] L. Golab, T. Johnson, and V. Shkapenyuk. Scalable scheduling of updates in streaming data warehouses. *IEEE Trans. Knowl. Data Eng.*, 24(6):1092–1105, 2012.
- [17] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD Conference*, pages 58–66, 2001.
- [18] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.
- [19] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.
- [20] 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 *SoCC*, pages 63–74, 2010.
- [21] S. Joshi and C. M. Jermaine. Materialized sample views for database approximation. *IEEE Trans. Knowl. Data Eng.*, 20(3):337–351, 2008.
- [22] J. LeFevre, J. Sankaranarayanan, H. Hacigümüs, J. Tatemura, N. Polyzotis, and M. J. Carey. Opportunistic physical design for big data analytics. In *SIGMOD Conference*, pages 851–862, 2014.
- [23] E. Liarou, S. Idreos, S. Manegold, and M. L. Kersten. MonetDB/DataCell: Online analytics in a streaming column-store. *PVLDB*, 5(12):1910–1913, 2012.
- [24] M. Mitzenmacher. A brief history of generative models for power law and lognormal distributions. *Internet Mathematics*, 1(2):226–251, 2003.
- [25] M. Nikolic, M. Elseidy, and C. Koch. Linview: incremental view maintenance for complex analytical queries. In *SIGMOD Conference*, pages 253–264, 2014.
- [26] F. Olken. *Random sampling from databases*. PhD thesis, University of California, 1993.
- [27] F. Olken and D. Rotem. Maintenance of materialized views of sampling queries. In *ICDE*, pages 632–641, 1992.
- [28] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, pages 251–264, 2010.
- [29] L. L. Perez and C. M. Jermaine. History-aware query optimization with materialized intermediate views. In *ICDE*, pages 520–531, 2014.
- [30] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman. Aggregation and degradation in jetstream: Streaming analytics in the wide area. In *NSDI*, 2014.
- [31] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.
- [32] C. Re and D. Suciu. Materialized views in probabilistic databases for information exchange and query optimization. In *VLDB*, pages 51–62, 2007.
- [33] V. Srinivasan and M. J. Carey. Compensation-based on-line query processing. In *SIGMOD Conference*, pages 331–340, 1992.
- [34] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.
- [35] J. Wang, S. Krishnan, M. J. Franklin, K. Goldberg, T. Kraska, and T. Milo. A sample-and-clean framework for fast and accurate query processing on dirty data. In *SIGMOD Conference*, pages 469–480, 2014.
- [36] K. Weil. Rainbird: Real-time analytics at twitter. In *Strata*, 2011.
- [37] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.
- [38] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438, 2013.
- [39] J. Zhou, P.-Å. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *VLDB*, pages 231–242, 2007.