Sample-View-Clean: A Data Cleaning and Outlier Indexing Approach For Managing Materialized Views

ABSTRACT

Materialized views, stored pre-computed query results, are widely used to facilitate fast queries on large datasets. When base tables are updated, however, materialized views can become out-of-date and queries will give stale results, especially in the Big Data era when new data arrives at an increasingly fast rate. In *Sample-View-Clean*, we approach staleness as 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 results suggest that we are able to provide more up-to-date query results with a smaller mean error and at a lower maintenance cost than previous methods.

1. INTRODUCTION

Materialized views, stored pre-computed query results, are used to facilitate fast query processing on large datasets [7,18,19]. Materialization and related concepts such as selecting 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 avoid expensive recalculation, incrementally updating the views, also called incremental maintenance, has been well studied [7,18].

Unfortunately, in many applications, incremental maintenance on each update can be very costly. Consequently, it is common to 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, for example, at night. Deferral allows for increased flexibility to meet the resource constraints of the system, but may not guarantee that the view will be up-to-date.

In this work, we address the view-maintenance problem from a different perspective. We explore whether refreshing stale rows in a materialized view can be modeled as a 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 also considered the costs of data cleaning [35]. 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 incurring the cost of full maintenance.

We propose Sample-View-Clean (SVC), a framework that uses a *sample* of up-to-date data to *clean* stale aggregation query results. While approximate, the corrected query result can be bounded within confidence intervals. This framework will be complementary to existing deferred maintenance approaches; when the materialized views are stale between maintenance cycles, we can apply SVC for approximate results for far less costs than having to maintain the entire view. Without SVC, existing work allows the user to control the freshness of queries by changing maintenance parameters (e.g. nightly maintenance vs. hourly maintenance) based on prior experience. Without bounds on the results, a burst of updates can lead to unexpected changes in query accuracy. On the other hand, SVC gives results that are fresh and the user controls a bounded approximation error with the sampling ratio.

SVC has three main components: (1) sampling, (2) correction, and (3) outlier indexing. In (1), we define an "update pattern" which represents how an update affects the derived view, and we take a sample of these patterns. (2) From the sample, we estimate how much the updates affect the query and we use this estimate to clean the stale query result. Finally, in (3) sampling is known to be sensitive to outliers [6]. 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 and staleness as a type of data error.
- We define the concept of "update patterns", a logical unit that represents how an update affects a materialized view,

and show how to sample the update patterns.

• 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 sampling and query processing of our technique. In Section 6, we describe the outlier indexing framework. Then, in Section 8, we evaluate our approach. Finally, we end with our Related Work in Section 9 and our Conclusions and Future Work in Section 10.

2. BACKGROUND

2.1 Incremental Maintenance

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 $\mathbf{V}_T$. For the "delta view" step, we apply the view definition to the updates U and we call the intermediate result a "delta" view $\Delta\mathbf{V}$. For the refresh step, 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. (See [7]).

2.2 Maintenance Strategies

There are two principle incremental 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 on updates to the base table. 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 in scheduling. For example, views can be updated at night when the system can use more resources to process the updates without affecting a critical application. However, this also means that during the day the materialized view becomes increasingly stale as it was computed the night before.

These costs can also be deferred to query execution time. In particular, we highlight a technique called lazy maintenance which applies updates to the view only when a user's query requires a row [39]. Both lazy maintenance and immediate maintenance hit a bottleneck when there rapid updates making these approaches impractical.

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]. Recent work has begun to consider the costs of data cleaning as well as how to budget data cleaning effort. The SampleClean project framework cleans a sample of data, and then bounds the results of aggregate queries on dirty datasets with respect to the clean data [35].

One of the algorithms in SampleClean, takes a random sample of dirty records, applies data cleaning, and learns how to correct a query applied to the dirty data. Instead of modeling data error on the base table, SampleClean considers how dirtiness affects queries on the data. In SVC, we look at queries on stale materialized views from this perspective. As materialized views are different problem setting, there are new challenges such as sampling from changing base tables, selection queries, and considering the effects of outliers.

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 use a sample to correct a query rather than directly estimating the query result. 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 8).

3. SYSTEM OVERVIEW

In this section, we give an overview of the entire architecture of SVC. We first define the scope of SVC including which types of views and queries we address. Then, we briefly present each of the components of our system.

3.1 Problem Setting

3.1.1 Materialized Views

In this paper, we evaluate SVC on three classes of materialized views, Select-Project, Foreign-Key Join, and Aggregation:

Select-Project Views (SPView): These views are defined by Select-Project expressions of the following form:

```
SELECT [a1,a2,...] FROM Table
WHERE Condition(A)
```

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.[a1,a2,...],
table2.[a1,a2,...] FROM Table1, Table2
WHERE
Table1.fk = Table2.fk  AND Condition(A);
```

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

```
SELECT [f1(a1),f2(a2),...]
FROM Table
WHERE Condition(A)
GROUP BY [a3,a4,...];
```

We selected these classes of views to be specific enough to analyze in terms of cost but general enough to evaluate using real datasets and query workloads. There are, however, other types of materialized views that SVC can support including views defined by nested queries, equality joins, and aggregate queries with `HAVING` clauses.

3.1.2 Updates

There are three types of updates that can affect a base table: `INSERT`, `DELETE`, `UPDATE`. We can model `UPDATE` as a `DELETE` of the old record then an `INSERT` of a new record with updated values. Insertion-only workloads are increasing common **{{Tim will find a citation}}**. Thus, the primary focus of this paper is on analysis and results for `INSERT`. In Section 7.2, we discuss how to modify our query processing to support `DELETE`, which in turn allows us to support `UPDATE`.

3.1.3 Supported Queries

In this paper, we focus on deriving corrections for the three commonly used aggregation queries on the view. For ease of presentation, we assume the query does not have a group-by clause, which can be easily extended by treating each group key as an additional condition of the query:

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

For these queries, we prove optimality of our approach with respect to estimate variance (Section 5.3). We also provide support for correcting stale Select queries of the following form:

```
SELECT * FROM View
WHERE Condition(A);
```

Since the results of these queries are not numeric, we instead bound the cardinality of the results. Refer to Section 7.1 for a discussion of how to correct Select queries with SVC.

3.2 System Architecture

Modeling view maintenance as a data cleaning problem gives us a new perspective for addressing staleness. Insertions to the base table can cause two types of errors in the view: rows are either out-of-date or missing altogether. 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 new records to process. Suppose, we cleaned a random sample of dirty rows processing only as many of the new insertions as necessary this would potentially reduce the amount of updates, join executions, and aggregates we would need to compute. From this sample, we can derive a correction to stale query results.

The architecture of SVC is shown in Figure 1. The diagram depicts the following components: (1) 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 implementation, SVC will work in conjunction with existing maintenance or re-calculation approaches. We envision the scenario where materialized views are being refreshed periodically, for example 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.

3.2.1 Sampling the Update Pattern

The three classes of views require different sampling techniques since they are affected by updates differently. For example, insertions to the base table only result in insertions to Select-Project views. But, for Aggregation Views, insertions to the base table can also result in 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.

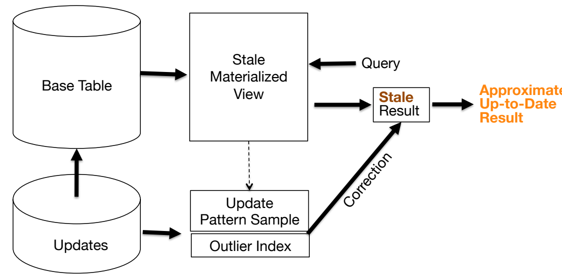

Figure 1: We clean a stale query result by deriving a correction from a sample of up-to-date data. To make this process robust to skewed datasets, we couple sampling with an outlier indexing approach. For aggregation queries, we can guarantee that our results are unbiased and bounded.

3.2.2 Outlier Indexing

We use an outlier index to reduce the sensitivity of our correction estimates to skewed distributions. We guarantee that records (or rows in the view derived from those records) in the outlier index are also included in the sample. This can be used to reduce variance in our estimates. See Section 6 for details on this component.

3.2.3 Correcting a Query

We can use information from the sample to correct stale query results. For the aggregate functions, `sum`, `count`, and `avg`, we calculate a correction which is bounded and provably optimal. Like the sampling, the algorithm to calculate the correction varies between the types of views. We detail query correction in Section 5.

3.2.4 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 2). 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
FROM Log
GROUP BY videoId;
```

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
```

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.

(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 r_1:	1	214	18 ms
Insert r_2:	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 r_1:	45	567	12 ms	0:34:23
Insert r_2:	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 r_1:	5	NULL	15690 ms
Update r_2:	1	1298 ms	6298 ms
Keep r_3:	65	798 ms	798 ms

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

Let us suppose the initial query result is 15. There now have been new log records inserted into the Log table making the old result of 15 stale. For example, if our sampling ratio is 5%, 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. The cost analysis of this section ties back to our discussion of incremental maintenance in Section 2.1 we analyze how sampling can save costs in comparison to full incremental 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 row into a view; (2) Update attribute values of 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.

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.

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.

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

The update-pattern model gives us a logical unit to sample the updates to a materialized view. In this section, we show how to construct these samples and show how the cost of sampling is less than full incremental maintenance. The cost is further parameterized by the sampling ratio allowing the user to tradeoff accuracy or performance. Potentially far cheaper than full incremental maintenance, update pattern sampling can be applied in situations where incremental maintenance was deemed infeasible such as between periodic maintenance cycles.

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 base table, and $T_1, T_2, \cdots, T_k$ also base tables with which are linked to T with foreign-key relationships. The 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 tables

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

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

(i.e., $T_i + \triangle T_i$ $(1 \leq i \leq k)$), and finally return as a sample the joined records that satisfy the FJView's predicate.

Cost Analysis. Let $n = |\triangle T|$ 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).

- 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. For SVC, there is first a cost to sample $\rho \cdot n$ records $\text{cost}_{rand}(n)$. Then sampling reduce the predicate and join evaluations by a factor of ρ as we only need to evaluate them on our sample.
- 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 when the view definitions are more complex to evaluation, for example when the predicate or the join is expensive to evaluate, our sampling reduces costs significantly.

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. We can create this sample when the stale materialized view is created. To sample the group keys, we apply a hash function to the key. We define a hash function, hashfunc$(\cdot)$, which takes a single group key as an input and outputs a number in the range of [0, 1], and filter

$$\text{hashfunc}(\text{“group key”}) \leq \rho. \tag{1}$$

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 records are inserted into the base table, we first calculate a delta *sample* AggView; which only keeps those group keys that satisfy the condition on the hash above (i.e $\leq \rho$). 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 the 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 hashfunc$(\cdot)$ for each inserted record, i.e., $\text{cost}_{hash}(n)$.
- Refresh. In full incremental view maintenance, δ_v rows have to be updated or inserted while our sampling approach only processes $\rho \cdot \delta_v$ rows. To avoid scanning the view for each update, if the view is indexed by group key, we can determine which updates are new insertions and which correspond to existing ones 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, sampling allows for greater savings when the cost maintaining the view is higher, for example, when the refresh step is expensive.

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 analyze the costs in Section 5.2. Then, we discuss the error bars and optimality of query corrections in Section 5.3.

5.1 Correction query processing

Given a stale view and a query on the view, let ans denote the stale query result. The goal of correction 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.

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.

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

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

$$\Delta\mathsf{ans}_{avg}, \Delta\mathsf{ans}_{cnt} = \text{SELECT } \texttt{avg}(a),\ \texttt{count}(a) \text{ FROM sample\_update\_patterns WHERE Condition(A);}$$

For the `sum` and `count` queries, we only need to compute $\Delta\mathsf{ans}_{sum}$ and $\Delta\mathsf{ans}_{count}$, respectively. But for the `avg` query, in addition to $\Delta\mathsf{ans}_{avg}$, we also need to compute $\Delta\mathsf{ans}_{cnt}$ for query correction. In Table 2, we show how to use the delta query result to correct a stale query result.

Table 2: Correcting a stale query result

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

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 re-scaled value, i.e., $\mathsf{ans}_{sum} + \Delta\mathsf{ans}_{sum}/\rho$.

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 $\mathsf{ans}_{count} + \Delta\mathsf{ans}_{count}/\rho$.

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} rows[2], thus its weight is ans_{cnt}; In the sample update patterns, ans_{avg} represents the average value of $\Delta\mathsf{ans}_{cnt}$ rows. Since this is computed on a sample, the weight w.r.t the full update patterns is $\Delta\mathsf{ans}_{cnt}/\rho$. Therefore, the corrected `avg` query result is computed as a weighted average between ans_{avg} and $\Delta\mathsf{ans}_{avg}$, i.e., $\frac{\mathsf{ans}_{cnt} \cdot \mathsf{ans}_{avg} + (\Delta\mathsf{ans}_{cnt}/\rho) \cdot \Delta\mathsf{ans}_{avg}}{\mathsf{ans}_{cnt} + (\Delta\mathsf{ans}_{cnt}/\rho)}$.

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

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

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

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

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., $\mathsf{ans}_{sum} + \Delta\mathsf{ans}_{sum}/\rho$; 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 delta query result to the old query result directly, i.e., $\mathsf{ans}_{avg} + \Delta\mathsf{ans}_{avg}$.

5.2 Cost Analysis

We analyze the costs of query correction in SVC. The cost of query execution has two components: we first apply the query to the stale view, and then correct the result by processing the update pattern sample. Therefore, the cost is proportional to the size of the stale view and update pattern sample in comparision to the size of the up-to-date view. For FJView and SPView, if the only updates to the base table are insertions, the combined size of the update pattern sample and stale view are guaranteed to be less than the size of the fully maintained view, thus our query execution time is guaranteed to be smaller than full maintenance.

We cannot make such a guarantee for AggView. In AggView, our update pattern sample also contains a sample of existing rows from the view. Thus, the combined size of the stale view and the update pattern sample may be larger than up-to-date view. Since our sample size is often small, for example 1%, the additional query execution time is often small as well. Furthermore, if the insertions to the base table result in many new groups added to AggView, then similar to the analysis for FJView and SPView our query execution cost can be less than full maintenance.

5.3 Error Bars and Optimality

5.3.1 Confidence Intervals

In Table 2, we present formulas to correct stale queries. In each of these formulas, all of the $\Delta\mathsf{ans}$ terms correspond to estimates and these estimates need to be bounded. The intuition is that we can rewrite the terms $\Delta\mathsf{ans}$ as a mean value of uniformly random variables. 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.[3] We can use this to bound the term with its 95% confidence interval (or any other user specified probability), e.g., $\bar{X} \pm 1.96\frac{\sigma}{\sqrt{k}}$.

For all of the queries in AggView, and for `sum`, `count` in SPView and FJView, there is only one term to bound. However, for `avg` in SPView and FJView, as presented there are two terms to bound. Using the associativity properties of these queries, we can re-write the correction for `avg` in terms of a single summation. The numerator can be written as a single `sum` query, where we move the stale result $\mathsf{ans}_{cnt} \cdot \mathsf{ans}_{avg}$ inside the estimate. Next, we notice that the reformulation is now a `sum` over a `count` of the same set of records (with an additional offset of ans_{cnt}. Thus, this

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

[3] For sampling without replacement, there is an additional term of $\sqrt{\frac{N-k}{N-1}}$. We consider estimates where N is large in comparison to k making this term insignificant.

can be re-written as a single mean value. See [35] for more details on the confidence intervals.

5.3.2 Optimality

We can prove that for the `sum`, `count`, and `avg` queries this estimate is optimal with respect to the variance.

PROPOSITION 1. *An estimator is called a minimum variance unbiased estimator (MVUE) of a parameter if it is unbiased and the variance of the parameter estimate is less than or equal to that of any other unbiased estimator of the parameter.*

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. The variance of the estimate determines the size of our confidence intervals thus is important to consider.

The `sum`, `count`, and `avg` queries are linear functions of their input rows. We explored whether our estimate was optimal for linear estimates, for example, should we weight our functions when applied to the sample. It turns out that the proposed corrections are the optimal strategy when nothing is known about the data distribution a priori.

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. In other words, there exists no other linear function of the set input rows $\{X_i\}$ that gives a lower variance correction.*

PROOF SKETCH. We can reformulate `sum`, `count`, and `avg` as means over the entire table. `sum` is the mean multiplied by the dataset size, and `count` is the sum where all the records have been set to one. Then, we prove the theorem by induction. Suppose, we have two independent estimates of the mean $\bar{X}_1$ and $\bar{X}_2$ with unknown variance and we want to merge them into one estimate $\bar{X} = c_1\bar{X}_1 + c_2\bar{X}_2$. Since, we do not know anything about the distribution by symmetry $c_1 = c_2 = 0.5$. Thus we can recursively induct, and we find that if we do not know the variance of data, as is impossible with new updates, then equally weighting all input rows is optimal. □

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 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 [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 AggView, we sample the update pattern by taking a hash of the group keys. If a record is in the outlier index, we must ensure that all records with its group key are also added to the sample. To achieve this, we have to select all of the distinct group 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.

We check the outlier index prior to sampling to ensure that rows added due to the outlier index are not double counted from the sample. The flags ensure that we always know which records were sampled and which come from the outlier index.

Outlier indexing adds additional overhead since for SPView and FJView 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. By guaranteeing that certain rows are in the index, we have to merge two results: one over the outliers and one over the regular records. For a given aggregation query, let N be the count of records that satisfy condition and l be the number of outliers that satisfy the condition. 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 $\Delta\mathsf{ans}$ on the outliers and the regular records seperately then averaging them together.

7. EXTENSIONS

In this section, we present two additional features of the SVC framework: Select queries and Deletions. These two features greatly broaden the scope of the problem that SVC addresses.

7.1 Select Queries

We can correct stale Select queries with SVC. For FJView and SPView, a stale Select query is missing rows. To correct this result, we can apply the query to the update pattern sample, we get a sample of records that satisfy the predicate. Then, we can take a union of the sampled selection and the stale selection. To quantify the approximation error, we can rewrite the Select query as `count` to get an estimate of the up-to-date size of the result. This gives us an estimate of how many rows are missing from our approximate result.

For AggView, stale Select queries can also have out-of-date rows as well as missing rows. In this case, we can still apply the query to the update pattern sample and get a sample of rows to update and new rows to insert. For the updated rows in the sample, we overwrite the out-of-date rows in the stale query result. To quantify the approximation error, we can give two bounds. As before, we can estimate the number of missing rows in the result with a `count` query. We further can estimate the number of existing rows that are not up-to-date with a `count` query.

In comparison to no maintenance, this approach gives a less stale result. Furthermore, we give estimates on how many rows are missing or stale in the result providing the user with key bounds on their result accuracy.

7.2 Deletions

In the previous sections, we presented SVC focusing on `INSERT` operations to the base table. These insertions defined "update pattern" which we sampled to calculate an approximate correction. To model `DELETE` operations, we can extend our approach to also maintain a "deletion" table, a table of records from the base table to be deleted. For FJView and SPView, deletions only result in rows to remove an analog to before when we have rows to insert. We can define the update pattern in the same way but modify our correction technique to subtract rather than add. Thus, we replace all of the additions in first column of Table 2 with subtractions (e.g., $\mathsf{ans}_{sum} - \Delta\mathsf{ans}_{sum}/\rho$). For AggView, the formulas presented in Table 2 apply without modifications. This is because the update patterns for AggView include the updated value of the records. By supporting `DELETE`, we can support `UPDATE` operations to the base table as well since an `UPDATE` can be modeled as an `INSERT` and then a `DELETE`.

8. RESULTS

First, we evaluate SVC on a synthetic benchmark dataset where we can control the data distribution and the update rate. We evaluate query accuracy, the efficiency of sampling, and query execution time. Next, we evaluate how SVC performs as we vary the complexity of the materialized view. Then, we evaluate the outlier indexing approach in terms of improved query accuracy and also evaluate the overhead associated with using the index. After evaluation on the benchmark, we present an end-to-end application of log analysis with a dataset from a video streaming company.

8.1 Experimental Setting

8.1.1 TPCD-Skew

Dataset Description and Views : 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 base dataset is 10GB corresponding to 60M records. Unless otherwise noted, we experiment with $z = 2$. 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:

SPView:

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

AggView:

```
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
```

FJView:

```
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
```

Queries on the views : We randomly generated 10,000 aggregate queries for each view, based on the query templates provided by the TPCD-Skew dataset. These queries are in the form of "SELECT $f(a)$ FROM View WHERE Condition(A)", where f is randomly selected from {`sum`, `count`, `avg` }, a is randomly selected from the aggregation attributes of the query templates, and $Condition(A)$ is randomly selected from the predicates of the query templates.

Experimental Platform : All of our experiments for the TPCD-Skew dataset are run on a single r3.large Amazon EC2 node (2x Intel Xeon E5-2670, 15.25 GB Memory, and 32GB SSD Disk) with a MySQL version 5.6.15 database. With MySQL, we create these views as tables. We construct a separate table of updates, and then measure the time needed to propagate the necessary updates (forming the delta table and writing the updates) to the views. In evaluating FJView, we only insert records into the lineitem table and create an index on all of the dimension tables. For AggView, we have an index on the group-by key of sampled materialized view. Note, that we do not assume there are any indices on the newly inserted records. We use the MySQL query profiling feature to isolate the cost of view maintenance in both the delta view phase and the refresh phase.

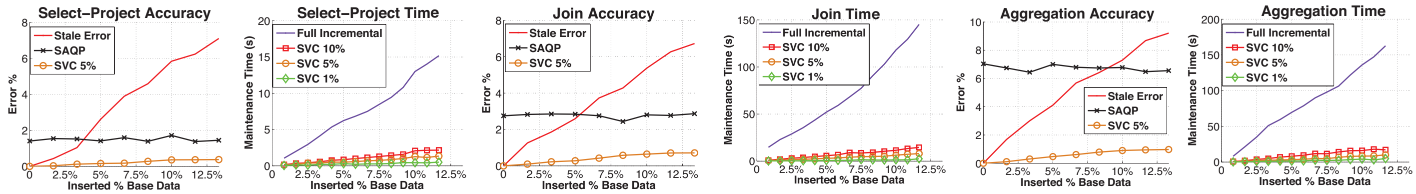

Figure 4: For each of the three views listed above, we plot the accuracy for 5% samples compared to SAQP and a stale baseline. We also plot the maintenance time for each of the views in comparison to full incremental maintenance for a 1%, 5% and 10% sample.

8.1.2 *Conviva*

Dataset Description : Conviva is a video streaming company and we evaluated our approach on user activity logs [**?**]. We experimented with a 1TB dataset of these logs which formed a single base table. With this dataset, there was a corresponding dataset of analyst SQL queries on the log table. Using the query dataset, we constructed 67 aggregation views, and 10,000 queries for each view using a similar method as the TPCD-Skew dataset. We used this dataset to evaluate the end-to-end accuracy and performance of the system in a real-world application.

Experimental Platform : We evaluated performance on Apache Spark 1.0.2 with a 20 node r3.large Amazon EC2 cluster. Spark supports materialized views through a distributed data structure called an RDD [37]. There is a SQL interface which transforms the RDD's using Map-Reduce chains. The RDD's are immutable, thus requiring significant overhead to maintain. As Spark does not have support for indices, we rely on partitioned joins for incremental maintenance of the aggregation views. We partitioned the aggregation views by group-by key, and joined the delta table with the aggregation view.

8.2 Accuracy

8.2.1 *Update Rate and Accuracy*

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 500K (0.8% base data) records to a final count of 8M 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 8M records were inserted the staleness was 7%. Furthermore, in 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.

8.2.2 *Sample Size and Accuracy*

Next, we explore how sample size affects query results. We inserted 5M (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.

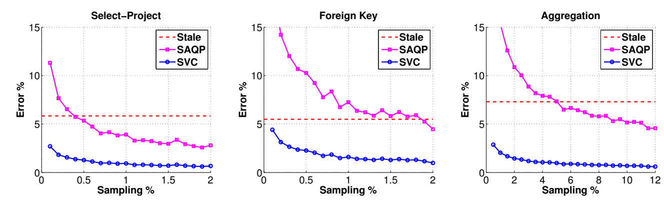

Figure 5: As we increase the sample size, the relative error goes down following a $\frac{1}{\sqrt{samplesize}}$ rate. SAQP also follows the same rate so the lines will never cross by only changing the sample size.

8.2.3 *Distribution of Query Error*

In our earlier experiments, we presented the average error for the queries on the views. We found that for inserted 5M 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. In the right figure, each green point corresponds to a query with the y-axis as SVC estimation error and the x-axis as staleness. Even though some queries are more than 60% stale, our corrections reduce the relative error to less than 20%.

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 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.

8.3 Efficiency

8.3.1 *Sampling Efficiency*

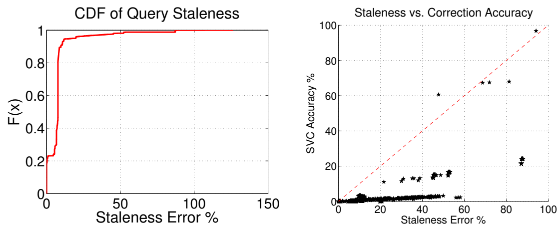

Figure 6: While the mean error is 9%, 5% of queries have more than 20% relative error. However, in the second plot, we show a scatter plot of staleness (x-axis) and a query correction accuracy (y-axis). Each black point corresponds to a single query. Points above the dotted line show that we improved query accuracy.

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. 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.

8.3.2 Overhead of Query Correction

In Section 5, we analyzed the query execution times. We found that while for SPView and FJView we have a reduced query time in comparison to full incremental maintenance, we can make no such guarantee for AggView. In this experiment, we evaluated the query execution time of SVC compared to SAQP, full incremental maintenace (query on the fresh view), and no maintenance (query on the stale view Since SPView and FJView are similar, we only evalute quer execution on FJView. For each of the views, we insert 5M records (8% of Base Data) and evaluate the average quer execution time; that is over the 10,000 generated queries th average time to return a result. For the aggregation view the insertions were such that 50% of the insertions update existing rows and 50% corresponded to new rows. In Figure 7, we show the results of the experiment. SAQP, which avoids querying the entire view, is much faster than both, however has as seen in the previous experiments has lower accuracy. For the AggView, it has a mean error of 6.6% compared to SVC's 0.8%. We also highlight that the introduced overhead for correction is orders of magnitude less than the maintenance time seen in the previous sections. For gView, full incremental maintenance requires 106.3 sec while 5% SVC only requires 5.5 seconds, and in compa the correction 0.08 seconds.

8.3.3 View Complexity

In our maintenance time experiments, we showed 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)
```

Figure 7: We compare the query execution time for SVC, SAQP, Full Maintenance, and No Maintenance. Since, we use a sample to correct query results on the stale view, we always require a query on the stale view. For FJView and SPView, we do not increase query execution time with our correction. However, for AggView, we introduce a small overhead.

We set the sample size to 5%, insert 5M 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 more 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 l_shipdate key, the cost savings were smaller 5.5x. These experiments emphasize that when views are large and complex, we can have significant improvements.

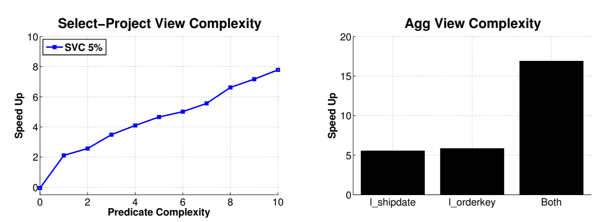

Figure 8: As the views definitions become more complex our approach gives increasing gains. For simplest view (selection no predicate), the overhead is small (5%).

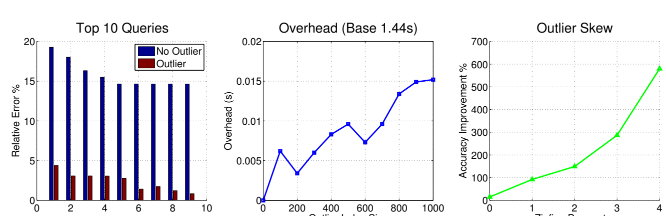

Figure 9: Outlier indexing greatly improves accuracy of our approach especially in skewed datasets for a small overhead of building the index and ensuring those rows are in the view.

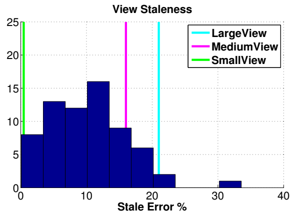

Figure 11: We chose three views that represented the distribution of all views well in terms of both accuracy. We plot the staleness of each of the 67 views and mark the three views in our experiment.

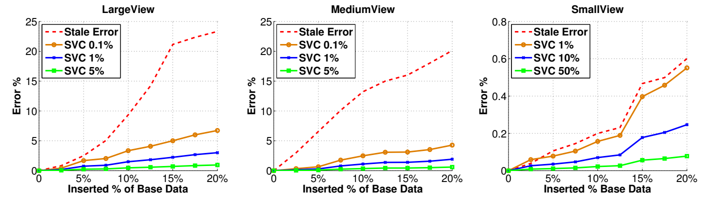

Figure 12: We compare average query staleness to our approach for a variety of sample sizes. For the larger views, we find that small sample sizes can give accurate results. Larger sample sizes are needed for accurate estimates in SmallView.

8.4 Outlier Indexing

We use a 5% sample size and 5M 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 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. 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 and 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 $z = 4$.

8.5 Application: Log Analysis For Conviva

We used a 1TB 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 a 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.

Since we evaluate accuracy in terms of query accuracy, we did not evaluate accuracy on the full 1TB dataset since it would have been prohibitive to execute thousands of queries for different sample sizes. Instead, for evaluating accuracy we used a 7GB subset of the logs corresponding to 5 days of data. However, for the scalability experiments, we use the full 1TB dataset. We present results in terms of relative sizes in comparison to the base table.

8.5.1 *Representativeness of the Views*

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 1.1GB (15%) of records to the table. Since the query accuracy is a function of the data distribution and staleness In Figure 11, we plot the average staleness of each of the views and mark the three views. While queries on LargeView and MediumView are 21% and 16% stale respectively, queries on SmallView are 0.5%. For the rest of our experiments, we use SmallView as the counter-example since the results are not affected much by the updates. Even so, we still show in subsequent experiments that we can achieve more accurate query results and save on maintenance time.

8.5.2 *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 1.4GB (20%) of records to the table. We built an outlier index on all of the attributes that represent time intervals or rates. Figure 12, 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.

8.5.3 *Performance in Conviva*

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. We derived the views from a 800GB base table and inserted records in approximately 20GB increments. In Figure 10, 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 120GB of inserted records (15%) acheives 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 recalcuation. 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.

9. 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], transac-

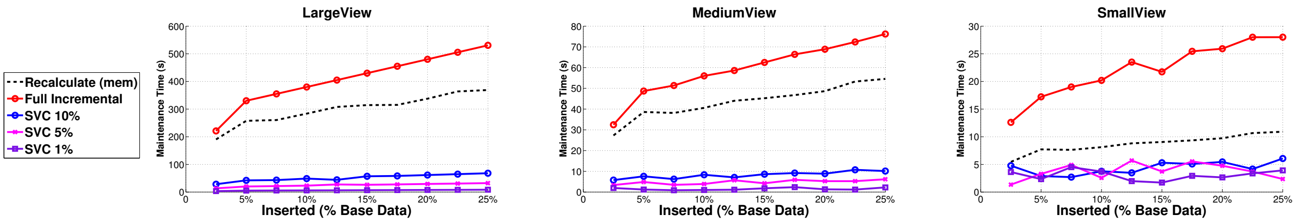

Figure 10: As before, we compare maintenance times for a variety of sample sizes and full incremental maintenance. We further include a comparison with view recalculation as our experimental platform, Spark, does not support indices or selective writes.

tion [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 steaming community has also studied the view maintenance problem [2,13,15,16,20]. In Spark Streaming, Zaharia et al. studied how they could exploit in-memory materialization [38], and in MonetDB, Liarou et al. 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 our approach is more accurate than SAQP 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 related problem to query correction which they called compensation-based query processing [33]. This work was applied in the context of concurrency control and did not consider sampling or materialization.

10. CONCLUSION AND FUTURE WORK

In this paper, we propose a new approach to the staleness problem in materialized views. We demonstrate 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 evaluate this approach on a single node and in a distributed environment and find that SVC can correct stale query results with orders of magnitude less cost than full incremental maintenance.

Our results are promising and suggest many avenues for future work. In particular, we are interested in deeper exploration of the multiple view setting. Here, given a storage constraint and throughput demans, we can optimize sampling ratios over all views. We are also interested in the possiblity of sharing computation between materialized views and maintenance on views derived from other views. 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.

11. 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.