Stale View Cleaning: Getting Fresh Answers from Stale Materialized Views

Sanjay Krishnan, Jiannan Wang, Michael J. Franklin, Ken Goldberg, Tim Kraska [†]
UC Berkeley, [†]Brown University
{sanjaykrishnan, jnwang, franklin, goldberg}@berkeley.edu
tim_kraska@brown.edu

ABSTRACT

Materialized views (MVs), stored pre-computed results, are widely used to facilitate fast queries on large datasets. When new records arrive at a high rate, it is infeasible to continuously update (maintain) MVs and a common solution is to defer maintenance by batching updates together. Between batches the MV becomes increasingly stale with incorrect, missing, and superfluous rows leading to increasingly inaccurate query results. We propose Stale View Cleaning (SVC) which addresses this problem from a data cleaning perspective. We take inspiration from recent results in data cleaning which combine sampling and cleaning for accurate query processing. In SVC, we efficiently clean a sample of rows from a stale MV, and use the clean sample to compute a query result correction to compensate for the dirtiness. While approximate, the corrected query results reflect the most recent data. SVC supports a wide variety of materialized views and aggregate queries on those views with optimality for SUM, COUNT, AVG. 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. SVC complements existing deferred maintenance approaches by giving accurate and bounded query answers between maintenance. We evaluate our method on a real dataset of workloads from the TPC-D benchmark and a real video distribution application. Our experiments confirm our theoretical results: (1) cleaning an MV sample is more efficient than full view maintenance, (2) the corrected results are more accurate than using the stale MV, and (3) SVC can be efficiently integrated with deferred maintenance.

1. INTRODUCTION

Materialized views (MVs), stored pre-computed results, are a well-studied approach to speed up queries on large datasets [10,25,33]. During the last 30 years, the research community has thoroughly studied MVs for traditional query processing and recently for more advanced analytics based on linear algebra and machine learning [37,53].

However, when the underlying data is changed MVs can become *stale*; the pre-computed results do not reflect the recent changes to the data. One solution would be to recompute the MV each time a change occurs; however, in many cases, it is more efficient to incrementally update the MV instead of recomputation. There has been substantial work in deriving incremental updates (incremental maintenance) for different classes of MVs and optimizing their execution [10]. For frequently changing tables even incremental maintenance can be expensive since every update to the base data requires updating all the dependent views. In many important applications, such as summary statistics from user activity logs, new records arrive at a fast rate and data are often distributed across multiple machines making incremental maintenance infeasible. As a result, in production environments, it is common to defer view maintenance [10,12,54] so that updates can be batched together to

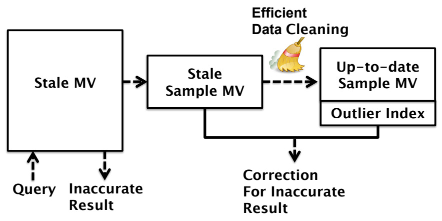


Figure 1: Deferred maintenance can lead to stale MVs which have incorrect, missing, and superflous rows. In SVC, we pose this as a data cleaning problem and show that we can use a sample of clean (up-to-date) rows from an MV to correct inaccurate query results on the stale view. We also devise an outlier indexing technique for more accurate corrections in skewed datasets.

amortize overheads and can be scheduled at times of low system utilization (e.g., nightly).

While deferring maintenance has benefits, a disadvantage is that MVs become increasingly stale between maintenance periods. As a result, queries using those MVs can return incorrect answers. The problem of stale MVs parallels the problem of dirty data studied in data cleaning [44]. As with dirty data, a stale MV has incorrect, missing, or superfluous rows. In this work, we explore how answering queries on a stale MV can be formalized as a data cleaning problem.

Data cleaning has been studied extensively in the literature (e.g., see Rahm and Do for a survey [44]) but increasing data volumes have led to development of new, efficient sampling-based approaches for coping with dirty data. In our prior work, we developed the SampleClean framework for scalable aggregate query processing on dirty data [48]. Since data cleaning is often expensive, we proposed cleaning a sample of data using this sample to improve the results of aggregate queries on the full dataset. Since stale MVs are dirty data, an approach similar to SampleClean raises a new possibility, namely, we can use a sample of "clean" rows in the MV to return more accurate query results.

In this paper, we propose Stale View Cleaning (SVC), which uses applies data cleaning to stale MVs. SVC (Figure 1) provides a framework that efficiently cleans a sample of rows from a stale MV resulting in a uniform sample of "clean" (up-to-date) rows. After cleaning, to answer a query, we can analyze how that sample has changed because of the cleaning, and calculate a correction that compensates for the data error. The query results from this procedure, while approximate, are up-to-date in the sense that they reflect the most recent data. The approximation error due to sampling is more manageable than staleness because: (1) the uniformity of sampling allows us to apply theory from statistics such as the Central Limit Theorem to give tight bounds on approximate results, and (2) the approximate error is parameterized by the sample size which the user can control trading off accuracy for computation. SVC is complementary to existing deferred maintenance approaches. When the MVs become stale between maintenance cycles, we apply SVC for a far smaller cost than having to maintain the entire view but still get approximate, up-to-date answers.

To summarize, our contributions are as follows: (1) we formalize maintenance of a sample MV as a data cleaning operation allowing us to apply query processing similar to SampleClean, (2) we propose efficient techniques to implement the view-cleaning operation on a sample of the view, (3) we devise a query processing approach to answer queries accurately using the sample of up-to-date data, (4) we propose an outlier index to increase the accuracy of the approach for power-law, long-tailed, and skewed distributions, and (5) we evaluate our approach on real and synthetic datasets confirming that indeed sampling can reduce view maintenance time while providing accurate query results.

The paper is organized as follows: In Section 2, we give the necessary background for our work. Next, in Section 3, we formalize the problem. In Sections 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 discuss Related Work in Section 9. In Section 10, we discuss the limitations and new opportunities of our approach, and we present our Conclusions in Section 11.

2. BACKGROUND

In this section, we briefly overview our prior work on sampling-based data cleaning and the new challenges in the MV setting.

2.1 Materialized View Maintenance

Views define logical relations which can be queried instead of physical base relations. MVs are a class of views that are precomputed and stored (i.e materialized). Any form of pre-computed, derived data encounters the problem of staleness when the physical base relations update.

One approach to this problem is to recompute the materialized view every time there are updates to the base tables. However, this approach is very inefficient if updates to the data generally have small or sparse effect on the MV. A contrasting approach is incremental view maintenance (IVM), where rows in the MV are incrementally updated based on the updates to the base table. Incremental maintenance of MVs has been well studied; see [10] for a survey of the approaches. At a high-level, incremental maintenance algorithms typically consist of the following steps: (1) maintain a cache of insertions and deletions for each physical base table, then using the view definition derive a *change propagation formula* in terms of the set of insertions and deletions, and finally apply the formula to the view. For a variety of view types, these rules are described in detail in [29,30].

In real-world systems, for large datasets or fast data update rate, it may not always be feasible to maintain MVs immediately. Therefore, deferring maintenance (periodically or adaptively) is an alternative and often preferred solution. The main insight of deferral is to avoid maintaining the view immediately and to schedule an update at a more convenient time. In deferred maintenance approaches, the user often accepts some degree of staleness for additional flexibility in scheduling. By using sampling, we give the user access to a new trade-off space between immediate (or close to immediate, i.e., mini-batch) maintenance and long-periodic maintenance.

2.2 SampleClean [48]

SampleClean is a framework for scalable aggregate query processing on dirty data. Traditionally, data cleaning has explored expensive, up-front cleaning of entire datasets for increased query accuracy. Those who were unwilling to pay the full cleaning cost avoided data cleaning altogether. We proposed SampleClean to add an additional trade-off to this design space by using sampling. The problem of high computational costs for accurate results mirrors the challenge faced in the MV setting with the tradeoff between immediate maintenance (inexpensive and stale) and deferred maintenance (expensive and up-to-date).

SampleClean has three parts: (1) sampling, (2) data cleaning, and (3) query result estimation. First, SampleClean creates a sample of dirty data (which may have erroneous values or duplicated records). Then, the framework applies a data cleaning procedure to the sample. Finally, when users query the dataset, the framework uses the cleaned sample to extrapolate clean query results. In the framework, the main challenge was that data cleaning can potentially change the statistics of a sample and the queries need to compensate for those effects.

The SampleClean work showed that there were two contrasting approaches to query processing on a sample of cleaned data. We could (1) clean the sample first and then run the query on the sample, or (2) look at the difference between the clean and dirty samples and calculate a correction to correct an existing dirty query result. Approach (1) is similar to those studied in the Approximate Query Processing (AQP) literature [4,27,40]. Approach (2), which we called NormalizedSC, outperformed (1) in datasets where data error was small or sparse. In the MV setting, we compare these approaches (see Section 8).

2.3 New Challenges

Stale MVs are dirty data, and we can think of view maintenance as a data cleaning operation. But note that we cannot simply extend SampleClean to a sample of a stale MV, and there are many significant new challenges that we have to address.

The first challenge relates to the data cleaning model. In SampleClean, we modeled data cleaning as a row-by-row transformation of relation. This transformation was a user-specified "black box" that operated on each row, and we applied this to a sample of dirty data. However, in a MV setting, we found that the "black-box" data cleaning model does not suffice. First, applying incremental maintenance to a sample may not be any more efficient than applying it to a whole MV if it does unnecessary computation (i.e., calculates updates for rows outside of the sample). In this work, we analyze the relational algebra of a MV maintenance procedure and propose a series of rules to optimize maintaining a sample. Second, stale MVs have not only incorrect rows, but also rows that are missing from the "dirty" view or conversely need to be deleted. We need to extend the row-by-row transformation model to handle this type of error.

In terms of supported queries, SampleClean mainly focused on three common aggregates: `sum`, `avg`, and `count` queries. In this work, we explore a broader set of aggregate queries such as `median` and design a bootstrap algorithm to bound the approximation error. We found that the queries that we studied before (`sum`, `avg`, and `count`) were actually a special case where our estimates are optimal.

Sampling is particularly sensitive to variance of the dataset, and large outliers can significantly reduce query accuracy, so in this work, we extend the query processing with an index of outliers. We show that with only a single pass of the base data, we can index records that exceed some threshold. Then, for all rows in an MV that are dependent on an indexed record, we can ensure that they are included in our sample MV. We found that empirically this approach leads to significant accuracy improvements in skewed datasets.

2.4 Example: Server Log Analysis

To illustrate our framework in the upcoming sections, we use the following running example which is a simplified schema of one of our experimental datasets. 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)
```

The Log table stores each visit to a specific video with primary key (sessionId), a foreign-key to the Video table (videoId), latency of the visit (responseTime), and the browser used to access the video (userAgent). The Video stores each video with the primary key (videoId), the video title (title), and the video duration (duration). Though SVC supports insertions, deletions, and updates to base data (See Section 3.1 for details), for clarity in our example, we consider insertions into Log which is cached in a temporary table:

```
LogIns(sessionId, videoId, responseTime, userAgent)
```

3. FRAMEWORK OVERVIEW

We first define notation, terminology, and the problem setting that we address in this work. Then, we formalize the two main problems that SVC addresses: (1) materialized view maintenance as data cleaning and optimizing this data cleaning, and (2) query processing on stale MVs with a sample of up-to-date data. Finally, we overview the system architecture and discuss a numerical example of how this works in practice.

3.1 Notation

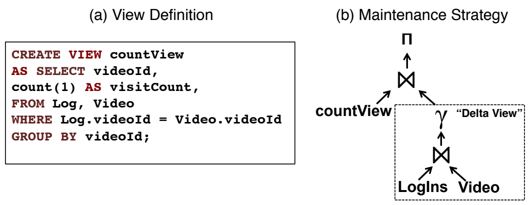
Materialized View: Let $\mathcal{D}$ be a database which is a collection of relations $\{R_i\}$. A *materialized view* S is the result of applying a *view definition* to $\mathcal{D}$. View definitions are composed of the following relational expressions:

- $\sigma_\phi(R)$: Selection selects all tuples r from R that satisfy the restriction $\phi(r)$.
- $\Pi_{a_1,a_2,...,a_k}(R)$: Generalized projection selects attributes $\{a_1, a_2, ..., a_k\}$ from R, allowing for new columns that are arithmetic transformations of attributes (e.g., $a_1 + a_2$).
- $\bowtie_{\phi(r1,r2)}(R_1, R_2)$: Join selects all tuples in $R_1 \times R_2$ that satisfy $\phi(r_1, r_2)$. We use $\bowtie$ to denote all types of joins even extended outer joins such as $⟕, ⟖, ⟗$.
- $\gamma_{f,A}(R)$: Apply the aggregate function f to the relation R grouped by the distinct values of A, where A is a subset of the attributes. The DISTINCT operation can be considered as a special case of the Aggregation operation.
- $R_1 \cup R_2$: Set union takes a union of the two sets.
- $R_1 \cap R_2$: Set intersection takes an intersection of the set.
- $R_1 - R_2$: Set difference.

The composition of relational expressions can be represented as a tree, which is called the *expression tree*. At the leaves of the tree are all of the *base relations* for a view. Each node of the tree is the result of applying one of the above relational expressions to a relation.

Staleness: We denote the set of insertions to a relation R_i as ΔR_i and deletions as ∇R_i. An "update" to a relation can be modeled as a deletion and then an insertion. A view S is considered *stale* when there exist insertions and deletions to its base relations.

Maintenance: There may be multiple ways (e.g., incremental maintenance or recomputation) to maintain a view S, and we denote the up-to-date view as S'. We formalize the procedure to maintain the view as a *maintenance strategy* $\mathcal{M}$. A maintenance strategy is a relational expression the execution of which will update the view S to produce S'. It is a function of the database $\mathcal{D}$, the stale view S, and all the insertion and deletion relations $\{\Delta R_i\} \cup \{\nabla R_i\}$. In this work, we consider maintenance strategies composed of the same relational expressions as materialized views described above.



```
CREATE VIEW countView
AS SELECT videoId,
count(1) AS visitCount,
FROM Log, Video
WHERE Log.videoId = Video.videoId
GROUP BY videoId;
```

Figure 2: For our example, we represent the expression tree of the maintenance strategy. We first calculate a delta view using the new insertions and then join this view with the old view.

Example: To make the formalization of a maintenance strategy concrete, we show an example MV in Figure 2 based on our running example dataset. Our example view joins the Log table with the Video table and counts the visits for each video grouped by videoId. If new records have been added to Log, then the expressions are needed to update the view (Figure 2):

1. Create a "delta view" by applying the view definition to LogIns. That is, calculate the count per video on the new logs.
2. Take the full outer join (equality on videoId) of the "delta view" and the stale view.
3. Apply the generalized projection operator to increment visitCount (adding the delta view visitCount and the stale visitCount where NULL is treated as zero).

3.2 Sampling

In this work, we focus on uniform samples of the rows in MVs. We define a sampling ratio $m \in [0, 1]$ and for each row in a view S, we include it into a sample with probability m. We use the "hat" notation (e.g., $\hat{S}$) to denote sampled relations.

While, uniform sampling supports a wide variety of query types, it may have issues with queries with highly selective predicates. Stratified sampling has been proposed to mitigate this problem as in the BlinkDB project [4]. However, this requires that we know our query workload in advance. In this paper, we do not discuss stratified sampling and will explore this further in future work.

3.3 Problem Statements

3.3.1 View Maintenance as Data Cleaning

We formalize the problem of correcting staleness as a data cleaning operation so we can apply our data cleaning approach. In the unsampled case, $\mathcal{M}$ defines a data cleaning operation. If we are given a materialized view S and we know the base relations have had insertions and deletions, then there are three possible types of error: (1) a row in S needs to be updated, (2) a row in S needs to be deleted, and (3) new row needs to be inserted into S. Applying $\mathcal{M}$ removes these errors making the view "clean".

However, now suppose we have a sampled view $\hat{S}$, simply applying updates to the rows in the sample may not suffice. If new rows need to be inserted into S, those will never be represented in the sample violating our uniform sampling. Thus, we define cleaning in the following way: suppose we have a stale uniform sample $\hat{S}$, cleaning this sample should give us $\hat{S}'$ a uniform sample of the up-to-date view S' with the same sampling ratio. Formally, this can be represented as the following operations: (1) if an update is needed, update the row, (2) if a row needs to be deleted, delete the row, and (3) for all new rows that need to be inserted into the view S insert a random sample of ratio m.

Due to the insertions, the defined data cleaning on a sample does not necessarily give a unique $\hat{S}'$, so the next question is how to formalize the link between $\hat{S}$ and $\hat{S}'$. To link a corresponding stale sample (dirty data) and up-to-date sample (clean data), we define the following property:

Definition 1 (Correspondence). *$\hat{S}'$ and $\hat{S}$ are uniform samples of S' and S, respectively. We say $\hat{S}'$ and $\hat{S}$ correspond if and only if:*

- *For every row r in $\hat{S}$ that required a delete, $r \notin \hat{S}'$*
- *For every row r in $\hat{S}$ that required an update to r', $r' \in \hat{S}'$*
- *For every row r in $\hat{S}$ that was unchanged, $r \in \hat{S}'$*
- *For every row r in S but not in $\hat{S}$, $r \notin \hat{S}'$*

The goal of SVC is to efficiently produce a corresponding up-to-date sample from a stale one thus cleaning the sample. In the first component of SVC (Section 4), we take as input a uniform sample of a stale view $\hat{S}$, a maintenance strategy $\mathcal{M}$, and a set of updates $\{\Delta R_i\} \cup \{\nabla R_i\}$. We return $\hat{S}'$, a clean uniform sample (a uniform sample of S') that satisfies the correspondence property with $\hat{S}$.

3.3.2 Query Correction

In the query correction phase, we take a query result on a stale view and use the up-to-date sample to compensate for the staleness. Given a query q which has been applied to the stale view $q(S)$ giving a stale result. Our query correction component takes the two corresponding samples $\hat{S}'$ and $\hat{S}$, and calculates a correction to $q(S)$.

Like similar restrictions in other sample-based systems [3], there are restrictions on the queries q on the view that we can answer. In the SampleClean work, we focused on `sum`, `count`, and `avg` queries of the form[1]:

```
SELECT f(a) FROM View WHERE Condition(A);
```

In this work, we expand the scope of the query processing, and consider general non-nested aggregate queries with predicates.

We also consider correcting stale non-nested select queries of the following form with predicates:

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

As with all sample estimates, the accuracy increases with sample size, thus less selective predicates lead to more accurate results.

3.4 System Architecture

We summarize the system architecture in Figure 1 in our introduction. SVC reduces the cost of view maintenance making it feasible to apply in resource-constrained settings where frequent maintenance was once impossible. SVC works in conjunction with existing deferred maintenance, periodic maintenance, or periodic re-calculation solutions. 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 answers for some queries.

3.5 Example Application

Returning to our example countView, suppose a user wants to know how many videos have received more than 100 views.

```
SELECT COUNT(1) FROM countView
WHERE visitCount > 100;
```

Let us suppose the initial query result is 45. There now have been new log records inserted into the Log table making the old result stale. For example, if our sampling ratio is 5%, that means for 5% of the videos (distinct videoId), we update just the view counts of those videos. Suppose 2 videos have changed their counts from less than 100 to greater than 100. From this sample, we extrapolate that 40 new videos throughout the view should now be included in the count. This means that we should correct the old result by 40 resulting in the estimate of $45 + 40 = 85$.

[1]For simplity, we exclude the group by clause for all queries in the paper, as it can be modeled as part of the Condition.

4. EFFICIENTLY CLEANING A SAMPLE

In the previous section, we formalized the procedure of taking a uniform sample of "dirty" rows $\hat{S}$, and "cleaning" it to produce a corresponding uniform sample of the up-to-date view $\hat{S}'$. This procedure is not unique and there are both efficient and inefficient ways to accomplish this. For example, a naive solution to derive a sample $\hat{S}'$ is to just apply the maintenance strategy to S and then sample the result. However, this does not save any computation.

Ideally, we want to integrate the sampling into the maintenance strategy $\mathcal{M}$ so that expensive operators need not operate on the full data. This efficient sampled maintenance strategy is our optimized data cleaning operation.

4.1 Uniform MV Sampling

For a sampling ratio m, we call a sample view $\hat{S}'$ a uniform sample of S', under the following condition:

Definition 2 (Uniform Sample). *We say the relation $\hat{S}'$ is a* uniform sample *of S' if*

(1) $\forall s \in \hat{S}' : s \in S'$; (2) $Pr(s_1 \in \hat{S}') = Pr(s_2 \in \hat{S}') = m$

A traditional "coin-flip" sampling algorithm is not suited for this property as it is known that such sampling commutes very poorly with many relational operations such as joins and aggregates [8]. Recall, the view in our example countView. Suppose, we sampled from the base relation Log, and then applied the view definition to the sample to form the "delta view". The "delta view" would have a mix of missing videos (videoId is not in the sample) and rows with incomplete aggregates (not all of the videos with videoId are in the sample). However, this is not what we require since it is not a uniform sample of the rows in the view.

To get a uniform sample of a view, the main problem is that for every row sampled in the view, our sampling technique needs to include all of the rows in sub-expressions that contribute to its materialization. Achieving this requires a definition of lineage; traceable, unique identification for rows.

4.2 Identification With Row Lineage

Lineage has been an important tool in the analysis of materialized views [15] and in approximate query processing [52]. We recursively define a set of consistent primary keys for all nodes in the expression tree:

Definition 3 (Primary Key). *For every relational expression R, we define the primary key of every expression to be:*

- *Base Case: All relations (leaves) must have an attribute p which is designated as a primary key. That uniquely identifies rows.*
- *$\sigma_\phi(R)$: Primary key of the result is the primary key of R*
- *$\Pi_{(a_1,...,a_k)}(R)$: Primary key of the result is the primary key of R. The primary key must always be included in the projection.*
- *$\bowtie_{\phi(r1,r2)} (R_1, R_2)$: The primary key of the result is the union of the primary keys of R_1 and R_2.*
- *$\gamma_{f,A}(R)$: The primary key of the result is the group by key A (which may be a set of attributes).*
- *$R_1 \cup R_2$: Primary key of the result is the primary key of R*
- *$R_1 \cap R_2$: Primary key of the result is the primary key of R*
- *$R_1 - R_2$: Primary key of the result is the primary key of R*

This definition of a primary key for a relational expression, allows us to trace the primary key through the expression tree.

4.3 Hashing Operator

If we have a deterministic way of mapping a primary key defined in the previous subsection to a sample, we can also ensure that all contributing rows are also sampled. To achieve this we use a hashing procedure. Let us denote the hashing operator $\eta_{a,m}(R)$. For all tuples in R, this operator applies a hash function whose range is $[0, 1]$ to primary key a (which may be a set) and selects those records with hash value less than or equal to m. If the hash function is sufficiently uniform, then the condition $h(a) \le m$ is true for close to a fraction m of the rows.

To achieve the performance benefits of sampling, we push down the hashing operator through the query tree. The further that we can push η down the expression tree, the more operators can benefit from the sampling. However, it is important to note that for some of the expressions, notably joins, the push down rules are more complex. It turns out in general we cannot push down even a deterministic sample through those expressions. We formalize the push down rules below:

DEFINITION 4 (HASH PUSHDOWN). *Let a be a primary key of a materialized view. The following rules can be applied to push $\eta_{a,m}(R)$ down the expression tree of the maintenance strategy.*

- $\sigma_\phi(R)$*: Push η through the expression.*
- $\Pi_{p,[a_2,\ldots,a_k]}(R)$*: Push η through if a is in the projection.*
- $\bowtie_{\phi(r1,r2)} (R_1, R_2)$*: Blocks η in general. There are special cases below where push down is possible.*
- $\gamma_{f,A}(R)$*: Push η through if a is in the group by clause A.*
- $R_1 \cup R_2$*: Push η through to both R_1 and R_2*
- $R_1 \cap R_2$*: Push η through to both R_1 and R_2*
- $R_1 - R_2$*: Push η through to both R_1 and R_2*

In special cases, we can push the hashing operator down through joins. Given the hash function $\eta_{a,m}(R)$:

Equality Join: If the join is an equality join and a is one of the attributes in the equality join condition $R_1.a = R_2.b$, then η can be pushed down to both R_1 and R_2. On R_1 the pushed down operator is $\eta_{a,m}(R_1)$ and on R_2 the operator is $\eta_{b,m}(R_2)$. This case often happens near the top of maintenance strategy expression tree where there is a equality outer join on the primary key of the stale view and a "delta view".

Foregin Key Join: If we have a join with two foreign-key relations R_1 (fact table with pk a) and R_2 (dimension table with fk b) and we are sampling the key a or (a, b), then we can push the sampling down to R_1. This is because we are guaranteed that for every $r_1 \in R_1$ there is only one $r_2 \in R_2$.

(Semi/Anti)-Join: Similarly, if we are hashing the primary key of a semi-join, we can always push η down R_1. For anti-joins we can push η down because we can rewrite the node as $R_1 - (R_1 \ltimes R_2)$ and apply the pushdown rules for set difference and Semi-Joins.

4.4 Corresponding Samples

We showed that we can optimize the sample cleaning procedure by using push-down rules. This gives us an expression to clean a sample of rows in the up-to-date view S'. When the insertion and deletion relations $\{\Delta R_i\} \cup \{\nabla R_i\}$ are not empty, we can use this expression to propagate changes to our sample. Thus, this procedure "cleans" the dirty sample S.

One benefit of deterministic hashing is that we get the Correspondence Property (Definition 1) for free.

THEOREM 1 (HASHING CORRESPONDENCE). *Suppose we have S which is the stale view and S' which is the up-to-date view. Both these views have the same schema and a primary key a. Let $\eta_{a,m}$ be our hash function that applies the hashing to the primary key a.*

$$\hat{S} = \eta_{a,m}(S), \ \hat{S}' = \eta_{a,m}(S')$$

Then, two samples $\hat{S}'$ and $\hat{S}$ correspond.

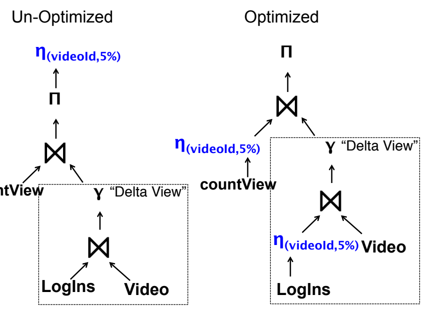


Figure 3: Applying the rules described in Section 4.3, we illustrate how to optimize the sampling of our example maintenance strategy.

PROOF SKETCH. Since the primary keys are key consistent between $\hat{S}'$ and $\hat{S}$, included and excluded rows are preserved by the hashing. There are four conditions for correspondence:

- 1. For every row r in $\hat{S}$ that required a delete, $r \notin \hat{S}'$
- 2. For every row r in $\hat{S}$ that required an update, $r \in \hat{S}'$
- 3. For every row r in $\hat{S}$ that was unchanged, $r \in \hat{S}'$
- 4. For every row r in S but not in $\hat{S}$, $r \notin \hat{S}'$

Condition 1 is satisfied since if r is deleted, then $r \notin S'$ which implies that $r \notin \hat{S}'$. Condition 2 and 3 are satisfied since if r is in $\hat{S}$ then it was sampled, and then since the primary key is consistent between S and S' it will also be sampled in $\hat{S}'$. Condition 4 is just the converse of 2 and 3 so it is satisfied. □

We will use this property in the Section 5 to get estimates for queries on the materialized view.

4.5 Example

We will illustrate our proposed approach on our example view countView (Figure 2). The maintenance strategy of this view is described in the previous section.

We first use the rules described in Section 4.2 to get the primary key of the view. The primary key for base relations Log and Video are sessionId and videoId, respectively. If we move up the tree in Figure 2, the first expression in the maintenance strategy is a join making the primary key of that expression (sessionId, videoId). Then, next there is an aggregation which groups by videoId making that the primary key of the MV.

We can apply our hash operator to this key, and use the pushdown rules described in Section 4.3 to efficiently sample the maintenance strategy. In Figure 3, we illustrate the pushdown process. At the top of the expression tree is the MV, whose primary key is videoId. We start by applying the hashing operator to this key. The next operator we see in the expression tree is a projection that increments the visitCount in the view, and this allows for push down since videoId is in the projection. The second expression is a hash of the equality join key which merges the aggregate from the "delta view" to the old view allowing us to push down on both branches of the tree. On the left side, we reach the stale view so we stop. Since the stale view does not change, we can calculate the sample of the stale view once (e.g., during periodic maintenance). On the right side, we reach the aggregate query (count) and since videoId is in group by clause, we can push down the sampling. Then, we reach another point where we hash the equality join key allowing us to push down the sampling to the relation LogIns and Video.

In terms of increased efficiency, since both the aggregation and joins are "above" the sampling operator, they require less computation and less memory.

5. CORRECTION QUERY PROCESSING

In this section, we discuss how to correct stale query results using the two corresponding samples $\hat{S}$ and $\hat{S}'$. Our data cleaning perspective allows us to look at each dirty row in an MV and what cleaning was applied to clean it (insert, delete, or update). We can use this analysis to calculate corrections that compensate for the effect of staleness of aggregate queries. The intuition is to take a point-wise difference of $\hat{S}'$ and $\hat{S}$ which is challenging in the presence of missing data. We first present our extensions to the existing SampleClean queries: `sum`, `count`, and `avg`. Then, we discuss how to extend this framework to other aggregate functions. We summarize these results in Table 1, where we list common aggregation queries and describe

5.1 SUM, COUNT, and AVG

NormalizedSC corrects a dirty query result (for `sum`, `count`, and `avg` queries), by taking a sample of dirty data, applying data cleaning, and estimating a compensation for the dirtiness from the sample. It then calculates the row-by-row difference for each row in the sample between dirty and clean (i.e., how much did cleaning change the data). We showed that the result of applying the query to the set of differences can be interpreted as a "correction" for dirty query results. Corrections differ from the traditional AQP approach which would be to apply a query directly to the up-to-date sample view. However, in prior work, our corrections only considered data error that updated rows, not those that required new rows to be inserted or deleted. In the MV setting, it is possible that there are rows in the new view and the old view that do not exist in the other.

In Definition 1, we formalized a notion of correspondence between two samples. We use this property to handle this problem of missing rows from either side of the difference. We define a new operator $\dot{-}$, called correspondence subtract, which will allow us to apply NormalizedSC to such samples.

DEFINITION 5 (CORRESPONDENCE SUBTRACT). *Given an aggregate query, and two corresponding relations R_1 and R_2 with the schema $(a_1, a_2, ...)$ where a_1 is the primary key for R_1 and R_2, and a_2 is the aggregation attribute for the query. $\dot{-}$ is defined as a projection of the full outer join on equality of $R_1.a_1 = R_2.a_1$:*

$$\Pi_{R_1.a_2 - R_2.a_2}(R_1 ⟗ R_2)$$

Null values $\emptyset$ are represented as zero.

To apply this operation, we rewrite the `sum` and `count` queries using a selection with a **case** statement, and use the primary key (pk) that we defined in the previous section. A case statement is defined as follows, we define pred$(*)$ to be 1 when the predicate is true and 0 when false.

For `sum`:

```
q(View) := SELECT pk, $a \cdot$ pred($*$) FROM View
```

and for `count`:

```
q(View) := SELECT pk, pred($*$) FROM View
```

We can use our correspondence subtract operator to get the point-wise difference: $d = q(\hat{S}') \dot{-} q(\hat{S})$. The definition of the correspondence subtract allows us to be agnostic to both insertions and deletions. For sampling ratio m, we can estimate the query correction for `sum` and `count`:

$\Delta \mathsf{ans}_{count,sum}$ = **SELECT** sum$(*)/m$ **FROM** d;

To apply the correction, we take the stale query result and add the correction:

$\mathsf{ans}'_{sum,count} \approx \mathsf{ans}_{sum,count} + \Delta\mathsf{ans}_{sum,count}$;

For the `avg` query, we can divide corrected results for `sum` and `count`:

$\mathsf{ans}'_{avg} = \frac{\mathsf{ans}'_{sum}}{\mathsf{ans}'_{count}}$;

In each of these formulas, all of the $\Delta\mathsf{ans}$ terms correspond to estimates, and these estimates can be bounded. In [48], we showed that the corrections for three queries can be rewritten as a sample mean. The basic idea is that 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. Refer to our prior work [48] for further details on how to bound these estimates with the Central Limit Theorem (CLT).

5.1.1 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 [14]. It turns out that the proposed corrections are the optimal strategy when nothing is known about the data distribution a priori.

THEOREM 2. *Suppose we have a set of real numbers X of size N. X defines an empirical (non-parametric) distribution as follows for a random draw from X takes on the value each $x \in X$ with probability $\frac{1}{N}$. For the family of non-parametric distributions, the sample mean is an MVUE of the expected value of X (the population mean). Thus, for `sum`, `count`, and `avg` queries, our estimate of the correction is optimal when no other information is known about the distribution.*

PROOF SKETCH. It is known that the sample mean is an MVUE for the population mean for an arbitrary distribution [45]; however we calculate a correction and we explore the optimality of this correction. In the first step of NormalizedSC, we rewrite `sum`, `count`, and `avg` as Select queries. This gives us a set of real numbers which defines our empirical distribution. Then, we apply our correspondence subtract operator and since point-wise subtraction is commutative with these three queries, we can re-write $\Delta\mathsf{ans}$ as a sample mean of the set of differences (where nulls are 0). Thus, $\Delta\mathsf{ans}$ is an MVUE for the correction and since the stale result is deterministic it does not affect the estimate. $\square$

The implication of this theorem is that there does not exist any other estimation algorithm for $\Delta\mathsf{ans}$ that has lower variance when nothing else is known about the data a priori. However, as we will see in Section 6, when we know additional information (e.g., outliers), we may be able to get better results. For example, we could imagine training a machine learning algorithm to predict how each row changes and apply this algorithm to answer aggregate queries; this theorem shows that for `sum`, `count`, and `avg` this would be suboptimal.

5.1.2 Correction vs. Direct Estimate

Correcting query results approximately is often more accurate than an AQP-style direct estimate. We can provide some mathematical intuition for this. For `sum`, `count`, and `avg`, our correction algorithm gives a confidence interval (via CLT) that is proportional to the variance of the *change* and inversely proportional to the sample size $\frac{\sigma_c^2}{k}$. On the other hand, an AQP approach would give us an estimate that is proportional to the variance of the up-to-date

Table 1: Query Result Bounds

Queries	Unbiased	Bounded Bias	Type of Bound
`sum`, `count`, `avg`	Yes	-	Optimal Analytical Via CLT
`histogram_numeric`, `corr`, `var`, `cov`	Yes	-	Empirical Via Bootstrap
`median`, `percentile`	No	Yes	Empirical Via Bootstrap
`max`, `min`	No	No	Loose Probability Bound via Cantelli's Inequality
`SELECT *`	Yes	Yes	Optimal bound on result size

data $\frac{\sigma^2_{S'}}{k}$. Since the change is the difference between the stale and up-to-date data, this can be rewritten as

$$\frac{\sigma^2_S + \sigma^2_{S'} - 2cov(S,S')}{k}$$

Therefore, a correction will have less variance when:

$$\sigma^2_S \leq 2cov(S,S')$$

This result shows that there is a point when updates to the stale MV are significant enough that direct estimates are more accurate. When we cross the break-even point we can switch from using corrections applying an AQP estimate. The AQP approach does not depend on $cov(S,S')$ which is a measure of how much the data has changed. Thus, we guarantee an approximation error of at most $\frac{\sigma^2_{S'}}{k}$. In our experiments (Figure 6(b)), we evaluate this break even point empirically.

5.2 General Aggregate Queries

In SampleClean, we proposed the NormalizedSC algorithm to give unbiased corrections to `sum`, `count`, and `avg`. We found that these queries are a special case of a broader class of aggregate queries; namely if a query has an unbiased sample estimate there also exists an unbiased correction. However, in general, these aggregate functions do not satisfy the optimality conditions of the previous section. The main condition that fails is the commutativity of the correspondence subtraction operation (i.e., $var(x-y) \neq var(x) - var(y)$).

These queries include: `histogram_numeric`, `corr`, `var`, `cov`, and the estimation approach is different. The general estimation procedure is the following:

1. Apply q to the stale sample,
2. Apply q to the up-to-date sample,
3. Apply q to the full stale view
4. Take the difference between (2) and (1) and add it to (3).

The implications of this are that any query that can be answered in prior work with SAQP (e.g., in BlinkDB [4]) in an unbiased way can also be answered with our approach.

Suppose, we have an aggregate query q and we apply the query to the stale view S. The query result is stale by c if: $c = q(S') - q(S)$.

LEMMA 1. *If there exists an unbiased sample estimator for q(S') then there exists an unbiased sample estimator for c.*

PROOF SKETCH. Suppose, we have an unbiased sample estimator $\bar{q}$ of q. Then, it follows that

$$\mathbb{E}\big[\bar{q}(\hat{S}')\big] = q(S')$$

If we substitute in this expression:

$$c = \mathbb{E}\big[\bar{q}(\hat{S}')\big] - q(s)$$

Applying the linearity of expectation:

$$c = \mathbb{E}\big[\bar{q}(\hat{S}') - q(s)\big]$$

□

Some queries do not have unbiased sample estimators, but the bias of their sample estimators can be bounded. Example queries include: `median`, `percentile`. A corollary to the previous lemma, is that if we can bound the bias for our estimator then we can achieve a bounded bias for c as well.

COROLLARY 1. *If there exists a bounded bias sample estimator for q then there exists a bounded bias sample estimator for c.*

Functionally, we can apply the same estimation procedure as the unbiased case.

For both cases above, we may not get analytic confidence intervals on our results, nor is it guaranteed that our estimates are optimal. We can use a technique called a statistical bootstrap [4] to empirically bound our correction. In this approach, we repeatedly subsample with replacement from our sample and apply the sample estimator. We use these repeated executions to build a distribution of values that the correction can take allowing us to bound the result.

PROPOSITION 2. *(BOOTSTRAP OVER DIFFERENCES) Let q be an aggregate query that has an unbiased sample estimate, and let $\hat{S}$ and $\hat{S}'$ be sample views as defined before. One sample of the bootstrap estimator s is defined as the difference of q applied to random subsample of size b_1 (with replacement) of $\hat{S}$ and $\hat{S}'$. We denote subsamples of the samples as $\hat{S}'_{sub}$ and $\hat{S}_{sub}$ respectively.*

$$q(\hat{S}'_{sub}) - q(\hat{S}_{sub})$$

To build the confidence interval, we repeatedly apply this procedure b_2 times.

There has been recent research in using techniques such as Poissonized resampling [3], analytical bootstrap [52], and bagging [28] to make this algorithm better suited for latency-sensitive query processing application.

5.2.1 MIN and MAX

`min` and `max` fall into their own category since there does not exist any unbiased sample estimator nor can that bias be bounded. We devise an estimation procedure that corrects these queries. However, we can only achieve bound that has a slightly different interpretation than the confidence intervals seen before. We can calculate the probability that a larger (or smaller) element exists in the unsampled view.

We devise the following correction estimate for `max`: (1) For all rows in both S and S', calculate the row-by-row difference, (2) let c be the max difference, and (3) add c to the max of the stale view.

We can give weak bounds on the results using Cantelli's Inequality. If X is a random variable with mean μ_x and variance $var(X)$, then the probability that X is larger than a constant ϵ

$$\mathbb{P}(X \geq \epsilon + \mu_x) \leq \frac{var(X)}{var(X) + \epsilon^2}$$

Therefore, if we set ϵ to be the difference between max value estimate and the average value, we can calculate the probability that we will see a higher value.

The same estimator can be modified for `min`, with a corresponding bound:

$$\mathbb{P}(X \leq \mu_x - a)) \leq \frac{var(x)}{var(x) + a^2}$$

This bound has a slightly different interpretation than the confidence intervals seen before. This gives the probability that a larger (or smaller) element exists in the unsampled view.

5.3 Select Queries

In SVC, we also explore how to extend this correction procedure to Select queries. Suppose, we have a Select query with a predicate:

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

We first run the Select query on the stale view, and this returns a set of rows. This result has three types of data error: rows that are missing, rows that are falsely included, and rows whose values are incorrect.

As in the `sum`, `count`, and `avg` query case, we can apply the query to the sample of the up-to-date view. From this sample, using our lineage defined earlier, we can quickly identify which rows were added, updated, and deleted. For the updated rows in the sample, we overwrite the out-of-date rows in the stale query result. For the new rows, we take a union of the sampled selection and the updated stale selection. For the missing rows, we remove them from the stale selection. To quantify the approximation error, we can rewrite the Select query as `count` to get an estimate of number of rows that were updated, added, or deleted (thus three "confidence" intervals).

6. OUTLIER INDEXING

Sampling is known to be sensitive to outliers [7,11]. Power-laws and other long-tailed distributions are common in large datasets [11]. We address this problem using a technique called outlier indexing which has been applied in AQP [7]. The basic idea is that we create an index of outlier records (records whose attributes deviate from the mean value greatly) and ensure that these records are included in the sample. The intuition is that these records greatly increase the variance of the data and since they are likely rare the probability of sampling them is low leading to wildly varying estimates when the sample contains and outlier and when it does not contain any outliers. However, as this has not been explored in the materialized view setting there are new challenges in using this index for improved result accuracy.

6.1 Indices on the Base Relations

In [7], the authors applied outlier indexing to improve the accuracy of AQP. We apply a similar technique, however, their problem setting is different in a few ways. First, in the AQP setting, queries are issued to base relations. In our problem, we issue queries to materialized views. We need to define how to propagate information from an outlier index on the base relation to a materialized view.

The first step is that the user selects an attribute of any base relation to index and specifies a threshold t and a size limit k. In a single pass of updates (without maintaining the view), 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.

There are many approaches to select a threshold. 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. Alternatively, we can calculate the variance of the dataset and set the threshold to represent c standard deviations above the mean.

This threshold can be adaptively set at each maintenance period to include more or less outliers. The caveat is that the outlier index should not be too expensive to calculate nor should it be too large as it negates the performance benefits of sampling. The query processing approach that we propose in the following sub-sections is agnostic to how we choose this threshold. In fact, our approach allows us to incorporate any deterministic subset into our sample-based correction calculations.

6.2 Adding Outliers to the Sample

We ensure that any row in a materialized view that is derived from an indexed record is guaranteed to be in the sample. This problem is sort of an inverse to the efficient sampling problem studied in Section 4. We need to propagate the indices upwards through the expression tree.

We add the condition that the only eligible indices are ones on base relations that are being sampled (i.e., we can push the hash operator down to that relation). Therefore, in the same iteration as sampling, we can also test the index threshold and add records to the outlier index. We formalize the propagation property recursively. Every relation can have an outlier index which is a set of attributes and a set of records that exceed the threshold value on those attributes.

The main idea is to treat the indexed records as a sub-relation that gets propagated upwards with the maintenance strategy.

DEFINITION 6 (OUTLIER INDEX PUSHUP). *Define an outlier index to be a tuple of a set of indexed attributes, and a set of records (I, O). The outlier index propagates upwards with the following rules:*

- *Base Relations: Outlier indices on base relations are pushed up only if that relation is being sampled, i.e., if the sampling operator can be pushed down to that relation.*
- *$\sigma_\phi(R)$: Push up with a new outlier index and apply the selection to the outliers $(I, \sigma_\phi(O))$*
- *$\Pi_{(a_1,...,a_k)}(R)$: Push upwards with new outlier index $(I \cap (a_1, ..., a_k), O)$.*
- *$\bowtie_{\phi(r1,r2)} (R_1, R_2)$: Push upwards with new outlier index $(I_1 \cup I_2, O_1 \bowtie O_2)$.*
- *$\gamma_{f,A}(R)$: For group-by aggregates, we set I to be the aggregation attribute. For the outlier index, we do the following steps. (1) Apply the aggregation to the outlier index $\gamma_{f,A}(O)$, (2) for all distinct A in O select the row in $\gamma_{f,A}(R)$ with the same A, and (3) this selection is the new set of outliers O.*
- *$R_1 \cup R_2$: Push up with a new outlier index $(I_1 \cap I_2, O_1 \cup O_2)$. The set of index attributes is combined with an intersection to avoid missed outliers.*
- *$R_1 \cap R_2$: Push up with a new outlier index $(I_1 \cap I_2, O_1 \cap O_2)$.*
- *$R_1 - R_2$: Push up with a new outlier index $(I_1 \cup I_2, O_1 - O_2)$.*

For all outlier indices that can propagate to the view (i.e., the top of the tree), we get a final set O of records. Given these rules, O is, in fact, a subset of our materialized view S'. Thus, our query processing can take advantage of the theory described in the previous section to incorporate the set O into our results. We implement the outlier index as an additional attribute on our sample with a boolean flag true or false if it is an outlier indexed record. If a row is contained both in the sample and the outlier index, the outlier index takes precedence. This ensures that we do not double count the outliers.

6.3 Query Processing

For result estimation, we can think of our sample $\hat{S}'$ and our outlier index O as two distinct parts. Since $O \subset S'$, and we give membership in our outlier index precedence, our sample is actually a sample restricted to the set $(\widehat{S' - O})$. 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.

For (2), we can also incorporate the outliers into our correction estimates. For a given query, let c_{reg} be the correction calculated on $(\widehat{S' - O})$ using the technique proposed in the previous section and adjusting the sampling ratio m to account for outliers removed from the sample. We can also apply the technique to the outlier set O since this set is deterministic the sampling ratio for this set is $m = 1$, and we call this result c_{out}. Let N be the count of records that satisfy the query's condition and l be the number of outliers that satisfy the condition. Then, we can merge these two corrections in the following way: $v = \frac{N-l}{N} c_{reg} + \frac{l}{N} c_{out}$.

For the queries in the previous section that are unbiased, this approach preserves unbiasedness. Since we are averaging two unbiased estimates c_{reg} and c_{out}, the linearity of the expectation operator preserves this property. Furthermore, since c_{out} is deterministic (and in fact its bias/variance is 0), c_{reg} and c_{out} are uncorrelated making the bounds described in the previous section applicable as well.

7. EXTENSIONS

7.1 Hash-Operator

We defined a concept of tuple-lineage with primary keys. However, a curious property of the deterministic hashing technique is that we can actually hash any attribute while retain the important statistical properties. This is because a uniformly random sample of any attribute (possibly not unique) still includes every individual row with the same probability. A consequence of this is that we can push down the hashing operator through arbitrary equality joins (not just many-to-one) by hashing the join key.

We defer further exploration of this property to future work as it introduces new tradeoffs. For example, sampling on a non-unique key, while unbiased in expectation, has higher variance in the size of the sample. Happening to hash a large group may lead to decreased performance.

Suppose our keys are duplicated μ_k times on average with variance σ_k^2, then the variance of the sample size is for sampling fraction m:

$$m(1-m)\mu_k^2 + (1-m)\sigma_k^2$$

This equation is derived from the formula for the variance of a mixture distribution. In this setting, our sampling would have to consider this variance against the benefits of pushing the hash operator further down the query tree.

7.2 Sampling Updates vs. Sampling Views

In SVC, we sample from views and work backwards through the view definition using relational algebra. An alternative approach is to sample the base relations of the view. However, this approach quickly leads to some bottlenecks. For example, if our view is an aggregate view with a nested selection, we can easily construct a distinct count problem rendering any aggregate query inestimable [6].

For some types of views, this model is actually a special case of SVC. For views where primary key of the base relations is an attribute of the view, we can sample those attributes. We can quickly see that based on our pushdown rules if there is a nested aggregate query, pushdown can fail in general. However, re-writing views and queries to better support sampling is an interesting avenue of future work.

7.3 Multi-View Setting

In a production environment, the database system might have many materialized views. With the sampling ratio, SVC gives the database administrator an additional degree of freedom to adjust throughput, storage, and accuracy. The sampling ratio of each view can be adaptively adjusted to suit the workload.

We can pose minimizing the expected estimation error as a Geometric Convex Program. In one time period, if view i has an expected cardinality of N_i, an average query variance of α_i, a sampling ratio of m_i, there is a total space budget of B, the cost for update is C_i secs/Record and throughput demand of D latency:

$$\arg\min_{m_i} \sum_i \frac{\alpha_i}{m_i \cdot N_i}$$

$$\text{subject to: } \sum_i m_i \cdot N_i \leq B$$

$$\sum_i m_i \cdot N_i \cdot C_i \leq D$$

8. RESULTS

We evaluate SVC first on a single node MySQL database to evaluate its accuracy, performance, and efficiency in a variety of materialized view scenarios. We look at three main applications, join view maintenance, aggregate view maintenance, and an application with complex materialized views, on a skewed version of the benchmark TPCD-Skew. 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 server log analysis with a dataset from a video streaming company, Conviva. In this application, we look at the real query workload of the company and materialize views that could improve performance of these queries. These queries correspond to summary statistics of customer data like how many visits to all of a customers videos or the number of vistors who encountered errors.

8.1 Experimental Setup

8.1.1 Single-node Experimental Setup

Our single node experiments are run on a 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. These experiments evaluate views from 10GB TPCD and TPCD-Skew datasets. TPCD-Skew dataset [9] 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 [36] is a long-tailed distribution with a single parameter $z = \{1, 2, 3, 4\}$ which a larger value means a more extreme tail. $z = 1$ corresponds to the basic TPCD benchmark. We implement incremental view maintenance with an "update...on duplicate key insert" command. We implement SVC's sampling operator with a linear hash written in C that is evoked in MySQL as a stored procedure. In all of the applications, the updates are kept in memory in a temporary table, and we discount this loading time from our experiments. We build an index on the primary keys of the view, the base data, but not on the updates. Below we describe the view definitions and the queries on the views:

Join View: In the TPCD specification, two tables receive insertions and updates: lineitem and orders. Out of 22 parameterized queries in the specification, 12 are group-by aggregates of the join of lineitem and orders (Q3, Q4, Q5, Q7, Q8, Q9, Q10, Q12, Q14, Q18, Q19, Q21). Therefore, we define a materialized view of the foreign-key join of lineitem and orders, and compare incremental view maintenance and SVC. We treat the 12 group-by aggregates as queries on the view.

Aggregate View: We apply SVC in an application similar to data cubes [23]. We define the following "base cube" as a materialized view that calculates the total revenue grouped by distinct customer, nation, region, and part number. The queries on this view are "roll-up" queries that aggregate over subsets of the groups (e.g., total of all customers in North America).

Complex Views: Our goal is to demonstrate the applicability of SVC outside of simple materialized views that include nested queries and other more complex relational algebra. We take the TPCD schema and denormalize the database, and treat each of the 22 TPCD queries as views on this denormalized schema. The 22

TPCD queries are actually parameterized queries where parameters, such as the selectivity of the predicate, are randomly set by the TPCD qgen program. Therefore, we use the program to generate 10 random instances of each query and use those as our materialized view. We remove views with a small result set making them not suitable for sampling or are static. 10 out of the 22 sets of views can benefit from SVC.

For each of the views, we generated *queries on the views*. Since the outer queries of our views were group by aggregates, we picked a random attribute a from the group by clause and a random attribute b from aggregation. We use a to generate a predicate. For each attribute a, the domain is specified in the TPCD standard. We select a random subset of this domain ,e.g., if the attribute is country then the predicate can be countryCode ¿ 50 and countryCode ¡ 100. We generated 100 random `sum`, `avg`, and `count` queries for each view.

8.1.2 Distributed Experimental Setup

We evaluated performance on Apache Spark 1.1.0 with a 10 node r3.large Amazon EC2 cluster. Systems like Spark are increasingly popular, and Spark supports materialization through a distributed data structure called an RDD [50]. In the most recent release of Spark, there is a SQL interface that allows users to persist query results in memory.

However, in real-world application, RDDs have limitations. As RDDs are an immutable data structure, any maintenance must be done synchronously. As Spark also does not have support for indices, we rely on partitioned joins for incremental maintenance of the views. We partitioned the views by primary-by key, and apply a full outer join of the updates with the partitioned view. With this set of experiments, we show that SVC implemented on Spark can help overcome some of these limitations and allow users to take advantage of materialized views and incremental maintenance.

We evaluate SVC on a 1TB dataset of logs from a video streaming company, Conviva [1]. The dataset is a denormalized user activity log corresponding to video views and various metrics such as data transfer rates, and latencies. 1TB corresponded to a sample of customer data from 1 year of logs. With this dataset, there was a corresponding dataset of analyst SQL queries on the log table. Using the dataset of analyst queries, we identified 8 common summary statistics-type queries that calculated engagement and error-diagnosis metrics for specific customers on a certain day. We generalized these queries by turning them into group-by queries over customers and dates; that is a view that calculates the metric for every customer on every day. We generated aggregate random queries over this dataset by taking either random time ranges or random subsets of customers. We list high-level characteristics of the queries: While, we cannot give the details of the queries, we can present some of the high-level characteristics of 8 summary-statistics type views.

- **V1.** Counts of various error types grouped by resources, users, date
- **V2.** Sum of bytes transferred grouped by resource, users, date
- **V3.** Counts of visits grouped by an expression of resource tags, users, date.
- **V4.** Nested query that groups users from similar regions/service providers together then aggregates statistics
- **V5.** Nested query that groups users from similar regions/service providers together then aggregates error types
- **V6.** Union query that is filtered on a subset of resources and aggregates visits and bytes transferred
- **V7.** Aggregate network statistics group by resources, users, date with many aggregates.
- **V8.** Aggregate visit statistics group by resources, users, date with many aggregates.

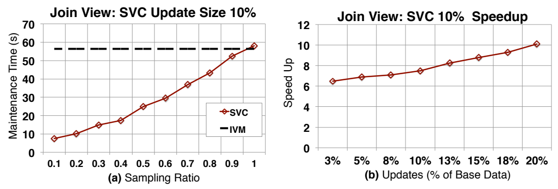


Figure 4: (a) On a 10GB dataset with 1GB of insertions and updates, we vary the sampling ratio and measure the maintenance time of SVC. The black line marks the time for full incremental view maintenance. (b) For a fixed sampling ratio of 10%, we vary the update size and plot the speedup compared to full incremental maintenance for SVC.

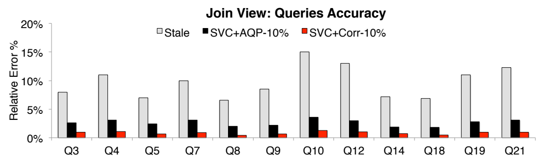


Figure 5: We generate 100 of each TPCD parameterized query and answer it using the stale materialized view, SVC+Corr, and SVC+AQP. We plot the median relative error for each query (since the result for each query might be multi-valued).

8.1.3 Metrics and Evaluation

To illustrate how SVC gives the user access to this new tradeoff space, we will illustrate that SVC is more accurate than the stale query result (No Maintenance); but is less computationally intensive than full IVM. In our evaluation, we separate maintenance from query processing. We use the following notation to represent the different approaches:

- No maintenance (Stale): The baseline for evaluation is not applying any maintenance to the materialized view.
- Incremental View Maintenance (IVM): We apply incremental view maintenance to the full view.
- SVC+AQP: We maintain a sample of the materialized view using SVC but estimate the result with AQP rather than using the correction technique proposed in this paper.
- SVC+Corr: We maintain a sample of the materialized view using SVC and process queries on the view using the correction method presented in this paper.

Since SVC has a sampling parameter, we denote a sample size of x% as SVC+Corr-x or SVC+AQP-x, respectively. To evaluate accuracy and performance, we define the following metrics:

- Relative Error: For a query result r and an incorrect result r', the relative error is $\frac{|r-r'|}{r}$. When a query has multiple results (a group-by query), then, unless otherwise noted, relative error is defined as the median over all the errors.
- Maintenance Time: We define the maintenance time as the time needed to produce the up-to-date view for incremental view maintenance, and the time needed to produce the up-to-date sample in SVC.

8.2 Single-node Accuracy and Performance

8.2.1 Join View

In our first experiment, we evaluate how SVC performs on a materialized view of the join of lineitem and orders. We generate a 10GB base TPCD dataset with skew $z = 1$, and derive the view from this dataset. We first generate 1GB (10% of the base data) of updates (insertions and updates to existing records), and vary the sample size.

Performance: Figure 4(a), shows the maintenance time of SVC as a function of sample size. With the bolded dashed line, we note the time for full IVM. For this materialized view, sampling allows

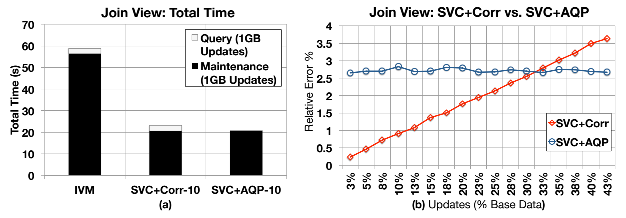


Figure 6: (a) For a fixed sampling ratio of 10% and update size of 10% (1GB), we measure the total time incremental maintenance + query time. SVC+Corr does take longer to query a view (due to the correction) than SVC+AQP and IVM, but this overhead is small relative to to the savings in maintenance time. (b) We vary the update rate to show that SVC+Corr is more accurate than SVC+AQP until the update size is 32.5% (3.2GB).

for significant savings in maintenance time; albeit for approximate answers. While full incremental maintenance takes 56 seconds, SVC with a 10% sample can complete in 7.5 seconds.

The speedup for SVC-10 is 7.5x which is far from ideal on a 10% sample. In the next figure, Figure 4(b), we evaluate this speedup. We fix the sample size to 10% and plot the speedup of SVC compared to IVM while varying the size of the updates. On the x-axis is the update size as a percentage of the base data. For small update sizes, the speedup is smaller, 6.5x for a 2.5% (250GB) update size. As the update size gets larger, SVC becomes more efficient, since for a 20% update size (2GB), the speedup is 10.1x. The super-linearity is because this view is a join of lineitem and orders and we assume that there is not a join index on the updates. Since both tables are growing sampling reduces computation super-linearly.

Accuracy: At the same design point with a 10% sample, we evaluate the accuracy of SVC. In Figure 5, we answer TPCD queries with this view. The TPCD queries are group-by aggregates and we plot the median relative error for SVC+Corr, No Maintenance, and SVC+AQP. On average over all the queries, we found that SVC+Corr was 11.7x more accurate than the stale baseline, and 3.1x more accurate than applying SVC+AQP to the sample.

SVC+Corr vs. SVC+AQP: While more accurate, it is true that SVC+Corr correction technique moves some of the computation from maintenance to query execution. SVC+Corr calculates a correction to a query on the full materialized view. On top of the query time on the full view (as in IVM) there is additional time to calculate a correction from a sample. On the other hand SVC+AQP runs a query only on the sample of the view, We evaluate this overhead in Figure 6(a), where we compare the total maintenance time and query execution time. For a 10% sample SVC+Corr required 2.69 secs to execute a `sum` over the whole view, IVM required 2.45 secs, and SVC+AQP required 0.25 secs. However, when we compare this overhead to the savings in maintenance time it is small.

SVC+Corr is most accurate when the materialized view is less stale, since it relies on correct a stale result. On the other hand SVC+AQP is more robust to the staleness and gives a consistent relative error. The error for SVC+Corr grows proportional to the staleness. In Figure 6(b), we explore which query processing technique should be used SVC+Corr or SVC+AQP. For a 10% sample, we find that SVC+Corr is more accurate until the update size is 32.5% of the base data.

8.2.2 Aggregate View

In our next experiment, we evaluate an aggregate view use case similar to a data cube. We generate a 10GB base TPCD dataset with skew $z = 1$, and derive the base cube as a materialized view from this dataset. We add 1GB of updates and apply SVC to estimate the results of all of the "roll-up" dimensions.

Performance: We observed the same trade-off as the previous experiment where sampling significantly reduces the maintenance time (Figure 7(a)). It takes 186 seconds to maintain the entire view, but a 10% sample can be cleaned with SVC in 26 seconds. As

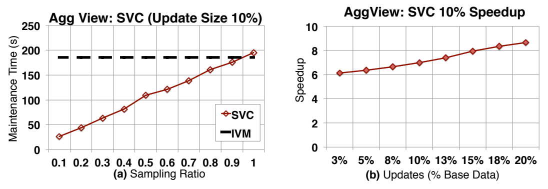


Figure 7: (a) On a 10GB dataset with 1GB of insertions and updates, we vary the sampling ratio and measure the maintenance time of SVC. The black line marks the time for full incremental view maintenance. (b) For a fixed sampling ratio of 10%, we vary the update size and plot the speedup compared to full incremental maintenance for SVC.

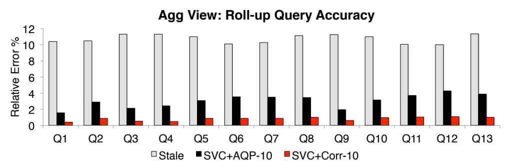


Figure 8: We measure the accuracy of each of the roll-up aggregate queries on this view. For a 10% sample size and 10% update size, we find that SVC+Corr is more accurate than SVC+AQP and No Maintenance.

before, we fix the sample size at 10% and vary the update size. We similarly observe that SVC becomes more efficient as the update size grows (Figure 7(b)), and at an update size of 20% the speedup is 8.7x.

Accuracy: In Figure 8, we measure the accuracy of each of the "roll-up" aggregate queries on this view. That is, we take each dimension and aggregate over the dimension. We fix the sample size at 10% and the update size at 10%. On average SVC+Corr is 12.9x more accurate than the stale baseline and 3.6x more accurate than SVC+AQP.

Other Queries: Finally, we also use the data cube to illustrate how SVC can support a broader range of queries outside of `sum`, `count`, and `avg`. We change all of the roll-up queries to use the **median** function (Figure 9). First, both SVC+Corr and SVC+AQP are more accurate as estimating the median than they were for estimating sums. This is because the median is less sensitive to variance in the data.

8.2.3 Complex Views

In this experiment, we demonstrate the breadth of views supported by SVC by using the TPCD queries as materialized views.

Performance: Figure 10, shows the maintenance time for a 10% sample compared to the full view. This experiment illustrates how the view definitions plays a role in the efficiency of our approach. For the last two views, V21 and V22, we see that sampling does not lead to as large of speedup indicated in our previous experiments. This is because both of those views contain nested structures which block the pushdown of hashing. V21 contains a subquery in its predicate that does not involve the primary key, but still requires a scan of the base relation to evaluate. V22 contains a string transformation of a key blocking the push down. There might be a way to derive an equivalent expression with joins that could be sampled more efficiently, and we will explore this in future work. For the most part, these results are consistent with our previous experiments showing that SVC is faster than IVM and more accurate than SVC+AQP and no maintenance.

8.2.4 Outlier Indexing

In our next experiment, we evaluate our outlier indexing. We index the l_extendedprice attribute in the lineitem table. We evaluate the outlier index on the complex TPCD views. We find that four views: V3, V5, V10, V15, can benefit from this index with our

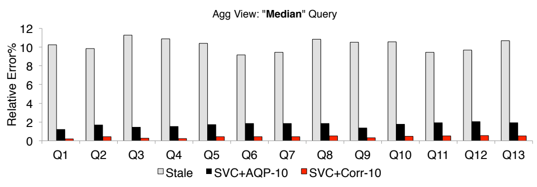


Figure 9: We run the same experiment but replace the `sum` query with a median query. We find that similarly SVC is more accurate.

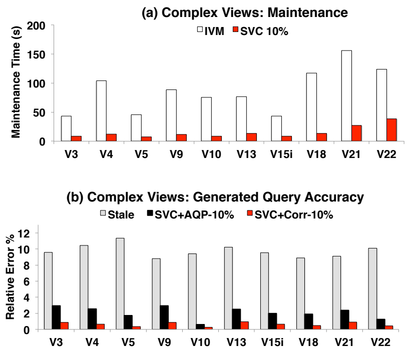


Figure 10: (a) For 1GB update size, we compare maintenance time and accuracy of SVC with a 10% sample on different views. V21 and V22 do not benefit as much from SVC due to nested query structures. (b) As before for a 10% sample size and 10% update size, SVC+Corr is more accurate than SVC+AQP and No Maintenance.

push-up rules. These are four views dependent on l_extendedprice that were also in the set of "Complex" views chosen before.

In our first outlier indexing experiment (Figure 11(a)), we analyze V3. We set an index of 100 records, and applied SVC+Corr and SVC+AQP to datasets with a skew parameter $z = \{1, 2, 3, 4\}$. We run the same queries as before, but this time we measure the error at the 75% quartile. We find in the most skewed dataset SVC with outlier indexing reduces query error by a factor of 2. Next, in $\big($Figure 11 (b)$\big)$, we plot the overhead for outlier indexing for V3 with an index size of 0, 10, 100, and 1000. While there is an overhead, it is still small compared to the gains made by sampling the maintenance strategy.

8.3 Conviva

8.3.1 Performance and Accuracy

We derive the views from 800GB of base data and add 80GB of updates. In Figure 12(a), we show that while full maintenance takes nearly 800 seconds for one of the views, a SVC-10% can complete sample cleaning in less than 100s for all of them. On average over all the views, SVC-10% gives a 7.5x speedup.

In Figure 12(b), we show that SVC also gives highly accurate results with an average error of 0.98%. In the following experiments, we will use V2 and V5 as exemplary views. V5 is the most expensive to maintain due to its nested query and V2 is a single level group by aggregate. These results show consistency with our results on the synthetic datasets.

8.3.2 End-to-end integration with periodic maintenance

We devised an end-to-end experiment simulating a real integration with periodic maintenance. However, unlike the MySQL case, Apache Spark does not support selective updates and insertions as the "views" are immutable. A further point is that the immutability

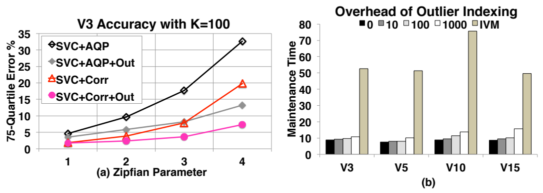


Figure 11: (a) For one view V3 and 1GB of updates, we plot the 75% quartile error with different techniques as we vary the skewness of the data. We find that SVC with an outlier index of size 100 is the most accurate. (b) While the outlier index adds [illegible]ve to the total maintenance time.

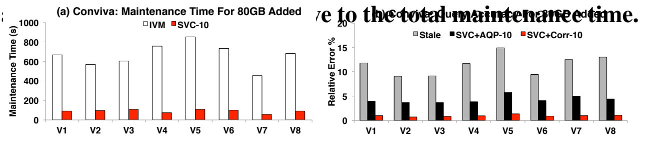


Figure 12: (a) We compare the maintenance time of SVC with a 10% sample and full incremental maintenance, and find that as with TPCD SVC saves significant maintenance time. (b) We also compare SVC+Corr to SVC+AQP and No Maintenance and find it is more accurate.

of these views and Spark's fault-tolerance requires that the "views" are maintained synchronously. Thus, to avoid these significant overheads, we have to update these views in batches. Spark does have a streaming variant [51], however, this does not support the complex SQL derived materialized views used in this paper, and still relies on mini-batch updates.

SVC and IVM will run in separate threads each with their own RDD materialized view. In this application, both SVC and IVM maintain respective their RDDs with batch updates. In this model, there are a lot of different parameters: batch size for periodic maintenance, batch size for SVC, sampling ratio for SVC, and the fact that concurrent threads may reduce overall throughput. Our goal is to fix the throughput of the cluster, and then measure whether SVC+IVM or IVM alone leads to more accurate query answers.

Batch Sizes: In Spark, larger batch sizes amortize overheads better. In Figure 13(a), we show a trade-off between batch size and throughput of Spark for V2 and V5. Throughputs for small batches are nearly 10x smaller than the throughputs for the larger batches.

Concurrent SVC and IVM: Next, we measure the reduction in throughput when running multiple threads. We run SVC-10 in loop in one thread and IVM in another. We measure the reduction in throughput for the cluster from the previous batch size experiment. In Figure 13(b), we plot the throughput against batch size when two threads (SVC and IVM) are running. While for small batch sizes the throughput of the cluster is reduced by nearly a factor of 2, for larger sizes the reduction is smaller. As we found in later experiments (Figure 15), larger batch sizes are more amenable to parallel computation since there was more idle CPU time.

Choosing a Batch Size: The results in Figure 13(a) and Figure 13(b) show that larger batch sizes are more efficient, however, larger batch sizes also lead to more staleness. Combining the results in Figure 13(a) and Figure 13(b), for both SVC+IVM and IVM, we get cluster throughput as a function of batch size. For a fixed throughput, we want to find the smallest batch size that achieves that throughput for both. For V2, we fixed this at 700,000 records/sec and for V5 this was 500,000 records/sec. For IVM alone the smallest batch size that met this throughput demand was 40GB for both V2 and V5. And for SVC+IVM, the smallest batch size was 80GB for V2 and 100GB for V5. When running periodic maintenance alone view updates can be more frequent, and when run in conjunction with SVC it is less frequent.

We run both of these approaches in a continuous loop, SVC+IVM and IVM, and measure their maximal error during a maintenance

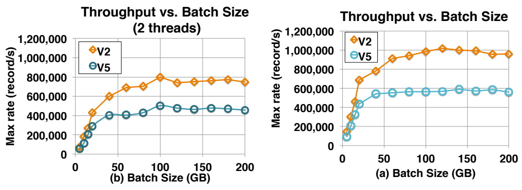


Figure 13: (a) Spark RDDs are most efficient when updated in batches. As batch sizes increase the system throughput in-crease ... n ... ighput reduce

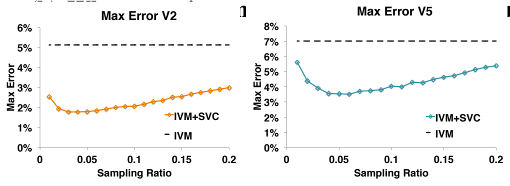


Figure 14: For a fixed throughput, SVC+Periodic Maintenance gives more accurate results for V2 and V5.

period. There is further a trade-off with the sampling ratio, larger samples give more accurate estimates however between SVC batches they go stale. We quantify the error in these approaches with the max error; that is the maximum error in a maintenance period (Figure 14). These competing objective lead to an optimal sampling ratio of 3% for V2 and 6% for V5. At this sampling point, we find that applying SVC gives results 2.8x more accurate for V2 and 2x more accurate for V5.

To give some intuition on why SVC gives more accurate results, in Figure 15, we plot the average CPU utilization of the cluster for both periodic IVM and SVC+periodic IVM. We find that SVC takes advantage of the idle times in the system; which are common during shuffle operations in a synchronous parallelism model. In a way, these experiments present a worst-case application for SVC, yet it still gives improvements in terms of query accuracy. In many deployments, throughput demands are variable (e.g., peak traffic) which means incremental maintenance is not at all feasible leaving systems with longer idle times.

9. RELATED WORK

Addressing the cost of materialized view maintenance is the subject of many recent papers, which focus on various perspectives including complex analytical queries [37], transactions [5], and physical design [34]. The increased research focus parallels a major concern in industrial systems for incrementally updating precomputed results and indices such as Google Percolator [42] and Twitter's Rainbird [49]. The streaming community has also studied the view maintenance problem [2,19,21,22,26,31]. In Spark Streaming, Zaharia et al. studied how they could exploit in-memory materialization [51], and in MonetDB, Liarou et al. studied how ideas from columnar storage can be applied to enable real-time analytics [35]. These works focus on correctness, consistency, and fault tolerance of materialized view maintenance. SVC proposes an alternative model where we allow approximation error (with guarantees) for queries on materialized views for vastly reduced maintenance time. In many decision problems, exact results are not needed as long as the probability of error is boundable.

Sampling has been well studied in the context of query processing [4,18,39]. Both the problems of efficiently sampling relations [39] and processing complex queries [3], have been well studied. In SVC, we look at a new problem, where we efficiently sample from a maintenance strategy, a relational expression that updates a materialized view. We generalize uniform sampling procedures to work in this new context using lineage [15] and hashing. We look the problem of approximate query processing [3,4] from a different perspective by estimating a "correction" rather than estimating query results. Srinivasan and Carey studied a problem related to query correction which they called compensation-based query processing [46] for concurrency control. However, this work did not consider applications when the correction was applied to a sample as in SVC.

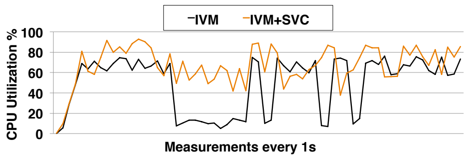


Figure 15: SVC better utilizes idle times in the cluster by maintaining the sample.

In the context of materialized view maintenance, sampling has primarily been studied from the perspective of maintaining samples [41]. Similarly, in [27], Joshi and Jermaine studied indexed materialized views that are amenable to random sampling. While similar in spirit (queries on the view are approximate), the goal of this work was to optimize query processing not address the cost of incremental maintenance. There has been work using sampled views in a limited context of cardinality estimation [32], which is the special case of our framework, namely, the `count` query. Nirkhiwale et al. [38], studied an algebra for sampling in aggregate queries. They studied how to build query plans for aggregate queries (with nested subqueries and joins) that involved a Generalized Uniform Sampling (GUS) primitive. This work did use lineage and similar to our work defined the GUS primitive as a random selection over the primary key. This is similar to our model where we have a materialized view and aggregate queries on the materialized view. However, they did not discuss the use of hashing to efficiently implement this primitive, and restricted their analysis to the `sum` query. The problem setting was also very different where the objective is efficient planning of aggregate queries with sampling operators and not efficient updates of materialized views.

Sampling has been explored in the streaming community, and a similar idea of sampling from incoming updates has also been applied in stream processing [17,43,47]. While some of these works studied problems similar to materialization, for example, the JetStream project (Rabkin et al.) looks at how sampling can help with real-time analysis of aggregates. None of these works formally studied the class views that can benefit from sampling or formalized queries on these views. There are a variety of other efforts proposing storage efficient processing of aggregate queries on streams [16,24] which is similar to our problem setting and motivation.

Finally, the theory community has has studied related problems. There has been work on the maintenance of approximate histograms, synopses, and sketches [13,20], which closely resemble aggregate materialized views. This work did not model queries on the approximate data structures as in SVC. Furthermore, the goals of this line work (including techniques such as sketching and approximate counting) has been to reduce the required storage, not to reduce the required update time.

10. LIMITATIONS AND OPPORTUNITIES

SVC proposes a new approach for accurate query processing with MVs. Our results are promising and suggest many avenues for future work. In particular, we are interested in deeper exploration of the multiple MV setting. There are many interesting design problems such as given storage constraints and throughput demands, optimize sampling ratios over all views. Furthermore, there is an interesting challenge about queries that join mutliple sample MVs managed by SVC. We are also interested in the possibility of sharing computation between MVs and maintenance on views derived from other views.

While our experiments show that SVC works for a variety applications, as with all sampling and approximation techniques, there are a few limitations which we summarize in this section. SVC does not support views with ordering or "top-k" clauses, as our sampling assumes no ordering on the rows of the MV. Furthermore, while there are a few key special cases (equality joins) arbitrary joins do not commute with sampling, making SVC ineffecient in those cases. SVC also requires the maintenance strategy to be parameterized in terms of relational algebra which may not always be possible if that is a black blox. In terms of queries on the view, as with previous work in AQP, sampling is best suited for aggregate queries. While we proposed a technique that can give answers for some select queries, in general, SVC will give poor results for very selective queries. Other limitations of sampling have been discussed extensively in [13] including challenges with distinct and nested queries.

11. CONCLUSION

Materialized view maintenance is often expensive, and in practice, immediate view maintenance is avoided due to its costs. However, this leads to stale materialized views which have incorrect, missing, and superfluous rows. In this work, we formalize the problem of staleness and view maintenance as a data cleaning problem.

SVC uses a sample-based data cleaning approach to get accurate query results that reflect the most recent data for a greatly reduced computational cost. To achieve this, we significantly extended our prior work in data cleaning, SampleClean [48], for efficient cleaning of stale MVs. This included processing a wider set of aggregate queries, handling missing data errors, and proving for which queries optimality of the estimates hold. Another significant contribution of SVC is our outlier indexing approach which reduces the sensitivity of sampling to skewed data distributions. We presented both empirical and theoretical results showing that our sample data cleaning approach is significantly less expensive than full view maintenance for a large class of materialized views, while still providing accurate aggregate query answers that reflect the most recent data. We evaluate SVC on a real dataset of server logs from Conviva and the TPCD benchmark dataset, and our experiments confirm our theoretical results: (1) cleaning an MV sample is more efficient than full view maintenance, (2) the corrected results are more accurate than using the stale MV, and (3) SVC can be efficiently integrated with deferred maintenance leading to increased query accuracy.

12. REFERENCES

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

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

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

[5] 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] M. Charikar, S. Chaudhuri, R. Motwani, and V. R. Narasayya. Towards estimation error guarantees for distinct values. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 15-17, 2000, Dallas, Texas, USA*, pages 268–279, 2000.

[7] 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.

[8] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *SIGMOD*, volume 28, pages 263–274. ACM, 1999.

[9] S. Chaudhuri and V. Narasayya. TPC-D data generation with skew. `ftp.research.microsoft.com/users/viveknar/tpcdskew`.

[10] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[11] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.

[12] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *SIGMOD Conference*, pages 469–480, 1996.

[13] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012.

[14] D. R. Cox and D. V. Hinkley. *Theoretical statistics*. CRC Press, 1979.

[15] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *VLDB J.*, 12(1):41–58, 2003.

[16] A. Dobra, M. N. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *SIGMOD Conference*, pages 61–72, 2002.

[17] M. Garofalakis, J. Gehrke, and R. Rastogi. *Data Stream Management: Processing High-Speed Data Streams*. Springer, 2011.

[18] M. N. Garofalakis and P. B. Gibbons. Approximate query processing: Taming the terabytes. In *VLDB*, 2001.

[19] 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.

[20] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. *ACM Trans. Database Syst.*, 27(3):261–298, 2002.

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

[22] 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.

[23] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[24] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD Conference*, pages 58–66, 2001.

[25] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[26] 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.

[27] S. Joshi and C. M. Jermaine. Materialized sample views for database approximation. *IEEE Trans. Knowl. Data Eng.*, 20(3):337–351, 2008.

[28] A. Kleiner, A. Talwalkar, S. Agarwal, I. Stoica, and M. I.

Jordan. A general bootstrap performance diagnostic. In *KDD*, pages 419–427, 2013.

[29] C. Koch. Incremental query evaluation in a ring of databases. In *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*, pages 87–98, 2010.

[30] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2):253–278, 2014.

[31] S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre. Continuous analytics over discontinuous streams. In *SIGMOD Conference*, pages 1081–1092, 2010.

[32] P.-A. Larson, W. Lehner, J. Zhou, and P. Zabback. Cardinality estimation using sample views with quality assurance. In *SIGMOD*, pages 175–186, 2007.

[33] P.-Å. Larson and H. Z. Yang. Computing queries from derived relations. In *VLDB*, pages 259–269, 1985.

[34] 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.

[35] 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.

[36] M. Mitzenmacher. A brief history of generative models for power law and lognormal distributions. *Internet Mathematics*, 1(2):226–251, 2003.

[37] M. Nikolic, M. Elseidy, and C. Koch. Linview: incremental view maintenance for complex analytical queries. In *SIGMOD Conference*, pages 253–264, 2014.

[38] S. Nirkhiwale, A. Dobra, and C. M. Jermaine. A sampling algebra for aggregate estimation. *PVLDB*, 6(14):1798–1809, 2013.

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

[40] F. Olken and D. Rotem. Simple random sampling from relational databases. In *VLDB*, pages 160–169, 1986.

[41] F. Olken and D. Rotem. Maintenance of materialized views of sampling queries. In *ICDE*, pages 632–641, 1992.

[42] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, pages 251–264, 2010.

[43] 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.

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

[45] J. J. Shuster. Nonparametric optimality of the sample mean and sample variance. *The American Statistician*, 36(3a):176–178, 1982.

[46] V. Srinivasan and M. J. Carey. Compensation-based on-line query processing. In *SIGMOD Conference*, pages 331–340, 1992.

[47] 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.

[48] 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.

[49] K. Weil. Rainbird: Real-time analytics at twitter. In *Strata*, 2011.

[50] 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.

[51] 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.

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

[53] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. In *SIGMOD Conference*, pages 265–276, 2014.

[54] J. Zhou, P.-Å. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *VLDB*, pages 231–242, 2007.

13. APPENDIX

13.1 Join View TPCD Queries

In our first experiment, we materialize the join of lineitem and orders. We treat the TPCD queries as queries on the view, and we selected 12 out of the 22 to include in our experiments. The other 10 queries did not make use of the join.

13.2 Data Cube Specification

We defined the base cube as a materialized view:

```
select
  sum(l_extendedprice * (1 - l_discount)) as revenue,
  c_custkey, n_nationkey,
  r_regionkey, L_PARTKEY
from
  lineitem, orders,
  customer, nation,
  region
where
  l_orderkey = o_orderkey and
  O_CUSTKEY = c_custkey and
  c_nationkey = n_nationkey and
  N_REGIONKEY = r_regionkey

group by
  c_custkey, n_nationkey,
  r_regionkey, L_PARTKEY
```

Each of queries was an aggregate over subsets of the dimensions of the cube, with a `sum` over the revenue column.

- Q1. all
- Q2. c_custkey
- Q3. n_nationkey
- Q4. r_regionkey
- Q5. l_partkey
- Q6. c_custkey,n_nationkey
- Q7. c_custkey,r_regionkey
- Q8. c_custkey,l_partkey
- Q9. n_nationkey, r_regionkey
- Q10. n_nationkey, l_partkey
- Q11. c_custkey,n_nationkey, r_regionkey
- Q12. c_custkey,n_nationkey,l_partkey
- Q13. n_nationkey,r_regionkey,l_partkey

When we experimented with the median query, we changed the `sum` to a median of the revenues.

13.3 Table Of TPCD Queries 2

We denormalize the TPCD schema and treat each of the 22 queries as views on the denormalized schema. In our experiments, we evaluate 10 of these with SVC. Here, we provide a table of the queries and reasons why a query was not suitable for our experiments. The main reason a query was not used was because the cardinality of the result was small. Since we sample from the view, if the result was small eg. ¡ 10, it would not make sense to apply SVC. Furthermore, in the TPCD specification the only tables that are affected by updates are lineitem and orders; and queries that do not depend on these tables do not change; thus there is no need for maintenance.

Listed below are excluded queries and reasons for their exclusion.

- Query 1. Result cardinality too small
- Query 2. The query was static
- Query 6. Result cardinality too small
- Query 7. Result cardinality too small
- Query 8. Result cardinality too small
- Query 11. The query was static
- Query 12. Result cardinality too small
- Query 14. Result cardinality too small
- Query 15. The query contains an inner query, which we treat as a view.
- Query 16. The query was static
- Query 17. Result cardinality too small
- Query 19. Result cardinality too small
- Query 20. Result cardinality too small