Sample-View-Clean: A Data Cleaning Approach for Fresh Query Answers From Stale Materialized Views

ABSTRACT

Materialized views (MVs), stored pre-computed results, are widely used to facilitate fast queries on large datasets. However, when base data is changed, MVs become stale requiring potentially expensive updates (maintenance). In this framework, which we call Sample-View-Clean (SVC), we address the expense of materialized view maintenance from a data cleaning perspective treating staleness as a type of data error. We take inspiration from recent results in data cleaning that greatly improve query accuracy by cleaning only a small sample of dirty data. The main idea is to maintain just a sample of a materialized view, and use the sample to correct incorrect query results on the stale view. Our framework supports a wide variety of materialized views and query processing for many different types of aggregate queries; 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 periodic maintenance approaches by giving accurate and bounded query answers between periods. We evaluate our method on real and synthetic datasets and results suggest that we are able to provide accurate query results without having to maintain the full view.

1. INTRODUCTION

Materialized views (MVs), stored pre-computed results, are a well-studied approach to speed up queries on large datasets [7,19, 20,24]. During the last 30 years, the research community has thoroughly studied MVs, and all major database vendors have added support for them. In a world of ever-increasing data sizes, MVs are becoming even more important, both for traditional query processing and for more advanced analytics based on linear algebra and machine learning [26,40].

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 every 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 [7].

For frequently changing tables even incremental maintenance can be expensive since every update to the base data requires updating all the dependent views. This problem is exacerbated in Big Data environments, where new records arrive at an increasingly fast rate and where data are often distributed across multiple machines. As a result, in production environments it is common to defer view maintenance to a later time [7,9,41] so that updates can be batched together to amortize overheads and maintenance can be scheduled at times of low system utilization (eg. nightly).

While deferring maintenance has compelling benefits, it unfortunately leads to the problem that the views are stale in between maintenance periods. As a result, as the time since the last maintenance increases, queries using those views return increasingly incorrect answers. The problem of stale MVs parallels the problem of dirty data studied in data cleaning [32]; as both staleness and erroneous "dirty" records are a type of data error. This observation leads us to the main insight behind our work; namely, that a data cleaning approach can be applied to mitigate the negative impacts of deferred MV maintenance.

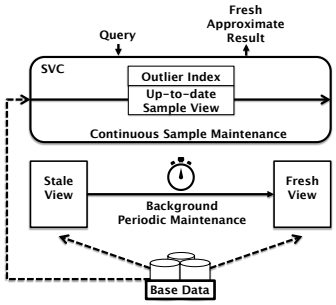

Figure 1: If a view is being periodically maintained, in the interim, query results may be stale. Sample-View-Clean sits above an existing view maintenance architecture, and provides an interface for approximately up-to-date query results by maintaining a sample. The user can tune the sampling ratio to meet the resource constraints of the system.

Data cleaning has been studied extensively in the literature (e.g., see Rahm and Do for a survey [32]) but increasing data volumes and arrival rates 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 [36]. We applied data cleaning to a sample of data and used this clean sample to improve the results of aggregate queries. This perspective raises a new possibility for MVs: we can use a sample of clean (up-to-date) rows in the view to return more accurate query results without incurring the cost of full view maintenance. Of course, the metaphor of stale MVs as dirty data only goes so far. View staleness is a different type of error than typical dirty data, which raises interesting new challenges in materializing a sample, updating the sample, and efficient query processing.

To address these new challenges, we propose Sample-View-Clean (SVC), a framework that applies sampling to approximately correct query results on stale views. SVC takes a sample of up-to-date rows from the view, analyzes how those rows have changed, and extrapolates a correction factor for query answers on the stale view. Figure 1 shows how SVC can be used as complementary to existing deferred maintenance approaches. When the MVs become stale between maintenance cycles, we apply SVC for query result estimation for a far smaller cost than having to maintain the entire

view. The query results from SVC are up-to-date in the sense that they reflect the most recent data, however they are approximate. The approximation error due to sampling is more manageable than staleness: (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. Sampling is known to be sensitive to skewed datasets, and we incorporate an outlier indexing technique to improve the accuracy of our approximation.

Since SVC is based on sampling, there are a subclass of views for which SVC can save significant computation and a subclass of queries on these views for which SVC can give accurate corrections. In this work, we explore these classes from both a theoretical perspective (i.e., when is our query correction optimal w.r.t estimate variance) and an empirical perspective for queries that do not satisfy the optimality conditions when is SVC still beneficial.

To summarize, our contributions are as follows: (1) we formalize maintenance of a sample of an MV as a data cleaning operation and we develop a framework to efficiently maintain (clean) this sample, (2) we extend the SampleClean approach to query processing on dirty data to support a wider set of general aggregate queries and show how this applies in the MV setting, (3) we use an outlier index to increase the accuracy of the approach for power-law, long-tailed, and skewed distributions, and (4) 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 7, we evaluate our approach. Finally, we discuss Related Work in Section 8 and present our Conclusions and Future Work in Section 9. We present full versions of our proofs and experimental details in our extended technical report [23].

2. BACKGROUND

In this section, we briefly overview our prior work and the new challenges in the materialized view setting.

2.1 SampleClean: Fast and Accurate Query Processing on Dirty Data

In our prior work on the SampleClean project [36], we proposed a framework for scalable aggregate query processing on dirty data. Similar to the accuracy-performance contrast between immediate maintenance and periodic maintenance in the materialized view setting, data cleaning also faces a similar challenge. Traditionally, data cleaning has explored expensive, up-front cleaning of entire datasets for increased query accuracy, and 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.

SampleClean has three parts: (1) sampling, (2) data cleaning, and (3) query result estimation. First, SampleClean creates a sample of dirty data (which are erroneous, missing, or otherwise corrupted records). Then, the framework applies a data cleaning procedure to the sample. Finally, when users query the dataset, the framework uses the clean sample to extrapolate clean query results. In this work, the main challenge was that data cleaning can potentially change the statistics of a sample and the queries need to compensate for those effects. In our initial work, SampleClean mainly focused on three common aggregates: `sum`, `avg`, and `count` queries.

The SampleClean project 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 result. Approach (1) is similar to those studied in the Approximate Query Processing (AQP) literature [3,21,29]. Approach (2), which we called NormalizedSC, outperformed (1) in datasets where data error was small or sparse. In the materialized view setting, we compare these approaches (see Section 7).

2.2 New Challenges

Applying SampleClean to materialized maintenance requries significant extensions to both the data cleaning, and query processing. In prior work, we modeled data cleaning as a row-by-row transformation of table. The data cleaning was a user-specified "black box" that operated on each row, and we applied this to a sample of dirty data. In this work, staleness is the data error and view maintenance is the cleaning; however, staleness is a very different sort of error than we considered before. Staleness can lead to rows that are missing from the "dirty" view or conversely need to be deleted. When coupled with sampling, this poses significant challenges since the statistics of the "dirty" (stale) sample are different from the "clean" (up-to-date) sample. We address this problem by formalizing the necessary updates to a sample materialized view as a data cleaning procedure. Since, we only have to propagate updates for a sample, there is further a new challenge in how to most efficiently execute this procedure to avoid irrelevant computation.

We also significantly extend SampleClean in terms of query processing. In SampleClean, NormalizedSC calculated the difference between the "dirty" and the "clean" sample, we extend this approach to settings where those samples do not match perfectly due to missing data. We also extend the generality of the framework to support queries other than the `sum`, `avg`, and `count` which studied before. Sampling is particularly sensitive to variance in the dataset, and large outliers can significantly reduce query accuracy, so in this work, we extend the query processing with an index of outliers.

2.3 Running Example: 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 title title, and the duration. Though SVC supports inserts, deletions, and updates (we formalize all three as data cleaning in Section 3.3.1), 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 three problems that SVC addresses: (1) how to efficiently maintain only a sample of a materialized view, (2) query processing on stale materialized

views with a sample of up-to-date data, and (3) using an outlier index to improve query accuracy. 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 select attributes $\{a_1, a_2, ..., a_k\}$ from, 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 select 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 $\ltimes, \rtimes, \bowtie$.
- $\gamma_{f,A}(R)$: 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 result of this operation is a row with the following schema $a \in A, f_a \in f(R)$ where a is the group by key and f_a is the aggregate of all rows with that key. A special case is the DISTINCT operation.
- $R_1 \cup R_2$: Set union take a union of the two sets.
- $R_1 \cap R_2$: Set intersection take an intersection of the set.
- $R_1 - R_2$: Set difference.

The composition of relational expressions can be represented as a tree, which we call 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 (or the result of another expression).

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 many different strategies to apply updates (maintenance) to S based on the insertions and deletions, 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'. $\mathcal{M}$ is formalism that can represent incremental update, recomputation, or a mix where some sub-expressions are recomputed and some are incrementally maintained.

Let, S be a materialized view. A maintenance strategy $\mathcal{M}$ is a relational expression the execution of which updates the view. 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.

Example: To make the concept of a maintenance strategy concrete, we show an example materialized view based on our running example dataset in Figure 2. Our example view joins the Log table with the Video table and counts the visits for each video grouped by id. 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 treating null as 0).

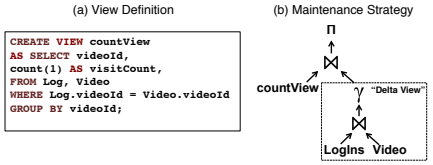

Figure 2: For our example, we represent the expression tree of the maintenance strategy. We first calculate a delta view using the new insertions and then join this view with the old view.

3.2 Sampling

First, we define what we mean by sampling. In this work, we focus on uniform samples of the rows in materialized views. 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 eg. $\hat{S}$ to denote sampled relations.

While, uniform sampling supports a wide variety of query types, it may have issues with queries with highly selective predicates. Stratfied sampling has been proposed to mitigate this problem as in the BlinkDB project [3]. 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

In SVC, we model staleness in a materialized view as a type of data error. We formalize the problem of correcting staleness as a data cleaning operation so we can apply our data cleaning approach. The staleness as a data error is clearly defined in the unsampled case. 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:

- A row in S needs to be updated.
- A row in S needs to be deleted.
- A new row needs to be inserted into S.

In the absence of these errors, we call a view *up-to-date*.

However, now suppose we have a sampled view $\hat{S}$. 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:

- If an update is needed, update the row.
- If the row needs to be deleted, delete the row.
- 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 way we 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 corrsponding stale samples (dirty data) and up-to-date samples (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 $\hat{S}$, $r \notin \hat{S}'$*

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 an efficient maintenance plan that produces $\hat{S'}$, a uniform sample of the up-to-date view 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 [2], 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:

```
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. From these queries, we exclude the group by clause, as we model group by clauses as part of the Condition.

3.3.3 Outlier Indexing

The query correction in the previous subsection is derived from a sample. Sampling is known to be sensitive to outliers, which we define as records whose values deviate significantly from the mean. However, a challenge is that since we do not materialize the entire up-to-date view detecting which records may be outliers is challenging. Instead, we define an outlier index on base relations of the database $\mathcal{D}$. This index tracks records whose attributes cross some threshold t. Then, for a given view S, this component gives a series of rules to propagate the information from the outlier index upwards. Basically, for every row in the view that is derived from a record in the outlier index, we ensure that it is incorporated into the sample. We use the set of outliers to return a more accurate correction result.

3.4 System Architecture

To summarize, the basic structure of SVC is: (1) analyze a view maintenance procedure to efficiently compute an up-to-date sample, (2) use this sample to correct stale query results, and (3) use an index of outliers make the sample robust to outliers in the data. The key insight is that SVC reduces the cost of view maintenance making it feasible to apply in resource-constrained settings where frequent maintenance was once impossible. In implementation, 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 query results for some queries. We illustrate this setup in Figure 1 in our introduction.

3.5 Example Application: Log Analysis

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. From this sample, we can estimate a correction (e.g., 40) of the old result. This means that we should correct the old result by 40 resulting in the estimate of $45 + 40 = 85$.

4. EFFICIENT MAINTENANCE WITH SAMPLING

In the previous section, we formalized the procedure of taking a uniform sample of 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 (i.e., not any easier than updating the entire materialized view) ways to accomplish this. For example, a naive solution to derive a sample $\hat{S'}$ is to just apply the maintenance strategy and then sample. However, this does not make the maintenance of the sample any more efficient.

Ideally, we want to integrate the sampling into the maintenance strategy $\mathcal{M}$ so that expensive operators need not operate on the full data. In this section, we discuss how to efficiently derive $\hat{S'}$ and the conditions under which maintaining $\hat{S'}$ is much cheaper than maintaining the entire view S'.

4.1 Uniform Sampling on Views

We first discuss some of the difficulties with uniform 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 [5]. 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 [12] and in approximate query processing [39]. 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*
- $\Pi_{(a_1,\ldots,a_k)}(R)$: *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 expressions 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 $h(a) \leq m$ samples close to a fraction m of the tuples.

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 Key: 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 $\eta_{b,m}(R_2)$.

Primary Key Many-to-one: If we are hashing the primary key of the result of a Foreign-Key join, the push down is possible. We have a join with two relations R_1 and R_2 and we know that for every $r_1 \in R_1$ there is exactly one r_2 in R_2 that satisfies the join condition. Based on the lineage rules defined earlier, the primary key is the union of the sets of primary keys of R_1 and R_2. However, since we know that there is only 1 r_2 for every r_1, it is equivalent to hash just the primary key of R_1. Thus, a in our hash function is the primary key of a Foreign-Key join, then we can push it down to R_1, $\eta_{a,m}(R_1)$.

(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.

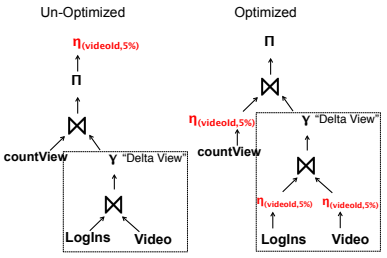

Figure 3: Applying the rules described in Section 4.3, we illustrate how to optimize the sampling of our example maintenance strategy.

4.4 Corresponding Samples

We showed that we can optimize a hashed maintenance plan $\eta(\mathcal{M})$ by using push-down rules. This gives us an expression to maintain 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.

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

PROPOSITION 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.

PROOF SKETCH. Since the primary keys are key consistent between $\hat{S'}$ and $\hat{S}$, included and excluded rows are preserved by the hashing. See the extended version for a full proof [23]. □

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.

Based on the rules described in Section 4.2, the primary key of the view is videoId. 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. This result is an equality the stale view (which has videoId as the primary key) making videoId the primary key of the epression

We can apply our sampling operator to this key, and use the push-down rules described in Section 4.3 to efficiently sample the maintenance strategy. In Figure 3, we illustrate the pushdown process. The the first 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 the stale view so we stop. Since the stale view does not change, we can calculate the sample of the stale view once (eg. 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 the previous section, we discussed how to efficiently maintain a sample of a materialized view by applying the hashing operator to the maintenance strategy. The result of this execution is an up-to-date sampled materialized view $\hat{S}'$. In this section, we discuss how to correct stale query results using the two corresponding samples $\hat{S}$ and $\hat{S}'$. The main idea is to take a point-wise difference of $\hat{S}'$ and $\hat{S}$, and apply the query to that difference. 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.

5.1 SUM, COUNT, and AVG

In our prior work, in data cleaning, NormalizedSC considered `sum`, `count`, and `avg` queries. NormalizedSC corrects a dirty query result, 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. However, this only considered data errors that updated rows, not those that required new rows to be inserted or deleted. In the materialized view 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. Our sampling procedure in the previous section, gives us two sample views that correspond with each other. 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 $R_1.a_1 = R_2.a_1$:*

$$\Pi_{R_1.a_2 - R_2.a_2}(R_1 \lessdot\!\!\!\!\gtrdot R_2)$$

If $R_1.a_1$ or $R_2.a_1$ is $\emptyset$ then they are respectively represented as a 0.

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 · 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`:

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

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

$\Delta\text{ans}_{avg} = \frac{\Delta\text{ans}_{sum}}{\Delta\text{ans}_{cnt}}$;

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

$agg(S') \approx agg(S) + \Delta\text{ans}_{avg,sum,count}$;

In each of these formulas, all of the Δans terms correspond to estimates, and these estimates can be bounded. In [36], 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 [36] 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 2. *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 [11]. It turns out that the proposed corrections are the optimal strategy when nothing is known about the data distribution a priori.

THEOREM 1. *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 an arbitrary non-parametric distribution, 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 [33]; however we calculate a correction and we explore the optimality of this correction. Then, we apply our correspondence subtract operator and since point-wise subtraction is commutative with these three queries, we can re-write Δans as a sample mean of the set of differences (where nulls are 0). Thus, Δans is an MVUE for the correction and since the stale result is deterministic it does not affect the estimate. See [23] for the full version of the proof. □

The implication of this theorem is that there does not exist any other estimation algorithm for Δans that has lower variance.

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 clean 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 [3]) 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. Without loss of generality, we assume this query is without a group by expression as we can always model this expression as part of the predicate. 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. Deterministic constants do not change the expected value of a random variable. The query result on the entire stale view is deterministic. Refer to our extended paper for more details [23]. □

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 [3] 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. There has been recent research in using techniques such as Poissonized resampling [2], analytical bootstrap [39], and bagging [22] to make this algorithm better suited for latency-sensitive query processing application. Refer to the extended technical report for the details on the bootstrap procedure [23].

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. Refer to the extended technical report for the details on `min` and `max` [23].

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 pred(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 [4,8]. Power-laws and other long-tailed distributions are common in large datasets [8]. We address this problem using a technique called outlier indexing which has been applied in SAQP [4]. The basic idea is that we create an index of outlier records (records whose attributes deviate from the expected value greatly) and ensure that these records are included in the sample.

6.1 Indices on the Base Relations

In [4], 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.

To select the threshold, there are many heuristics that we can use. For example, we can use our knowledge about the dataset to set a threshold. Or we can use prior information from the base table, a calculation which can be done in the background during the periodic maintenance cycles. If our size limit is k, we can find the records with the top k attributes in the base table as to set a threshold to maximally fill up our index. Then, the attribute value of the lowest record becomes the threshold t.

6.2 Adding Outliers to the Sample

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.

The next challenge is the outlier index must not require any additional effort to materialize. 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 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. 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 the standard TPCD benchmark and 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 log analysis with a dataset from a video streaming company. In this application, we look at the real query workload of the company and materialize views that could improve performance of these queries.

7.1 Experimental Setup

7.1.1 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 [6] 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 [25] 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. Refer to our extended paper on more details about the experimental setup [23]. 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 [17]. 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

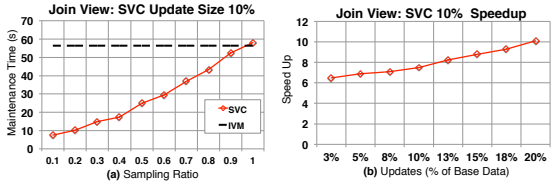

Figure 4: (a) On a 10GB dataset with 1GB of insertions and updates, we vary the sampling ratio and measure the maintenance time of SVC. The black line marks the time for full incremental view maintenance. (b) For a fixed sampling ratio of 10%, we vary the update size and plot the speedup compared to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

for each view.

7.1.2 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 [37]. In the most recent release of Spark, there is a SQL interface that allows users to persist query results in memory.

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. 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.

7.1.3 Metrics and Evaluation

To illustrate how SVC gives the user access to this new trade-off 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.

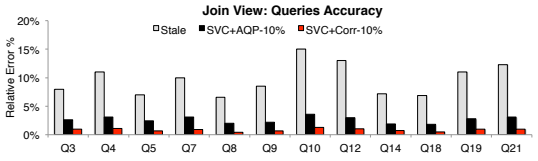

Figure 5: We generate 100 of each TPCD parameterized query and answer it using the stale materialized view, SVC+Corr, and SVC+AQP. We plot the median relative error for each query (since the result for each query might be multi-valued).

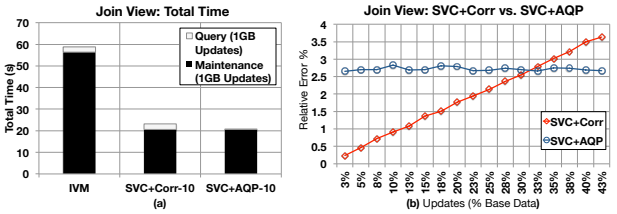

Figure 6: (a) For a fixed sampling ratio of 10% and update size of 10% (1GB), we measure the total time incremental maintenance + query time. SVC+Corr does take longer to query a view (due to the correction) than SVC+AQP and IVM, but this overhead is small relative to to the savings in maintenance time.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

7.2 Single-node Accuracy and Performance

7.2.1 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 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 calcu-

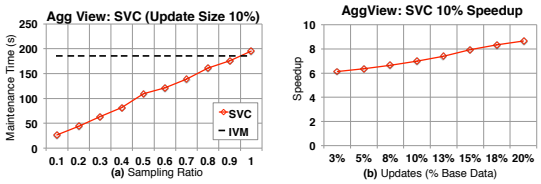

Figure 7: (a) In the aggregate view case, sampling can save significant maintenance time. (b) As the update size grows SVC tends towards an ideal speedup of 10x.

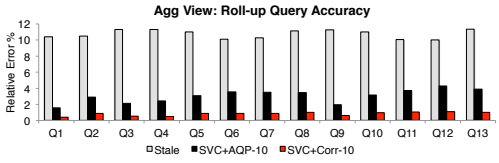

Figure 8: (a) We measure the accuracy of each of the roll-up aggregate queries on this view. For a 10% sample size and 10% update size, we find that SVC+Corr is more accurate than SVC+AQP and No Maintenance.

late 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 time maintenance 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.

7.2.2 *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 maintained in 26 seconds. As 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: In the extended version of the paper [23], we evaluate the median query. We found that for this query SVC+Corr was more accurate than SVC+AQP.

7.2.3 *Complex Views*

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

Performance: Figure 9, shows the maintenance time for a 10% sample compared to the full view. As before, we find that sampling saves a significant amount of maintenance time. However, 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.

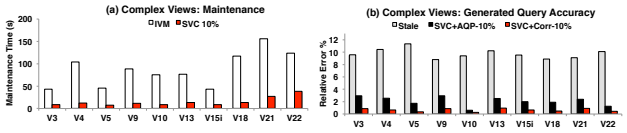

Figure 9: (a) For 1GB update size, we compare maintenance time and accuracy of SVC with a 10% sample on different views. V21 and V22 do not benefit as much from SVC due to nested query structures. (b) As before for a 10% sample size and 10% update size, SVC+Corr is more accurate than SVC+AQP and No Maintenance.

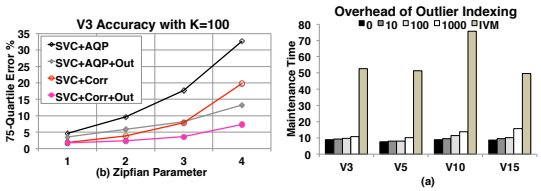

Figure 10: (a) For one view V3 and 1GB of updates, we plot the 75% quartile error with different techniques as we vary the skewness of the data. We find that SVC with an outlier index of size 100 is the most accurate. (b) While the outlier index adds an overhead this is small relative to the total maintenance time.

7.2.4 *Outlier Indexing*

In our next experiment, we evaluate our outlier indexing. Our outlier indices are constructed on the base relations. 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 push-up rules. These are four views dependent on l_extendedprice that were also in the set of "Complex" views chosen before. 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. In Figure 10(a), we find in the most skewed dataset SVC with outlier indexing reduces query error by a factor of 2. In the next Figure (Figure 10 (b)), we plot the overhead for outlier indexing for 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.

7.3 Conviva

In our next set of experiments, we applied SVC to a 1TB dataset of logs from Conviva.

7.3.1 *Performance and Accuracy*

We derive the views from 800GB of base data and add 80GB of updates. In Figure 11(a), we show that while full maintenance takes

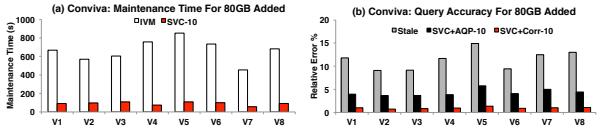

Figure 11: (a) We compare the maintenance time of SVC with a 10% sample and full incremental maintenance, and find that as with TPCD SVC saves significant maintenance time. (b) We also compare SVC+Corr to SVC+AQP and No Maintenance and find it is more accurate.

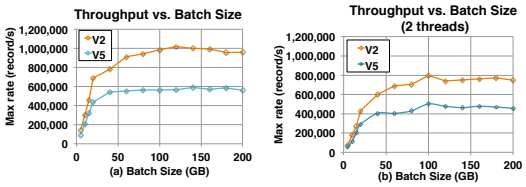

Figure 12: (a) Spark RDDs are most efficient when updated in batches. As batch sizes increase the system throughput increases. (b) When running multiple threads, the throughput reduces. However, larger batches are less affected by this reduction.

nearly 800 seconds for one of the views, a SVC-10% can complete sample maintenance in less than 100s for all of them. On average over all the views, SVC-10% gives a 7.5x speedup.

In Figure 11(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.

7.3.2 *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 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 [38], 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 12(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 12(b), we plot the throughput against batch size when two maintenance threads 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

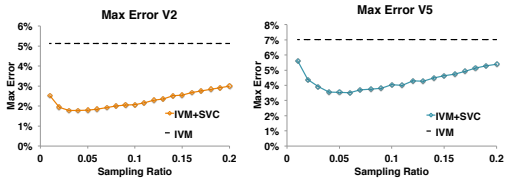

Figure 13: For a fixed throughput, SVC+Periodic Maintenance gives more accurate results for V2 and V5.

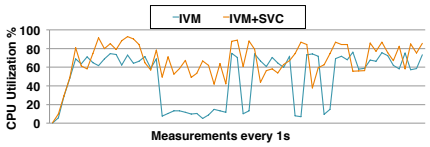

Figure 14: SVC better utilizes idle times in the cluster by maintaining the sample.

experiments (Figure 14), larger batch sizes are more amenable to parallel computation since there was more idle CPU time.

Choosing a Batch Size: The results in Figure 12(a) and Figure 12(b) show that larger batch sizes are more efficient, however, larger batch sizes also lead to more staleness. Combining the results in Figure 12(a) and Figure 12(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 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 13). 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 14, 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 typical deployments throughput demands are variable forcing maintenance periods to be longer, e.g., nightly. The same way that SVC takes advantage of micro idle times during communication steps, it can provide large gains during controlled idle times when no maintenance is going on concurrently.

8. RELATED WORK

Sampling has been well studied in the context of query processing [3,15,28]. Both the problems of efficiently sampling relations [28] and processing complex queries [2], 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 [12] and hashing. We look the problem of approximate query processing [2,3] 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 [34] for concurrency control. However, this work did not consider applications when the correction was applied to a sample as in SVC.

In the context of materialized view maintenance, sampling has primarily been studied from the perspective of maintaining samples [30]. Similarly, in [21], Joshi and Jermaine studied indexed materialized views that are amenable to random sampling. While similar in spirit (queries on the view are approximate), this work does not consider the cost of maintaining the materialized view only the cost of fast, approximate queries on the view. Nirkhiwale et al. [27], 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 [14,31,35]. 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 [13,18] 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 [10,16], 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.

9. CONCLUSION AND FUTURE WORK

Since view maintenance is often expensive, in practice, many prefer periodic maintenance solutions. SVC uses sampled-based data cleaning approach to trade-off computation and a query result ccuracy in the materialized view setting. This trade-off allows more frequent maintenance to be applied in settings where before system resource constraints made impossible and the result is that between maintenance periods users can get more accurate query results.

We greatly expanded the scope of the SampleClean project [36] to general aggregate queries, missing data errors, and proved when optimality of the estimates hold. For materialized views, we found that our data cleaning approach can significantly reduce the maintenance time for a large class of materialized views, while still providing accurate aggregate query answers. The materialized view setting also encouraged us to consider the relational algebra of sampling, and we devised a hash-based approach to efficiently restrict general view maintenance procedures to a sample. One concern with sampling is its sensitivity to data skew, and our results suggest that outlier indexing can mitigate these concerns.

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

10. REFERENCES

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

[2] 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.

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

[4] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. R. Narasayya. Overcoming limitations of sampling for aggregation queries. In *ICDE*, pages 534–542, 2001.

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

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

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

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

[9] 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.

[10] 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.

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

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

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

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

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

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

[17] 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.

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

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

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

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

[22] A. Kleiner, A. Talwalkar, S. Agarwal, I. Stoica, and M. I. Jordan. A general bootstrap performance diagnostic. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*, pages 419–427, 2013.

[23] S. Krishnan, J. W. M. J. Franklin, K. Goldberg, and T. Kraska. Sample-view-clean: A data cleaning approach for fresh query answers from stale materialized views. 2014.

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

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

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

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

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

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

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

[31] 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.

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

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

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

[35] 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.

[36] 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.

[37] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[38] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438, 2013.

[39] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo. The analytical bootstrap: a new method for fast error estimation in approximate query processing. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 277–288, 2014.

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

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