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

ABSTRACT

Materialized views, stored pre-computed query results, are widely used to facilitate fast queries on large datasets. In the Big Data era when new data arrives at an increasingly fast rate, maintaining materialized views can be costly. Due to this cost, maintenance is often deferred to a later time, but at the new cost of returning inaccurate query results on stale views. In this paper, we address this problem from a data cleaning perspective and model staleness as a type of data error. We take inspiration from recent data cleaning results that greatly improve query accuracy by cleaning only a small sample of dirty data. To complement existing deferred maintenance approaches, we maintain a sample of a materialized view to correct stale query results between maintenance periods, without the cost of full maintenance. This requires an analysis of the view maintenace procedure to efficiently maintain only a sample of a view. We use this sample to estimate how the updates affect a query on the view, and correct the query results based on this estimate. We show any query that has an unbiased sample estimator can be reformulated to give an unbiased correction, and for the most common aggregate queries (SUM,COUNT, AVG) our corrections are optimal with respect to estimate variance. As sampling can be sensitive to long-tailed distributions, we further explore an outlier indexing technique to give increased accuracy when the data distributions are skewed. We evaluate our method on real and synthetic datasets and results suggest that we are able to provide more accurate query results without having to maintain the full view.

1. INTRODUCTION

Materialized views (MVs), storing pre-computed query results, are a well-studied approach to speed up queries on large datasets [10,26,27,35]. 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 [5,36,45] and for more advanced analytics based on linear algebra and machine learning [39,59].

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 recomputing the query. There has been substantial work in deriving these incremental updates (incremental maintenance algorithms) for different classes of MVs and optimizing their execution [10,24–26,33,49,52].

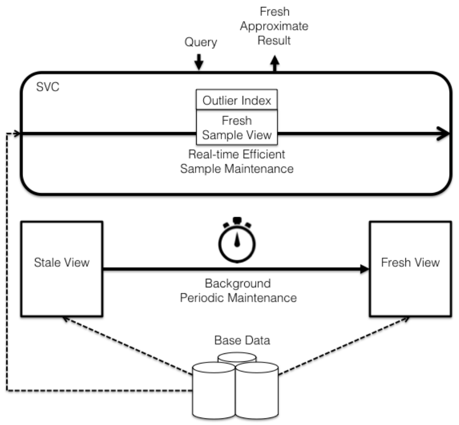


Figure 1: If a view is being periodically maintained, in the interim, query results may be stale. Sample-View-Clean sits above an existing view maintenance architecture, and provides an interface for approximately up-to-date query results by maintaining a sample. The user can tune the sampling ratio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

For frequently changing tables even incremental maintenance can be expensive; every update to the underlying 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 [10,12,60] so that updates can be batched together to amortize overheads and maintenance work can be scheduled for times of low system utilization.

While deferring maintenance has compelling benefits, it unfortunately brings its own costs, namely that views become increasingly stale in-between maintenance periods. As a result, queries on those views can return increasingly incorrect answers. The problem of stale MVs parallels the problem of dirty data studied in data cleaning [47]; as both staleness and erroneous records lead to inaccurate query answers. The observation that stale MVs are a type of dirty data leads us to the key insight behind our work; namely, that data cleaning techniques can be used to mitigate the negative impacts of deferred MV maintenance.

Data cleaning has been studied extensively in the literature (e.g., see Rahm and Do for a survey [47]) 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 Sample-

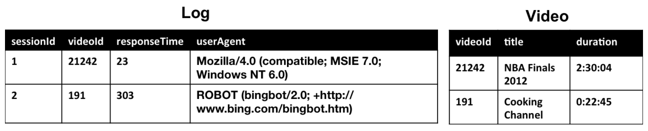
Clean framework to greatly improve query accuracy while cleaning only a small sample of dirty records [54]. We proposed an approach called NormalizedSC that corrects dirty query results by using a sample of clean data to learn how the dirtiness affects that query and then calculates a correction. This perspective raises a new possibility for MVs: we can use a sample of clean (up-to-date) data 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 sampling, cleaning, 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, 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 in expectation up-to-date. While they are approximate results, the approximation error 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 as opposed to potentially unbounded staleness. Sampling is known to be sensitive to skewed datasets, and we incorporate an outlier indexing technique to improve the accuracy of our approximate query results.

In practice, both MVs and queries can be arbitrarily complex. 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:

- We model the incremental maintenance problem as a data cleaning problem and staleness as a type of data error.
- We show how sampling with a `hash` operation can reduce computation during maintenance.
- We derive a correction for aggregate queries using the sample and show that, in fact, our correction is optimal for `sum`, `count`, and `avg`.
- We use an outlier index to increase the accuracy of the approach for power-law, long-tailed, and skewed distributions.
- We evaluate our approach on a single-node MySQL database with a 10GB skewed TPCD benchmark dataset and on a 20-node Apache Spark cluster with a 1TB log dataset from a video streaming company. Our results show that sampling is significantly faster than full view maintenance, and also can give highly accurate results for a variety of queries and views.


Log

sessionId	videoId	responseTime	userAgent
1	21242	23	Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0)
2	191	303	ROBOT (bingbot/2.0; +http:// www.bing.com/bingbot.htm)

Video

videoId	title	duration
21242	NBA Finals 2012	2:30:04
191	Cooking Channel	0:22:45

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

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 Section 4 and 5, we describe the sampling and query processing of our technique. In Section 6, we describe the outlier indexing framework. In Section 7, we discuss extensions to our framework. Then, in Section 8, we evaluate our approach. Finally, we discuss Related Work in Section 9 and present our Conclusions and Future Work in Section 10.

2. BACKGROUND

In this section, we briefly overview the current challenges with view maintenance and our prior work in scalable data cleaning.

2.1 Running Example: Log Analysis

To illustrate our framework, we use the following running example which is a simplified schema of one of our experimental datasets (Figure 2). Imagine, we are querying logs from a video streaming company. These logs record visits from users as they happen and grow over time. We have two tables, Log and Video, with the following schema:

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

These tables are related with a foreign-key relationship between Log and Video. Though SVC supports inserts, deletions, and updates, for clarity in our example, we consider insertions into Log which is cached in a temporary table:

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

2.2 Materialized View Maintenance

Views define logical relations which can be queried instead of physical base relations. MVs are a class of views that are pre-computed 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 [32,33].

Incremental maintenance may not be efficient in all cases. Consider the view that calculates the median responseTime grouped by userAgent on our running example dataset. In general, to ensure correctness, the view has to store the entire set of responseTime attributes for each group to allow for incremental maintenance. Along the lines of this example, there are cases when recomputation may require less storage of state or even less computation. Thus, materialized views are maintained either with incremental maintenance, recomputation, or a mix.

2.2.1 *Practical Considerations*

In real-world systems, for large datasets or fast data update rate, it may not always be feasible to maintain MVs immediately. Therefore, deferred maintenance 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 either in a pre-set way or adaptively. In deferred maintenance approaches, the user often accepts some degree of staleness for additional flexibility in scheduling. These costs can also be deferred to query execution time. In particular, we highlight a technique called lazy maintenance which applies updates to the view only when a user's query requires a row [60]. While always fresh, both lazy maintenance and immediate maintenance hit a bottleneck when there are rapid updates, and this results increasingly degraded performance if a user wants to query a view. The alternative is a periodic strategy, but this means that there is unbounded error on queries between maintenance periods.

The data cleaning perspective that SVC offers on this problem is that there is a trade-off between accuracy and computation. 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.

2.3 SampleClean: Fast and Accurate Query Processing on Dirty Data

In our prior work on the SampleClean project [54], we proposed a framework for scalable data cleaning. 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 (Figure 3) 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 [4,30,42]; however in cases when data errors were small, we found that the approximation error dominated. To address this problem, we developed approach (2), which we called NormalizedSC, and we found that NormalizedSC was more accurate in mostly clean datasets as it leverages existing deterministic results leading to reduced approximation error. In the materialized view setting, we found that NormalizedSC led to more accurate results in our experiments (see Section 8).

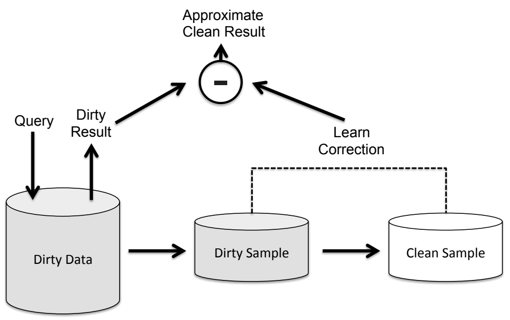


Figure 3: A basic overview of SampleClean. SampleClean uses a random sample of dirty data to learn how a data cleaning algorithm affects queries on the sample. We can then derive a correction to compensate for the dirtiness.

2.4 New Challenges

Inspired by SampleClean, SVC samples a stale view, cleans the sample view by restricting the maintenance to just the rows in the sample, and then applies NormalizedSC to correct the results of queries on the stale view. Applying this data cleaning framework to the materialized view setting leads to some interesting theoretical challenges with new insights for both materialized view maintenance and data cleaning. In SampleClean, we simply treated the data cleaning as a black box and did not study how to clean a dirty sample data. However, in SVC, we cannot make such assumption and have to devise efficient data-cleaning techniques to "clean" a stale sample view.

Staleness is a new type of data error. In materialized views, staleness can lead to rows that are missing from the "dirty" view or conversely need to be deleted. These issues pose new challenges in query correction. We further explore and formalize the class of queries that SVC can support. We extend the generality of the framework to support queries 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. In this work, we give an explicit treatment of outliers records.

3. FRAMEWORK OVERVIEW

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 resourceconstrained settings where frequent maintenance was once impossible. We first define notation, terminology, and the problem setting that we address in this work. Then, we formalize the three problems that the different components of SVC address: (1) analyzing a maintenance procedure to efficiently maintain a sample, (2) query processing using the sample, and (3) using an outlier index. Finally, we overview the system architecture and discuss a toy numerical example of how this works in practice.

3.1 Notation

Let $\mathcal{D}$ be a database which is a collection of relations $\{R_i\}$, and let S be a materialized view of this database. We denote the set of insertions to a relation R_i as ΔR_i and deletions as ∇R_i. In this paper, we refer to ΔR_i and ∇R_i as "delta relations". An update to a relation can be modeled as a deletion and then an insertion.

When the base relations have been updated (there exists insertions and/or deletions), queries on the view S will be stale. We call the procedure to update the view as a *maintenance strategy* $\mathcal{M}$. A maintenance strategy is a relational expression the execution of which will update the view S, and we call the updated view S'.

DEFINITION 1 (MAINTENANCE STRATEGY). *Let, $\mathcal{D}$ be a database which is a collection of relations $\{R_i\}$. Let, S be a materialized view. We denote the insertions to a relation R_i as ΔR_i and the deletions as ∇R_i. A maintenance strategy $\mathcal{M}$ is a relational expression the execution of which updates the view. It is a function of the database $\mathcal{D}$, S, and all the delta relations $\{\Delta R_i\} \cup \{\nabla R_i\}$.*

We look at maintenance strategies composed of the following relational operations:

- $\sigma_\phi(R)$: Selection select all tuples r from R that satisfy the restriction $\phi(r)$
- $\Pi_{a_1,a_2,...,a_k}(R)$: Projection select attributes $\{a_1, a_2, ..., a_k\}$ from R
- $\bowtie_{\phi(r1,r2)}$ (R_1, R_2): Join select all tuples in $R_1 \times R_2$ that satisfy $\phi(r_1, r_2)$.
- $\gamma_{f,A}(R)$: 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 output schema is of the result is a, f_i. [1]
- $R_1 \cup R_2$: 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.

Relational expressions can be expressed in a tree form which we call an *expression tree.*

To illustrate the concept of a maintenance strategy, consider the following view that finds view counts for all videos:

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

[1]A special case of this operator is δ the deduplication operator i.e. f always returns null. Another special case, is when there is a expression in the group by clause which we treat as creating a set of virtual columns.

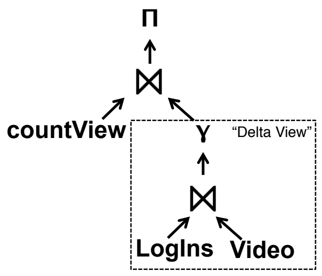


Figure 4: For our example, we represent the expression tree of the maintenance strategy. We first calculate a delta view using the new insertions and then join this view with the old view.

If new records have been added to Log and Video, then the maintenance strategy consists of the following expressions: (1) create a "delta view" by applying the view definition to LogIns, (2) update the view by taking the join of the "delta view" with countView, (3) project the result by incrementing visitCount. We illustrate this process with the expression tree in Figure 4.

3.2 Sampling

In this work, we focus on uniform samples of 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. To differentiate sampled relations from un-sampled ones we use the "hat" notation eg. $\hat{S}$.

While, uniform sampling supports a wide variety of query types, it may have issues with queries with highly selective predicates. In principle, our framework can be extended to handle stratified samples similar to 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.

Note, that this definition is slightly different from the reservoir sampling techniques studied in AQP [53] which find a uniform sample of fixed *size* $k \leq | S |$. Our sampling ratio gives a sample of the size k in expectation, however, the actual size from any given instance may be slightly different. For large sample sizes, there is little difference between the techniques since the actual size of using a sample ratio will be close to k. The uniform sample model represents our algorithm which uses hashing better and also makes the presentation of our analysis more clear.

Furthermore, any "black-box" uniform sampling algorithm can be used to achieve a reservoir sample. The use of one technique over another does not affect the general principles or the statistics of SVC, only the notation in the analysis.

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. Materialized views whose base relations have been updated potentially give stale query results. We formalize the problem of correcting staleness as 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:

- 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 this case, it is clear that simply executing the maintenance strategy $\mathcal{M}$ will achieve the necessary results and update the view S to view S'. However, now suppose we have a sampled view $\hat{S}$, and we want to do just enough effort to update only the rows in the sample. This problem is challenging since the errors are not just updates to rows in the views, as there might be new insertions needed. Thus, we define cleaning in the following way: suppose we have a stale uniform sample $\hat{S}$ with sampling ratio m, we want to find an up-to-date uniform sample $\hat{S'}$. We define the cleaning procedure as preserving the uniformity of sampling, so after cleaning $\hat{S'}$ is a uniform sample of S':

- 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.

It is important to note that the way we defined data cleaning on a sample does not necessarily give a unique $\hat{S'}$. In this paper, we take advantage of this property and propose a technique that analyzes $\mathcal{M}$ and can find a particular $\hat{S'}$ efficiently.

In, SampleClean [54], NormalizedSC query result estimation is done taking a row-by-row difference of the dirty and cleaned data. However, in prior work, we did not consider the effect of deletions or insertions so taking this difference was well defined. In this application, we have to define a new property to handle this problem, which we call "correspondence". SVC compares two "corresponding" samples to derive a correction.

Definition 2 (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, $r \in \hat{S'}$

For every row r in $\hat{S}$ that was unchanged, $r \in \hat{S'}$

For every row r in S but not in $\hat{S}$, $r \notin \hat{S'}$

This definition of correspondence gives us a way to get two samples from which we can take a row-by-row difference. There is some nuance in how to handle null values which we discuss in Section 5.

Example 1 (Correspondence). *Suppose countView has 4 videos:*

```
V1 (visitCount = 4), V2 (visitCount = 6), V3 (
    visitCount = 1), V4 (visitCount = 1)
```

We take a sample of countView and call it countViewSample that contains V1 and V2. LogIns has new logs of 1 visit for V1 and 1 visit for a new video V5. An up-to-date sample that corresponds is:

```
V1 (visitCount = 4+1), V2 (visitCount = 6)
```

An up-to-date sample that does not *corresponds is:*

```
V1 (visitCount = 4+1), V3 (visitCount = 1)
```

This is because V2 was unchanged and therefore should be included in the sample.

3.3.2 Query Correction

Given a query q which has been applied to the stale view $q(S)$ giving a stale result. Our query correction component takes two corresponding samples $\hat{S'}$ and $\hat{S}$, and calculates a correction to $q(S)$. Like similar restrictions in other sampled-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:

```
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 simple predicates.

We also consider correcting stale non-nested select queries of the following form with simple 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 explore the conditions under which we can make this guarantee, and discuss query processing with the outlier index in Section 6.

3.4 System Architecture

In implementation, SVC works in conjunction with existing periodic maintenance or periodic re-calculation. 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

{{Make sure that we do a good survey on "sampling from a view" and discuss them in the related work, e.g., [Frank et al., VLDB 86], [Nirkhiwale et al., VLDB 13]}} In the previous section, we formalized the computation of $\hat{S'}$ as a data cleaning procedure. 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 Formalizing Uniform Sampling

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

Definition 3 (Uniform Sample). *We say the relation $\hat{S'}$ is a* uniform sample *if*

$$\forall s \in \hat{S'} : s \in S'$$

and

$$Pr(s_1 \in \hat{S'}) = Pr(s_2 \in \hat{S'}) = m$$

A traditional "coin-flip" sampling algorithm is not suited for this property as it is known that such sampling commutes very poorly many relational operations such as joins and aggregates [8]. Recall, the view in our example countView. If we sampled the base relation the result would have a mix of missing rows from the view and rows with incorrect aggregates. However, this is not what we require. We want 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 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 [14] and in approximate query processing [58]. **{{Add Kai into acknowledgement for helping us with problem formulation. }}** We recursively define a set of consistent primary keys for all nodes in the expression tree:

Definition 4 (Primary Key). *For every relational expression R, we define the primary key of every subexpression 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 key of R_1 and R_2.*
- *$\gamma_{f,A}(R)$: The primary key of the result is the tuple A.*
- *$R_1 \cup R_2$: 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. Our definition of the primary key is also constructive; that is, if an expression has a null primary key then we modify every projection operation to ensure that primary key of the subrelation is never projected out.

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 on average 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 than we can push η down the query 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 5 (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.*
- *$\Pi_{p,[a_2,...,a_k]}(R)$: Push η through if a is in the projection.*
- *$\bowtie_{\phi(r1,r2)} (R_1, R_2)$: Blocks η in general.*
- *$\gamma_{f,A}(R)$: 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.

Many-to-one Join: If we have an join with two relations R_1 and R_2 and we know that for every $r_1 \in R_1$ there is at most one r_2 in R_2 that satisfies the join condition, then we push η down to R_1.

One-to-one Join: If we have the previous condition and also the converse is true for every $r_2 \in R_2$ there is at most one r_1 in R_1, then we push η down to R_1 and R_2.

(Semi/Anti)-Join: We can always push η down on Semi-joins. For anti-joins we can push η down because we can rewrite the node as $R_1 \dot{-} (R_1 \ltimes R_2)$ and apply the push-down rules for set difference and Semi-Joins.

4.4 Hashing and Correspondence

Another benefit of deterministic hashing is that when applied in conjunction to the primary keys of a view, we get the Correspondence Property (Definition 2) 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. 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 next section to get estimates for queries on the materialized view.

4.5 Example

We will illustrate our proposed approach on our example view countView. 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. We can apply our sampling operator to this key, and use the pushdown rules described in Section 4.3 to efficiently sample the maintenance strategy.

In Figure 5, 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 one-to-one join which merges the aggregate from the "delta view" to the old view allowing us to push

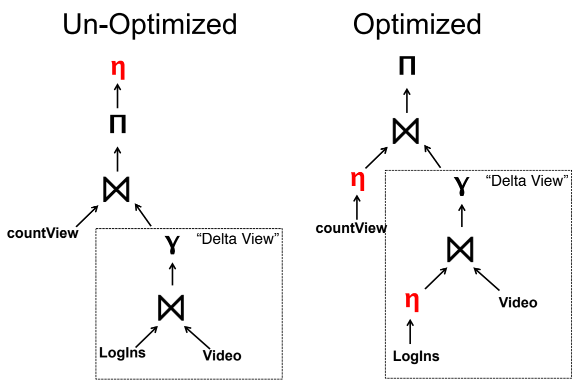


Figure 5: Applying the rules described in Section 4.3, we illustrate how to optimize the sampling of our example maintenance plan.

down on both branches of the tree. On the left side, we reach the the stale view so we stop. 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 a many-to-one (foreign key) join allowing us to push down the sampling to the "many" relation LogIns.

In terms of increased efficiency, since both the aggregate and foreign key join 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 find $\hat{S}'$ 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 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 that have unbiased estimators. We also show how SampleClean can work with biased estimates as well and what statistical tools we can use to bound these estimates. Next, we discuss selection queries and how SampleClean can give limited support to these. We summarize these results in Table 1, where we list each of the aggregate queries supported in Apache HiveQL and describe how SVC estimates its results. Finally, we analyze the cost of query correction.

5.1 SUM, COUNT, and AVG

In our prior work, in the context of data cleaning, NormalizedSC considered `sum`, `count`, and `avg` of the form:

SELECT $f(a)$ FROM View
WHERE Cond(A)

Group-by clauses can be handled as part of the predicate. As input NormalizedSC is given a query (`sum`, `count`, or `avg`), the result of running this query on the dirty data, and a dirty sample and a cleaned version of the sample. We then calculate the row-by-row difference for each row in the sample between dirty and clean. Then, we apply the query to the sample set of differences scaling appropriately (eg. for `sum`). 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, NormalizedSC only considered data errors that

Table 1: Query Result Semantics

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

transformed 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. This makes the a row-by-row subtraction between two samples of data ill-defined.

In Definition 2, 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 subtraction. 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 6 (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 $R_1.a_1 = R_2.a_1$:*

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

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

Then, we rewrite the `sum`, `count`, and `avg` queries using a selection with a **case** statement, and use the primary key (pk) that we defined in the previous section. Let $\text{Cond}(\hat{S}') = 1$ ($\text{Cond}(\hat{S}) = 1$) be **case** statements to denote that its old (new) record satisfies the query condition; otherwise, $\text{Cond}(\hat{S}') = 0$ ($\text{Cond}(\hat{S}) = 0$).

For `sum`:

```
q(View) := SELECT pk, $a \cdot \text{Cond}(A)$ FROM View
```

and for `count`:

```
q(View) := SELECT pk, Cond($A$) 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`, `count`, and `avg`, respectively:

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

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

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

We use the estimated correction to correct the stale query result as follows:

$\mathtt{sum}(S') \approx \mathtt{sum}(S) + \Delta\mathsf{ans}_{sum}$;

$\mathtt{count}(S') \approx \mathtt{count}(S) + \Delta\mathsf{ans}_{count}$;

$\mathtt{avg}(S') \approx \mathtt{avg}(S) + \Delta\mathsf{ans}_{avg}$;

In each of these formulas, all of the $\Delta\mathsf{ans}$ terms correspond to estimates, and these estimates can be bounded. In [54], we showed that the corrections for three queries can be rewritten as a sample mean. A sample mean is an unbiased estimator of a population mean for all data distributions and sample sizes. Furthermore, the Central Limit Theorem states that asymptotically this mean approaches a normal distribution giving very tight bounds on the error.

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.[2] We can use this to bound the term with its 95% confidence interval (or any other user specified probability), e.g., $\bar{X} \pm 1.96\frac{\sigma}{\sqrt{k}}$; refer to our prior work [54] for further details.

In the statistical literature, the class of aggregate functions for which we can apply this technique is formalized as the class of U-statistics, see [29].

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 [13]. Unbiased estimators are ones that, in expectation, are correct. However, on its own, the concept of an unbiased estimate is not useful as we can construct bad unbiased estimates. For example, if we simply pick a random element from a set, it is still an unbiased estimate of the mean of the set. The variance of the estimate determines the size of our confidence intervals thus is important to consider.

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

Theorem 1. *For `sum`, `count`, and `avg` queries, our estimate of the correction is optimal over the class of linear*

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

estimators when no other information is known about the distribution. In other words, there exists no other linear function of the set input rows $\{X_i\}$ that gives a lower variance correction.

Proof Sketch. As discussed before, we reformulate `sum`, `count`, and `avg` as means over the entire table. We prove the theorem by induction. Suppose, we have two independent unbiased estimates of the mean $\bar{X}_1$ and $\bar{X}_2$ and we want to merge them into one estimate $\bar{X} = c_1\bar{X}_1 + c_2\bar{X}_2$. Since both estimates are unbiased, $c_1 + c_2 = 1$ to ensure the merged estimate is also unbiased, and now we consider the variance of the merged estimate. The variance of the estimate is $var(\bar{X}) = c_1^2 var(\bar{X}_1) + c_2^2 var(\bar{X}_2)$. Since, we don't know anything about the distribution of the data, by symmetry $var(\bar{X}_1) = var(\bar{X}_2)$. Consequently, the optimal choice is $c_1 = c_2 = 0.5$. We can recursively induct, and we find that if we do not know the variance of data, as is impossible with new updates, then equally weighting all input rows is optimal. □

The implication of this proof is that there does not exist any other weighted average that gives better estimate (for any distribution or sample size).

5.2 General Aggregate Queries

In SampleClean, we proposed the NormalizedSC algorithm to give unbiased corrections to `sum`, `count`, and `avg`. In this work, 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. The implications of this are that any query that can be answered in prior work with SAQP (eg. 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. 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 a factor of 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:

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

□

However, in general, these aggregate functions do not satisfy the optimality conditions of the previous section. 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, and (4) finally take the difference between (2) and (1) and add it to (3).

This procedure will also work for `sum`, `count`, and `avg` and in fact will give the same result estimate. However, unlike before, it does not give us an analytic way to calculate a confidence interval. This is because variance does not commute with subtraction for correlated random variables. 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 repeat executions to build a distribution of values that the correction can take allowing us to bound the result. The conditions under which this technique is valid are nuanced, and detecting when it is valid is a subject of [3].

Proposition 3. *(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.

It is true that bootstrap is more computationally expensive than the analytical derivations for the `sum`, `count`, and `avg` queries, and it is also sensitive to the choice of parameters b_1 and b_2. There has been recent research in using techniques such as Poissonized resampling [3], analytical bootstrap [58], and bagging [31] to make this algorithm better suited for latency-sensitive query processing application.

5.3 Bounded Bias Aggregates

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.*

What this means for a user is that the error bars are asymmetric with increased error in the direction of the bias (often in the direction of skew in the dataset). Functionally, we can apply the same technique discussed in the previous section for these queries and apply the bootstrap to bound the results.

5.4 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 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.

5.5 Select Queries

We can correct stale Select queries with SVC. Suppose, we have a Select query with a simple predicate:

```
SELECT * FROM View
WHERE Cond(A)
```

We can first run the Select query on the stale view, and the result of this is a multiset. First, we must define what it means to "correct" this stale result. 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 aggregate 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 not updated, added, or deleted (thus three "confidence" intervals).

5.6 Query Execution Cost

It is true that SVC adds to the query execution time by issuing a correction. However, this correction is only calculated on a sample and thus is small compared to the query execution time over the entire old view. In our experiments, we show that the overheads are small in comparison to the execution time over the entire old view and the savings from only maintaining a sample (Section 8.3.2).

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 SAQP [7]. 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. 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], outlier indices are defined on the sampled relation. In SVC, we sample the materialized view and only maintain those rows in the sample. Therefore, there is a problem since the outlier records are not materialized. To address this problem, we define the outlier index on the base relation.

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

Given this index, the next question is how we can use this information in our materialized views. We ensure that any row in a materialized view that derives 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 query 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 plan.

DEFINITION 7 (OUTLIER INDEX PUSHUP). *Define an outlier index to be a triple 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 (I, 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 $(I_1 \cup I_2, O_1 \bowtie O_2)$.*
- *$\gamma_{f,A}(R)$: Recall our earlier notation where the resulting schema of the group-by aggregation was (A, f_i). For group-by aggregates, we set $I = \{f_i\}$. 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 with the Outlier Index

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 AND DISCUSSION

7.1 Correction vs. Direct Estimate

We argued that in many cases correcting query results approximately is more accurate than an AQP-style direct estimate. We can provide some back-of-the-envelope 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 data:

$$\frac{\sigma_{S'}^2}{k}$$

Since the change is the difference between the stale and up-to-date data, this can be rewritten as:

$$\frac{\sigma_S^2 + \sigma_{S'}^2 - 2cov(S, S')}{k}$$

Therefore, a correction will be more precise when:

$$\sigma_S^2 - 2cov(S, S') \leq 0$$

If the difference is small, i.e. S is nearly identical to S',then $cov(S, S') \approx \sigma_S^2$ and it is clearly less than 0. On the other hand, there is a break even point where changes are significant enough that direct estimates might be more accurate.

7.2 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.

7.3 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.

7.4 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 a data cube application, 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. We implement SVC in Spark 1.1 and deploy it on a 10-node cluster to analyze 1TB of logs.

8.1 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 [38] 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 [22]. 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). The specification of this data cube and the queries are listed in the appendix **{{TR}}**.

Complex Views: 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, and the reasons why queries were excluded is listed in the appendix **{{TR}}**.

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 eg. if the attribute is country then the predicate can be countryCode > 50 and countryCode < 100. We generated sum, avg, and count of the following form:

```
SELECT f(b) FROM View
WHERE subset of domain of a;
```

We generated 100 random aggregate queries for each view.

8.2 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 [56]. 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 1TB 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. In this workload, there were annotated summary statistics queries, and we filtered for the most common types. 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.

We used this dataset to evaluate the end-to-end accuracy and performance of the system in a real-world application.

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.

8.2.1 *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.

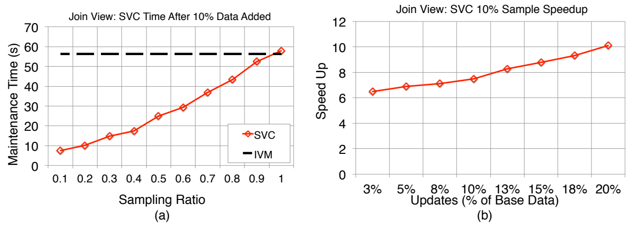


Figure 6: (a) On a 10GB dataset with 1GB of insertions and updates, we vary the sampling ratio and measure the maintenance time of SVC. The black line marks the time for full incremental view maintenance. (b) For a fixed sampling ratio of 10%, we vary the update size and plot the speedup compared to full incremental maintenance for SVC.

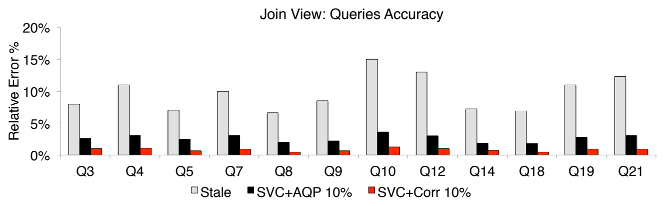


Figure 7: We generate 100 of each TPCD parameterized query and answer it using the stale materialized view, SVC+Corr, and SVC+AQP. We plot the median relative error for each query (since the result for each query might be multi-valued). Our experiments suggest that on this view, SVC+Corr is more accurate than SVC+AQP and No Maintenance.

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
$$\frac{\mid r - r' \mid}{r}.$$
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.

8.3 Single-node Accuracy and Performance

8.3.1 *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. Figure 6(a), shows the maintenance time of SVC as a function of sample size. With the bolded dashed line, we note the time for full

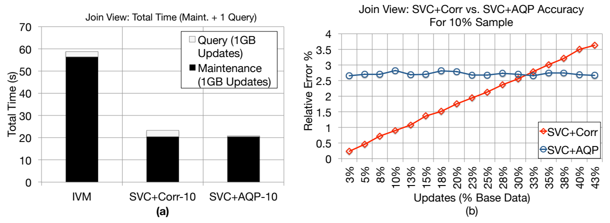


Figure 8: (a) For a fixed sampling ratio of 10% and update size of 10% (1GB), we measure the total time incremental maintenance + query time. SVC+Corr does take longer to query a view (due to the correction) than SVC+AQP and IVM, but this overhead is small relative to to the savings in maintenance time. (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).

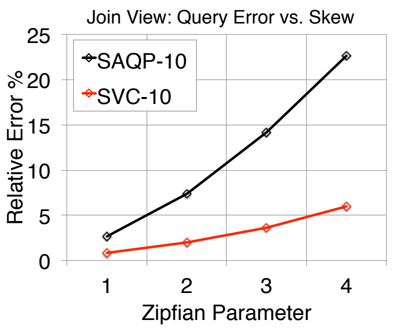


Figure 9: We fix the update size at 1GB and the sampling ratio at 10% and vary the skew of the dataset. The gap between SVC+AQP and SVC increases in more skew datasets suggesting that corrections are more robust to skew.

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.

Performance: The speedup for SVC-10 is 7.5x which is far from ideal on a 10% sample. In the next figure, Figure 6b, 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 7, 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 was 11.7x more accurate than the stale baseline, and 3.1x more accurate

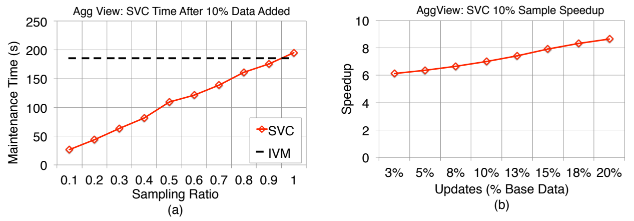


Figure 10: (a) In the aggregate view case, sampling can save significant maintenance time. (b) As the update size grows SVC tends towards an ideal speedup of 10x.

than applying SVC+Corr 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, Accordingly, 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. We evaluate this overhead in Figure 8a, where we compare the total time maintenance and query execution time.

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 8b, 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.

Effect of Skew: Finally in Figure 9, we vary the Zipfian parameter and show how the accuracy of SVC+Corr and SVC+AQP changes. While both techniques are sensitive to skew, we find that the gap between SVC+Corr and SVC+AQP widens for more skewed datasets. SVC's accuracy is dependent on the variance of the difference between tuples in the up-to-date view and the stale view; which is different from SVC+AQP which is dependent on the variance of the tuples themselves. Even though a dataset might be skewed, if (in relative terms) the updates are more uniform SVC+Corr will give more accurate results. We can make SVC+Corr and SVC+AQP even more accurate using the outlier indexing which we will evaluate in the later sections.

8.3.2 *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 10(a)). It takes 186 seconds

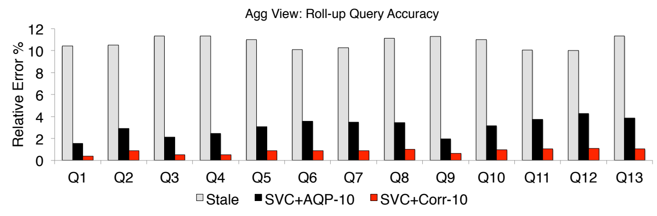


Figure 11: We measure the accuracy of each of the roll-up aggregate queries on this view. For a 10% sample size and 10% update size, we find that SVC+Corr is more accurate than SVC+AQP and No Maintenance.

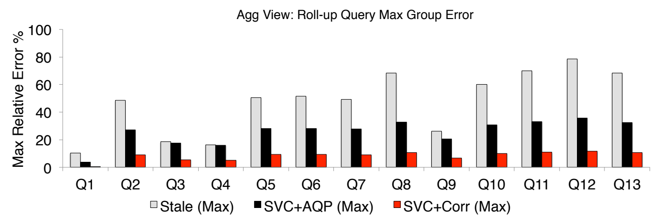


Figure 12: For 1GB of updates, we plot the max error as opposed to the median error in the previous experiments. Even though updates are 10% of the dataset size, some queries are nearly 80% incorrect. SVC helps significantly mitigate this error.

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 10(b)), and at an update size of 20% the speedup is 8.7x.

Accuracy: In Figure 11, 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 (Figure 10(c)).

Since the data cubing operation is primarily constructed by group-by aggregates, we can also measure the max error for each of the aggregates. We see that while the median staleness is close to 10%, for some queries some of the group aggregates have nearly 80% error (Figure 12). SVC greatly mitigates this error to less than 12% for all queries.

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 13). 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.

8.3.3 Complex Views

In the the last single-node experiment, we demonstrate the breadth of views supported by SVC. Each parameterized TPCD query is treated as a materialized view, and we use the `qgen` program to generate 10 instances for each query. When we report results, we average over these 10 instances

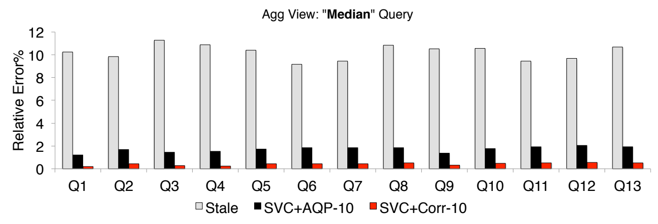


Figure 13: We run the same experiment but replace the `sum` query with a median query. We find that similarly SVC is more accurate.

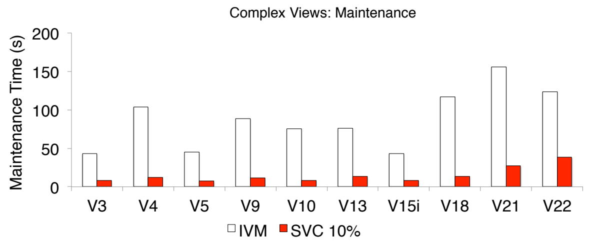


Figure 14: For 1GB update size, we compare maintenance time and accuracy of SVC with a 10% sample on different views. V21 and V22 do not benefit as much from SVC due to nested query structures.

for each of the query types.

Performance: Figure 15, 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 our sampling. 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.

8.3.4 Outlier Indexing

We hinted at the sensitivity to dataset skew of sampling-based approaches earlier. 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 TPCD query 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 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 16(a), we find in the most skewed

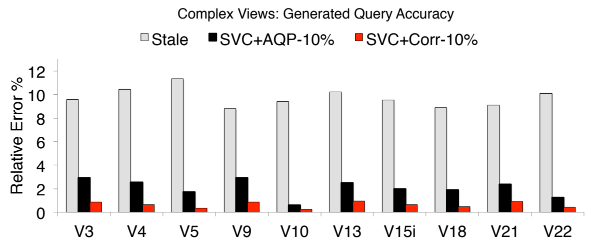


Figure 15: As before for a 10% sample size and 10% update size, SVC+Corr is more accurate than SVC+AQP and No Maintenance.

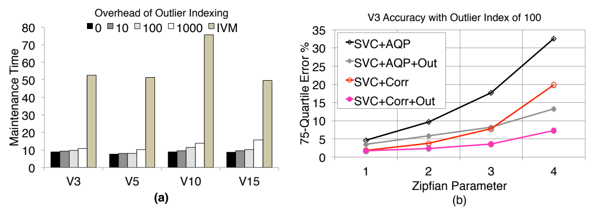


Figure 16: (a) For one view V3 and 1GB of updates, we plot the 75% quartile error with different techniques as we vary the skewness of the data. We find that SVC with an outlier index of size 100 is the most accurate. (b) While the outlier index adds an overhead this is small relative to the total maintenance time.

dataset SVC with outlier indexing reduces query error by a factor of 2. In the next Figure (Figure 16 (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.

8.4 Conviva

In our next set of experiments, we applied SVC to a 1TB dataset of logs from Conviva. We implement SVC on a 10-node Spark cluster.

8.4.1 *Performance and Accuracy*

We derive these views from 800GB of base data and add 80GB of updates. In Figure 17a, we show that while full maintenance takes nearly 800 seconds for one of the views, a 10% sample SVC can complete sample maintenance in less than 100s for all of them. On average SVC gives a 7.5x speedup.

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.

8.4.2 *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

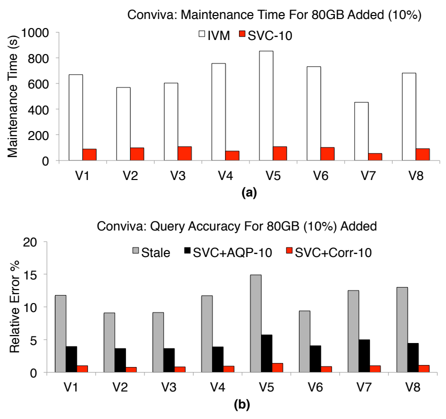


Figure 17: (a) We compare the maintenance time of SVC with a 10% sample and full incremental maintenance, and find that as with TPCD SVC saves significant maintenance time. (b) We also compare SVC+Corr to SVC+AQP and No Maintenance and find it is more accurate.

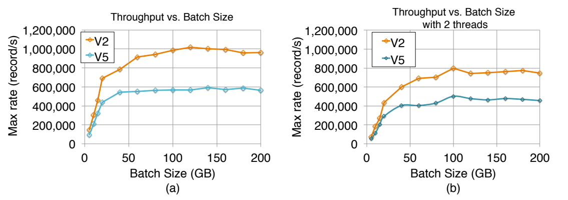


Figure 18: (a) Spark RDDs are most efficient when updated in batches. As batch sizes increase the system throughput increases. (b) When running multiple threads, the throughput reduces. However, larger batches are less affected by this reduction.

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 [57], 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 18(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

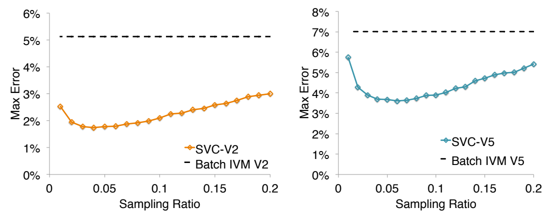


Figure 19: For a fixed throughput, SVC+Periodic Maintenance gives more accurate results for V2 and V5.

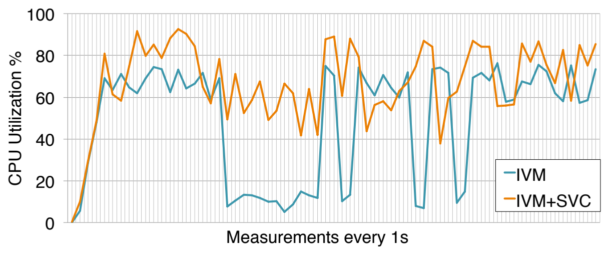


Figure 20: SVC better utilizes idle times in the cluster by maintaining the sample.

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 18(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 experiments (Figure 20), larger batch sizes are more amenable to parallel computation since there was more idle CPU time.

Choosing a Batch Size: The results in Figure 18(a) and Figure 18(b) show that larger batch sizes are more efficient, however, larger batch sizes also lead to more staleness. Combining the results in Figure 18(a) and Figure 18(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 19). 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 20, 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.

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 [39], transactions [5], and physical design [36]. The increased research focus parallels a major concern in industrial systems for incrementally updating pre-computed results and indices such as Google Percolator [44] and Twitter's Rainbird [55]. The streaming community has also studied the view maintenance problem [2,18,20,21,28,34]. In Spark Streaming, Zaharia et al. studied how they could exploit in-memory materialization [57], and in MonetDB, Liarou et al. studied how ideas from columnar storage can be applied to enable real-time analytics [37]. 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 [4,17,41]. Both the problems of efficiently sampling relations [41] 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 [14] and hashing. We further extended the work in [3,4] to consider how we can better leverage existing deterministic results 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 [50]. This work was applied in the context of concurrency control. However, this work did not consider applications when the correction was applied to a sample as in SVC. The sampling in SVC introduces new challenges such as sensitivity to outliers, questions of bias, and estimate optimality.

In the context of materialized view maintenance, sampling has primarily been primarily been studied from the perspective of maintaining samples [43]. Similarly, in [30], 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. [40]{{?}}, 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 [16,46,51]. 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. However, there are ideas from Rabkin et al. that could be applied in SVC in future work, for example, their description of coarsening operations in aggregates is very similar to our experiments with the "roll-up" queries in aggregate views. There are a variety of other efforts proposing storage efficient processing of aggregate queries on streams [15,23] which is similar to our problem setting and motivation.

Finally, the theory community has has studied related problems. Gibbons et al. studied the maintenance of approximate histograms [19], which closely resemble aggregate materialized views. This work did not model queries on the approximate histograms 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. There is also close relationship between sampling and probabilistic databases, and view maintenance and selection in the context of probabilistic databases have also been studied [48]. While this line work studies query processing on probabilistic views, it does not study their maintenance as in SVC.

10. CONCLUSION AND FUTURE WORK

In this paper, we propose a new framework, SVC, to address the staleness problem in materialized views. Since view maintenance is expensive, in practice, many prefer periodic maintenance solutions. In between these maintenance periods, materialized views become stale and can give incorrect query answers. SVC uses sampling to trade-off computation and accuracy 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 achieve this by modeling query answering on stale materialized views as a data cleaning problem, where stale rows are data error.

Our results suggest insights for both data cleaning and materialized views. We greatly expanded the scope of the SampleClean project [54] 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.

We evaluated our approach on real and synthetic datasets. We use the TPCD benchmark to illustrate three common use cases of materialized views: joins, data-cube aggregates, and nested queries. We also evaluated our approach on an industrial dataset from Conviva, where we demonstrated similar performance and accuracy. We simulated a real deployment where we integrate SVC with periodic maintenance, and showed that SVC can exploit idle times in the system to give more accurate query results.

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. We also believe there is a strong link between pre-computed machine learning models and materialized views, and the principles of our approach could be applied to build fast, approximate streaming machine learning applications. There is also a link to time series analysis and forecasting.

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

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

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

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

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

[18] 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.

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

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

[21] 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.

[22] 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.

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

[24] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.*, pages 328–339, 1995.

[25] T. Griffin, L. Libkin, and H. Trickey. An improved algorithm for the incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):508–511, 1997.

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

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

[28] 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.

[29] W. Hoeffding. A class of statistics with asymptotically normal distribution. *The Annals of Mathematical Statistics*, pages 293–325, 1948.

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

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

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

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

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

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

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

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

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

[45] L. L. Perez and C. M. Jermaine. History-aware query optimization with materialized intermediate views. In *ICDE*, pages 520–531, 2014.

[46] 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.

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

[48] C. Re and D. Suciu. Materialized views in probabilistic databases for information exchange and query optimization. In *VLDB*, pages 51–62, 2007.

[49] S. Samtani, V. Kumar, and M. Mohania. Self maintenance of multiple views in data warehousing. In *Proceedings of the eighth international conference on Information and knowledge management*, pages 292–299. ACM, 1999.

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

[51] 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.

[52] T. M. Truta and A. Campan. *K*-anonymization incremental maintenance and optimization techniques. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), Seoul, Korea, March 11-15, 2007*, pages 380–387, 2007.

[53] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[54] 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.

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

[56] 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.

[57] 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.

[58] 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.

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

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

12. APPENDIX

12.1 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.

12.2 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.

12.3 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