Stale View Cleaning: Getting Fresh Answers from Stale Materialized Views

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

ABSTRACT

Materialized views (MVs), stored pre-computed results, are widely used to facilitate fast queries on large datasets. When new records arrive at a high rate, it is infeasible to continuously update (maintain) MVs and a common solution is to defer maintenance by batching updates together. Between batches the MVs become increasingly stale with incorrect, missing, and superfluous rows leading to increasingly inaccurate query results. We propose Stale View Cleaning (SVC) which addresses this problem from a data cleaning perspective. In SVC, we efficiently clean a sample of rows from a stale MV, and use the clean sample to estimate aggregate query results. While approximate, the estimated query results reflect the most recent data. As sampling can be sensitive to long-tailed distributions, we further explore an outlier indexing technique to give increased accuracy when the data distributions are skewed. SVC complements existing deferred maintenance approaches by giving accurate and bounded query answers between maintenance. We evaluate our method on a generated dataset from the TPC-D benchmark and a real video distribution application. Experiments confirm our theoretical results: (1) cleaning an MV sample is more efficient than full view maintenance, (2) the estimated results are more accurate than using the stale MV, and (3) SVC is applicable for a wide variety of MVs.

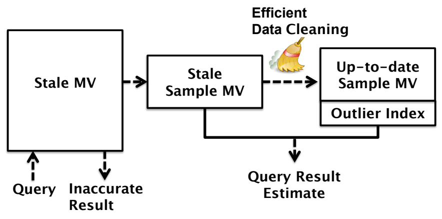

Figure 1: Deferred maintenance can lead to stale MVs which have incorrect, missing, and superfluous rows. In SVC, we pose this as a data cleaning problem and show that we can use a sample of clean (up-to-date) rows from an MV to correct inaccurate query results on stale views.

1. INTRODUCTION

Storing pre-computed query results, also known as materialization, is an extensively studied approach to reduce query latency on large data [9,19,28]. Materialized Views (MVs) are now supported by all major commercial vendors. However, as with any precomputation or caching, the key challenge in using MVs is maintaining their freshness as base data changes. While there has been substantial work in incremental maintenance of MVs [9,24], eager maintenance (i.e., immediately applying updates) is not always feasible.

In applications such as monitoring or visualization [32,43], analysts may create many MVs by slicing or aggregating over different dimensions. Eager maintenance requires updating all affected MVs for every incoming transaction, and thus, each additional MV reduces the available transaction throughput. This problem becomes significantly harder when the views are distributed and computational resources are contended by other tasks. As a result, in production environments, it is common to batch updates together to amortize overheads [9]. Batch sizes are set according to system constraints, and can vary from a few seconds to even nightly.

While increasing the batching period gives the user more flexibility to schedule around system constraints, a disadvantage is that MVs are stale between maintenance periods. Other than an educated guess based on past data, the user has no way of knowing how incorrect their query results are. Some types of views and query workloads can be sensitive to even a small number of base data updates, for example, if updates disproportionately affect a subset of frequently queried rows. Thus, any amount of staleness is potentially dangerous, and this presents us a dichotomy between facing the cost of eager maintenance or coping with consequences of unknown inaccuracy. In this paper, we explore an intriguing middle ground, namely, we can derive a bounded approximation of the correct answer for a fraction of the cost. With a small amount of up-to-date data, we can compensate for the error in aggregate query results induced by staleness.

Our method relies on modeling query answering on stale MVs as a data cleaning problem. A stale MV has incorrect, missing, or superfluous rows, which are problems that have been studied in the data cleaning literature (e.g., see Rahm and Do for a survey [40]). Increasing data volumes have led to development of new, efficient sampling-based approaches for coping with dirty data. In our prior work, we developed the SampleClean framework for scalable aggregate query processing on dirty data [42]. Since data cleaning is often expensive, we proposed cleaning a sample of data and using this sample to improve the results of aggregate queries on the full dataset. Since stale MVs are dirty data, an approach similar to SampleClean raises a new possibility of using a sample of "clean" rows in the MVs to return more accurate query results.

Stale View Cleaning (SVC illustrated in Figure 1) approximates aggregate query results from a stale MV and a small sample of up-

Proceedings of the VLDB Endowment, Vol. 8, No. 12
Copyright 2015 VLDB Endowment 2150-8097/15/08.

to-date data. We calculate a relational expression that materializes a uniform sample of up-to-date rows. This expression can be interpreted as "cleaning" a stale sample of rows. We use the clean sample of rows to estimate a result for an aggregate query on the view. The estimates from this procedure, while approximate, reflect the most recent data. Approximation error is more manageable than staleness because: (1) the uniformity of sampling allows us to apply theory from statistics such as the Central Limit Theorem to give tight bounds on approximate results, and (2) the approximate error is parametrized by the sample size which the user can control trading off accuracy for computation.

However, the MV setting presents new challenges that we did not consider in prior work. To summarize our contributions:

1. We propose a hashing-based technique that efficiently materializes an up-to-date sample view.
2. We process queries on this view by applying techniques proposed in SampleClean, but significantly extending them to a broader class of queries. Expanding the class of queries results in new challenges in bounding these estimates in confidence intervals. To address this challenge, we also derive a statistical bootstrap estimator to calculate bounds.
3. We apply an outlier indexing technique to reduce sensitivity to skewed datasets. We propose an index push-up algorithm that allows us to propagate the outliers to derived relations.
4. We evaluate this technique on real and synthetic datasets to show that SVC gives highly accurate results for a relatively small maintenance cost.

The paper is organized as follows: In Section 2, we give the necessary background for our work. Next, in Section 3, we formalize the problem. In Sections 4 and 5, we describe the sampling and query processing of our technique. In Section 6, we describe the outlier indexing framework. Then, in Section 7, we evaluate our approach. Finally, we discuss Related Work in Section 8. In Section 9, we discuss the limitations and future opportunities of our approach, and we present our Conclusions in Section 10.

2. BACKGROUND

2.1 Motivation and Example

Materialized view maintenance can be very expensive resulting in staleness. Many important use-cases require creating a large number of views including: visualization, personalization, privacy, and real-time monitoring. The problem with eager maintenance is that every view created by an analyst places a bottleneck on incoming transactions. There has been significant research on fast MV maintenance algorithms, most recently DBToaster [24] which uses SQL query compilation and higher-order maintenance. However, even with these optimizations, some materialized views are computationally difficult to incrementally maintain. For example, incremental maintenance of views with correlated subqueries can grow with the size of the data. Additionally, large views may require distribution and this further increases maintenance costs due to coordination. In real deployments, it is common to use the same infrastructure to maintain multiple MVs (along with other analytics tasks) adding further contention to computational resources and reducing overall available throughput. When faced with such challenges, it is common to batch updates to amortize maintenance overheads and add flexibility to scheduling.

Log Analysis Example: Suppose we are a video streaming company analyzing user engagement. Our database consists of two tables Log and Video, with the following schema:

```
Log(sessionId, videoId)
Video(videoId, ownerId, duration)
```

The Log table stores each visit to a specific video with primary key (`sessionId`) and a foreign-key to the Video table (`videoId`). For our analysis, we are interested in finding aggregate statistics on visits, such as the average visits per video and the total number of visits predicated on different subsets of owners. We could define the following MV that counts the visits for each `videoId` associated with owners and the duration:

```
CREATE VIEW visitView
AS SELECT videoId, ownerId, duration,
          count(1) as visitCount
FROM Log, Video
WHERE Log.videoId = Video.videoId
GROUP BY videoId
```

As Log table grows, this MV becomes stale, and we denote the insertions to the table as:

```
LogIns(sessionId, videoId)
```

Staleness does not affect every query uniformly. Even when the number of new entries in `LogIns` is small relative to `Log`, some queries might be very inaccurate. For example, views to newly added videos may account for most of `LogIns`, so queries that count visits to the most recent videos will be more inaccurate. The amount of inaccuracy is unknown to the user, who can only estimate an expected error based on prior experience. This assumption may not hold in rapidly evolving data. We see an opportunity for approximation through sampling which can give bounded query results for a reduced maintenance cost. In other words, a small amount of up-to-date data allows the user to estimate the magnitude of query result error due to staleness.

2.2 SampleClean [42]

To estimate up-to-date query results from stale materialized views, we leverage theory developed for query processing on dirty data. SampleClean is a framework for scalable aggregate query processing on dirty data. Traditionally, data cleaning has explored expensive, up-front cleaning of entire datasets for increased query accuracy. Those who were unwilling to pay the full cleaning cost avoided data cleaning altogether. We proposed SampleClean to add an additional trade-off to this design space by using sampling, i.e., bounded results for aggregate queries when only a sample of data is cleaned. The problem of high computational costs for accurate results mirrors the challenge faced in the MV setting with the tradeoff between immediate maintenance (expensive and up-to-date) and deferred maintenance (inexpensive and stale). Thus, we explore how samples of "clean" (up-to-date) data can be used for improved query processing on MVs without incurring the full cost of maintenance.

However, the metaphor of stale MVs as a Sample-and-Clean problem only goes so far and there are significant new challenges that we address in this paper. In prior work, we modeled data cleaning as a row-by-row black-box transformation. This model does not work for missing and superfluous rows in stale MVs. In particular, our sampling method has to account for this issue and we propose a hashing based technique to efficiently materialize a uniform sample even in the presence of missing/superfluous rows. Next, we greatly expand the query processing scope of SampleClean beyond `sum`, `count`, and `avg` queries. Bounding estimates that are not `sum`, `count`, and `avg` queries, is significantly more complicated. This requires new analytic tools such as a statistical bootstrap estimation to calculate confidence intervals. Finally, we add an outlier indexing technique to improve estimates on skewed data.

3. FRAMEWORK OVERVIEW

In this section, we formalize the two main problems that SVC addresses: (1) cleaning a stale sample MV and (2) answering an aggregate query with the clean sample.

3.1 Notation and Definitions

SVC returns a bounded approximation for aggregate queries on stale MVs for a flexible additional maintenance cost.

Materialized View: Let $\mathcal{D}$ be a database which is a collection of relations $\{R_i\}$. A *materialized view* S is the result of applying a *view definition* to $\mathcal{D}$. View definitions are composed of standard relational algebra expressions: Select (σ_ϕ), Project (Π), Join ($\bowtie$), Aggregation (γ), Union ($\cup$), Intersection ($\cap$) and Difference ($-$). We use the following parametrized notation for joins, aggregations and generalized projections:

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

The composition of the unary and binary relational expressions can be represented as a tree, which is called the *expression tree*. The leaves of the tree are the *base relations* for the view. Each non-leave node is the result of applying one of the above relational expressions to a relation. To avoid ambiguity, we refer to tuples of the base relations as *records* and tuples of derived relations as *rows*.

Primary Key: We assume that each of the base relations has a *primary key*. If this is not the case, we can always add an extra column that assigns an increasing sequence of integers to each record. For the defined relational expressions, every row in a materialized view can also be given a primary key [14,46], which we will describe in Section 4. This primary key is formally a subset of attributes $u \subseteq \{a_1, a_2, ..., a_k\}$ such that all $s \in S(u)$ are unique.

Staleness: For each relation R_i there is a set of insertions ΔR_i (modeled as a relation) and a set of deletions ∇R_i. An "update" to R_i can be modeled as a deletion and then an insertion. We refer to the set of insertion and deletion relations as "delta relations", denoted by $\partial\mathcal{D}$:

$$\partial\mathcal{D} = \{\Delta R_1, ..., \Delta R_k\} \cup \{\nabla R_1, ..., \nabla R_k\}$$

A view S is considered *stale* when there exist insertions or deletions to any of its base relations. This means that at least one of the delta relations in $\partial\mathcal{D}$ is non-empty.

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

$$S' = \mathcal{M}(S, \mathcal{D}, \partial D)$$

Staleness as Data Error: The consequences of staleness are incorrect, missing, and superfluous rows. Formally, for a stale view S with primary key u and an up-to-date view S':

- **Incorrect:** Incorrect rows are the set of rows (identified by the primary key) that are updated in S'. For $s \in S$, let $s(u)$ be the value of the primary key. An incorrect row is one such that there exists a $s' \in S'$ with $s'(u) = s(u)$ and $s \neq s'$.
- **Missing:** Missing rows are the set of rows (identified by the primary key) that exist in the up-to-date view but not in the stale view. For $s' \in S'$, let $s'(u)$ be the value of the primary key. A missing row is one such that there does not exist a $s \in S$ with $s(u) = s'(u)$.
- **Superfluous:** Superfluous rows are the set of rows (identified by the primary key) that exist in the stale view but not in the up-to-date view. For $s \in S$, let $s(u)$ be the value of the primary key. A superfluous row is one such that there does not exist a $s' \in S'$ with $s(u) = s'(u)$.

Uniform Random Sampling: We define a sampling ratio $m \in [0, 1]$ and for each row in a view S, we include it into a sample with probability m. We use the "hat" notation (e.g., $\widehat{S}$) to denote sampled relations. We say the relation $\widehat{S}$ is a *uniform sample* of S if

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

We say a sample is *clean* if and only if it is a uniform random sample of the up-to-date view S'.

EXAMPLE 1. *In this example, we summarize all of the key concepts and terminology pertaining to materialized views, stale data error, and maintenance strategies. Our example view, visitView, joins the Log table with the Video table and counts the visits for each video grouped by videoId. Since there is a foreign key relationship between the relations, this is just a visit count for each unique video with additional attributes. The primary keys of the base relations are: sessionId for Log and videoId for Video.*

If new records have been added to the Log table, the visitView is considered stale. Incorrect rows in the view are videos for which the visitCount is incorrect and missing rows are videos that had not yet been viewed once at the time of materialization. While not possible in our running example, superfluous rows would be videos whose Log records have all been deleted. Formally, in this example our database is $\mathcal{D} = \{Video, Log\}$, and the delta relations are $\partial\mathcal{D} = \{LogIns\}$.

Suppose, we apply the change-table IVM algorithm proposed in [19]:

1. *Create a "delta view" by applying the view definition to LogIns. That is, calculate the visit count per video on the new logs:*
$$\gamma(Video \bowtie LogIns)$$
2. *Take the full outer join of the "delta view" with the stale view visitView (equality on videoId).*
$$VisitView ⟗ \gamma(Video \bowtie LogIns)$$
3. *Apply the generalized projection operator to add the visitCount in the delta view to each of the rows in visitView where we treat a NULL value as 0:*
$$\Pi(VisitView ⟗ \gamma(Video \bowtie LogIns))$$

Therefore, the maintenance strategy is:

$$\mathcal{M}(\{VisitView\}, \{Video, Log\}, \{LogIns\}) = \Pi(VisitView ⟗ \gamma(Video \bowtie LogIns))$$

3.2 SVC Workflow

Formally, the workflow of SVC is:

1. We are given a view S.
2. $\mathcal{M}$ defines the maintenance strategy that updates S at each maintenance period.

3. The view S is stale between periodic maintenance, and the up-to-date view should be S'.
4. *(Problem 1. Stale Sample View Cleaning)* We find an expression $\mathcal{C}$ derived from $\mathcal{M}$ that cleans a uniform random sample of the stale view $\widehat{S}$ to produce a "clean" sample of the up-to-date view $\widehat{S'}$.
5. *(Problem 2. Query Result Estimation)* Given an aggregate query q and the state query result $q(S)$, we use $\widehat{S'}$ and $\widehat{S}$ to estimate the up-to-date result.
6. We optionally maintain an index of outliers o for improved estimation in skewed data.

Stale Sample View Cleaning: The first problem addressed in this paper is how to clean a sample of the stale materialized view.

PROBLEM 1 (STALE SAMPLE VIEW CLEANING). *We are given a stale view S, a sample of this stale view $\widehat{S}$ with ratio m, the maintenance strategy $\mathcal{M}$, the base relations $\mathcal{D}$, and the insertion and deletion relations $\partial\mathcal{D}$. We want to find a relational expression $\mathcal{C}$ such that:*

$$\widehat{S'} = \mathcal{C}(\widehat{S}, \mathcal{D}, \partial\mathcal{D}),$$

where $\widehat{S'}$ is a sample of the up-to-date view with ratio m.

Query Result Estimation: The second problem addressed in this paper is query result estimation.

PROBLEM 2 (QUERY RESULT ESTIMATION). *Let q be an aggregate query of the following form [1]:*

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

If the view S is stale, then the result will be incorrect by some value c:

$$q(S') = q(S) + c$$

Our objective is to find an estimator f such that:

$$q(S') \approx f(q(S), \widehat{S}, \widehat{S'})$$

EXAMPLE 2. *Suppose a user wants to know how many videos have received more than 100 views.*

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

Let us suppose the user runs the query and the result is 45. However, there have now been new records inserted into the Log table making this result stale. First, we take a sample of visitView and suppose this sample is a 5% sample. In Stale Sample View Cleaning (Problem 1), we apply updates, insertions, and deletions to the sample to efficiently materialize a 5% sample of the up-to-date view. In Query Result Estimation (Problem 2), we estimate aggregate query results based on the stale sample and the up-to-date sample.

4. EFFICIENTLY CLEANING A SAMPLE

In this section, we describe how to find a relational expression $\mathcal{C}$ derived from the maintenance strategy $\mathcal{M}$ that efficiently cleans a sample of a stale view $\widehat{S}$ to produce $\widehat{S'}$.

4.1 Challenges

To setup the problem, we first consider two naive solutions to this problem that will not work. We could trivially apply $\mathcal{M}$ to the entire stale view S and update it to S', and then sample. While the result is correct according to our problem formulation, it does not save us on any computation for maintenance. We want to avoid materialization of up-to-date rows outside of the sample. However, the naive alternative solution is also flawed. For example, we could just apply $\mathcal{M}$ to the stale sample $\widehat{S}$ and a sample of the delta relations $\widehat{\partial\mathcal{D}}$. The challenge is that $\mathcal{M}$ does not always commute with sampling.

4.2 Provenance

To understand the commutativity problem, consider the maintaining a group by aggregate:

```
SELECT videoId, count(1) FROM Log
GROUP BY videoId
```

The resulting view has one row for every distinct `videoId`. We want to materialize a sample of S', that is a sample of distinct `videoId`. If we sample the base relation `Log` first, we do not get a sample of the view. Instead, we get a view where every count is partial.

To achieve a sample of S', we need to ensure that for each $s \in S'$ all contributing rows in subexpressions to s are also sampled. This is a problem of row provenance [14]. Provenance, also termed lineage, has been an important tool in the analysis of materialized views [14] and in approximate query processing [46].

DEFINITION 1 (PROVENANCE). *Let r be a row in relation R, let R be derived from some other relation $R = exp(U)$ where $exp(\cdot)$ be a relational expression composed of the expressions defined in Section 3.1. The provenance of row r with respect to U is $p_U(r)$. This is defined as the set of rows in U such that for an update to any row $u \notin p_U(r)$, it guarantees that r is unchanged.*

4.3 Primary Keys

For the relational expressions defined in the previous sections, this provenance is well defined and can be tracked using primary key rules that are enforced on each subexpression [14]. We recursively define a set of primary keys for all relations in the expression tree:

DEFINITION 2 (PRIMARY KEY GENERATION). *For every relational expression R, we define the primary key attribute(s) of every expression to be:*

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

For every node at the expression tree, these keys are guaranteed to uniquely identify a row.

These rules define a constructive definition that can always be applied for our defined relational expressions.

EXAMPLE 3. *A variant of our running example view that does not have a primary key is:*

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

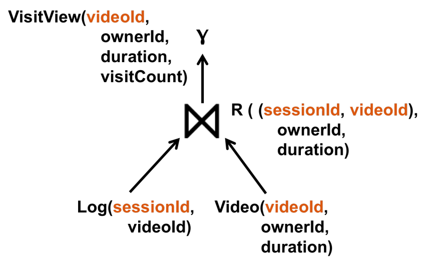

Figure 2: Applying the rules described in Definition 2, we illustrate how to assign a primary key to a view.

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

We illustrate the key generation process in Figure 2. Suppose there is a base relation, such as Log, that is missing a primary key (sessionId)[2]*. We can add this attribute by generating an increasing sequence of integers for each record in Log. Since both base tables Video and Log have primary keys videoId and sessionId respectively, the result of the join will have a primary key (videoId, sessionId). Since the group by attribute is videoId, that becomes the primary key of the view.*

4.4 Hashing Operator

The primary keys allow us to determine the set of rows that contribute to a row r in a derived relation. If we have a deterministic way of mapping a primary key to a Boolean, we can ensure that all contributing rows are also sampled. To achieve this we use a hashing procedure. Let us denote the hashing operator $\eta_{a,m}(R)$. For all tuples in R, this operator applies a hash function whose range is $[0,1]$ to primary key a (which may be a set) and selects those records with hash less than or equal to m [3].

In this work, we study uniform hashing where the condition $h(a) \leq m$ implies that a fraction of approximately m of the rows are sampled. Such hash functions are utilized in other aspects of database research and practice (e.g. hash partitioning, hashed joins, and hash tables). Hash functions in these applications are designed to be as uniform as possible to avoid collisions. There are numerous empirical studies that establish that for many commonly applied hash functions (e.g., Linear, SDBM, MD5, SHA) the difference with a true uniform random variable is very small [22,29]. Cryptographic hashes work particularly well and are supported by most commercial and open source systems, for example MySQL provides MD5 and SHA1/2.

To avoid materializing extra rows, we push down the hashing operator through the expression tree. The further that we can push η down, the more operators (i.e., above the sampling) can benefit. This push down is analogous to predicate push-down operations used in query optimizers. In particular, we are interested in finding an optimized relational expression that materializes an identical sample before and after the push down. We formalize the push down rules below:

Definition 3 (Hash push down). *For a derived relation R, the following rules can be applied to push $\eta_{a,m}(R)$ down the expression tree.*

- $\sigma_\phi(R)$: *Push η through the expression.*
- $\Pi_{(a_1,\ldots,a_k)}(R)$: *Push η through if a is in the projection.*
- $\bowtie_{\phi(r1,r2)}(R_1,R_2)$: *No push down in general. There are special cases below where push down is possible.*
- $\gamma_{f,A}(R)$: *Push η through if a is in the group by clause A.*
- $R_1 \cup R_2$: *Push η through to both R_1 and R_2*
- $R_1 \cap R_2$: *Push η through to both R_1 and R_2*
- $R_1 - R_2$: *Push η through to both R_1 and R_2*

[2]It does not make sense for Video to be missing a primary key in our running example due to the foreign key relationship

[3]For example, if hash function is a 32-bit unsigned integer which we can normalize by `MAXINT` to be in $[0,1]$.

Special Case of Joins: In general, a join $R \bowtie S$ blocks the pushdown of the hash function $\eta_{a,m}(R)$ since a possibly consists of attributes in both R and S. However, when there is a constraint that enforces these attributes are equal then pushdown is possible.

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

Equality Join. If the join is an equality join and a is one of the attributes in the equality join condition $R_1.a = R_2.b$, then η can be pushed down to both R_1 and R_2. On R_1 the pushed down operator is $\eta_{a,m}(R_1)$ and on R_2 the operator is $\eta_{b,m}(R_2)$.

Example 4. *We illustrate our hashing procedure in terms of SQL expressions on our running example. We can pushdown the hash function for the following expressions:*

```
SELECT * FROM Video WHERE Predicate()
SELECT * FROM Video,Log WHERE Video.videoId = Log.videoId
SELECT videoId, count(1) FROM Log GROUP BY videoId
```

The following expressions are examples where we cannot push-down the hash function:

```
SELECT * FROM Video, Log

SELECT c, count(1)
FROM (
        SELECT videoId, count(1) as c FROM Log
        GROUP BY videoId
     )
GROUP BY c
```

Theorem 1. *Given a derived relation R, primary key a, and the sample $\eta_{a,m}(R)$. Let S be the sample created by applying $\eta_{a,m}$ without push down and S' be the sample created by applying the push down rules to $\eta_{a,m}(R)$. S and S' are identical samples with sampling ratio m.*

Proof Sketch. We can prove this by induction. The base case is where the expression tree is only one node, trivially making this true. Then, we can induct considering one level of operators in the tree. $\sigma, \cup, \cap, -$ clearly commute with hashing the key a allowing for push down. Π commutes only if a is in the projection. For $\bowtie$, a sampling operator on Q can be pushed down if a is in either k_r or k_s, or if there is a constraint that links k_r to k_s. For group by aggregates, if a is in the group clause (i.e., it is in the aggregate) then a hash of the operand filters all rows that have a which is sufficient to materialize the derived row. □

4.5 Efficient View Cleaning

If we apply the hashing operator to $\mathcal{M}$, we can get an optimized cleaning expression $\mathcal{C}$ that avoid materializing unnecessary rows. When applied to a stale sample of a view $\widehat{S}$, the database $\mathcal{D}$, and the delta relations $\partial\mathcal{D}$, it produces an up-to-date sample with sampling ratio m:

$$\widehat{S}' = \mathcal{C}(\widehat{S}, \mathcal{D}, \partial\mathcal{D})$$

Thus, it addresses Problem 1 from the previous section.

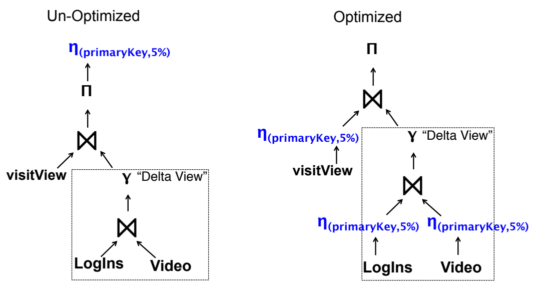

Figure 3: Applying the rules described in Section 4.4, we illustrate how to optimize the sampling of our example maintenance strategy.

EXAMPLE 5. *We illustrate our proposed approach on our example view* `visitView` *(Figure 3). The primary key for the view is the tuple (*`videoId`*) making that the primary key of the MV. We start by applying the hashing operator to this key. The next operator we see in the expression tree is a projection that increments the* `visitCount` *in the view, and this allows for push down since primary key is in the projection. The second expression is a hash of the equality join key which merges the aggregate from the "delta view" to the old view allowing us to push down on both branches of the tree using our special case for equality joins. On the left side, we reach the stale view so we stop. On the right side, we reach the aggregate query (count) and since the primary key is in group by clause, we can push down the sampling. Then, we reach another point where we hash the equality join key allowing us to push down the sampling to the relations LogIns and Video.*

4.6 Corresponding Samples

We started with a uniform random sample $\widehat{S}$ of the stale view S. The hash push down allows us to efficiently materialize the sample $\widehat{S}'$. $\widehat{S}'$ is a uniform random sample of the up-to-date view S. While both of these samples are uniform random samples of their respective relations, the two samples are correlated since $\widehat{S}'$ is generated by cleaning $\widehat{S}$. In particular, our hashing technique ensures that the primary keys in $\widehat{S}'$ depend on the primary keys in $\widehat{S}$. Statistically, this positively correlates the query result $q(\widehat{S}')$ and $q(\widehat{S})$. We will see how this property can be leveraged to improve query estimation accuracy (Section 5.1).

PROPERTY 1 (CORRESPONDENCE). *Suppose $\widehat{S}'$ and $\widehat{S}$ are uniform samples of S' and S, respectively. Let u denote the primary key. We say $\widehat{S}'$ and $\widehat{S}$ correspond if and only if:*

- *Uniformity: $\widehat{S}'$ and $\widehat{S}$ are uniform random samples of S' and S respectively with a sampling ratio of m*
- *Removal of Superfluous Rows: $D = \{\forall s \in \widehat{S} \nexists s' \in S' : s(u) = s'(u)\}$, $D \cap \widehat{S}' = \emptyset$*
- *Sampling of Missing Rows: $I = \{\forall s' \in \widehat{S}' \nexists s \in S : s(u) = s'(u)\}$, $\mathbb{E}(\mid I \cap \widehat{S}' \mid) = m \mid I \mid$*
- *Key Preservation for Updated Rows: For all $s \in \widehat{S}$ and not in D or I, $s' \in \widehat{S}' : s'(u) = s(u)$.*

5. QUERY RESULT ESTIMATION

SVC returns two corresponding samples, $\widehat{S}$ and $\widehat{S}'$. $\widehat{S}$ is a "dirty" sample (sample of the stale view) and $\widehat{S}'$ is a "clean" sample (sample of the up-to-date view). In this section, we first discuss how to estimate query results using the two corresponding samples. Then, we discuss the bounds and guarantees on different classes of aggregate queries.

5.1 Result Estimation

Suppose, we have an aggregate query q of the following form:

```
q(View) := SELECT f(attr) FROM View WHERE cond(*)
```

We quantify the staleness c of the aggregate query result as the difference between the query applied to the stale view S compared to the up-to-date view S':

$$q(S') = q(S) + c$$

The objective of this work is to estimate $q(S')$. In the Approximate Query Processing (AQP) literature, sample-based estimates have been well studied [4,38]. This inspires our first estimation algorithm, SVC+AQP, which uses SVC to materialize a sample view and an AQP-style result estimation technique.

SVC+AQP: Given a clean sample view $\widehat{S}'$, the query q, and a scaling factor s, we apply the query to the sample and scale it by s:

$$q(S') \approx s \cdot q(\widehat{S}')$$

For example, for the `sum` and `count` the scaling factor is $\frac{1}{m}$. For the `avg` the scaling factor is 1. Refer to [4,38] for a detailed discussion on the scaling factors.

SVC+AQP returns what we call a direct estimate of $q(S')$. We could, however, try to estimate c instead. Since we have the stale view S, we could run the query q on the full stale view and estimate the difference c using the samples $\widehat{S}$ and $\widehat{S}'$. We call this approach SVC+CORR, which represents calculating a correction to $q(S)$ instead of a direct estimate.

SVC+CORR: Given a clean sample $\widehat{S}'$, its corresponding dirty sample $\widehat{S}$, a query q, and a scaling factor s:

1. Apply SVC+AQP to $\widehat{S}'$: $r_{est\_fresh} = s \cdot q(\widehat{S}')$
2. Apply SVC+AQP to $\widehat{S}$: $r_{est\_stale} = s \cdot q(\widehat{S})$
3. Apply q to the full stale view: $r_{stale} = q(S)$
4. Take the difference between (1) and (2) and add it to (3):

$$q(S') \approx r_{stale} + (r_{est\_fresh} - r_{est\_stale})$$

A commonly studied property in the AQP literature is unbiasedness. An unbiased result estimate means that the expected value of the estimate over all samples of the same size is $q(S')$ [4]. We can prove that if SVC+AQP is unbiased (there is an AQP method that gives an unbiased result) then SVC+CORR also gives unbiased results.

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 e_q of q. Then, it follows that $\mathbb{E}\big[e_q(\widehat{S}')\big] = q(S')$ If we substitute in this expression: $c = \mathbb{E}\big[e_q(\widehat{S}')\big] - q(S)$. Applying the linearity of expectation: $c = \mathbb{E}\big[e_q(\widehat{S}') - q(S)\big]$ □

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

EXAMPLE 6. *We can formalize our earlier example query in Section 2 in terms of SVC+CORR and SVC+AQP. 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, and suppose we are working with a sampling ratio of 5%. For SVC+AQP, we*

[4]The `avg` query is considered conditionally unbiased in some works, however, this difference is largely notational and does not affect any subsequent bounds.

SQL Query	Family	Unbiased	Variance
`avg`, `sum`, `count`	Mean	Yes	Optimal
`std`, `var`	Variance	Yes	Optimal
`median`, `percentile`	Ranking	Bounded	Suboptimal
`max`, `min`	Extrema	Unbounded	Suboptimal

Table 1: SQL queries and the properties of their statistical estimation family.

count the number of videos in the clean sample that currently have counts greater than 100 and scale that result by $\frac{1}{5\%} = 20$*. If the count from the clean sample is* 4*, then the estimate for SVC+AQP is* 80*. For SVC+CORR, we also run SVC+AQP on the dirty sample. Suppose that there are only two videos in the dirty sample with counts above 100, then the result of running SVC+AQP on the dirty sample is* $20 \cdot 2 = 40$*. We take the difference of the two values* $80 - 40 = 40$*. This means that we should correct the old result by* 40 *resulting in the estimate of* $45 + 40 = 85$.

5.2 Estimate Accuracy

To analyze the estimate accuracy, we taxonomize common SQL aggregate queries into different *estimator families*. For example, `sum`, `count`, and `avg` can all be written as sample means. `sum` is the sample mean scaled by the relation size and `count` is the mean of the indicator function scaled by the relation size. There are some key properties of interest within different estimator families: unbiasedness, existence analtyic confidence intervals, and optimality. SVC+AQP and SVC+CORR inherit the properties of the estimator family.

Table 1 describes these families and their properties for common queries. The sample mean family of estimators (`sum`, `count`, and `avg`) has analytic solutions and has been the focus of other approximate query processing works [38,42], and we analyze this family in detail. The general case can only be bounded empirically which is more challenging.

5.2.1 Confidence Intervals For Sample Means

Now we will discuss bounding our estimates in confidence intervals for `sum`, `count`, and `avg`, which can be estimated with "sample mean" estimators. Sample means for uniform random samples (also called sampling without replacement) converge to the population mean by the Central Limit Theorem (CLT). Let $\bar{\mu}$ be a sample mean calculated from k samples, σ^2 be the variance of the sample, and μ be the population mean. Then, the error $(\mu - \bar{\mu})$ is normally distributed: $N(0, \frac{\sigma^2}{k})$. Therefore, the confidence interval is given by:

$$\bar{\mu} \pm \gamma\sqrt{\frac{\sigma^2}{k}}$$

where γ is the Gaussian tail probability value (e.g., 1.96 for 95%, 2.57 for 99%).

We discuss how to calculate this confidence interval in SQL for SVC+AQP. The first step is a query rewriting step where we move the predicate cond(*) into the SELECT clause (1 if true, 0 if false). Let *attr* be the aggregate attribute and m be the sampling ratio. We define an intermediate result $trans$ which is a table of transformed rows with the first column the primary key and the second column defined in terms of `cond(*)` statement and scaling. For `sum`:

```
trans= SELECT pk,1.0/m·attr·cond(*) as trans_attr FROM s
```

For `count`:

```
trans= SELECT pk, 1.0/m · cond(*) as trans_attr FROM s
```

For `avg` since there is no scaling we do not need to re-write the query:

```
trans= SELECT pk, attr as trans_attr FROM s WHERE cond(*)
```

SVC+AQP: The confidence interval on this result is defined as:

```
SELECT γ·stdev(trans_attr)/sqrt(count(1)) FROM trans
```

To calculate the confidence intervals for SVC+CORR we have to look at the statistics of the difference, i.e., $c = q(S) - q(S')$, from a sample. If all rows in $\widehat{S}$ exist in $\widehat{S}'$, we could use the associativity of addition and subtraction to rewrite this as: $c = q(S - S')$, where $-$ is the row-by-row difference between S and S'. The challenge is that the missing rows on either side make this ill-defined. Thus, we have define the following null-handling semantics with a subtraction operator we call $\dot{-}$.

DEFINITION 4 (CORRESPONDENCE SUBTRACT). *Given an aggregate query, and two corresponding relations* R_1 *and* R_2 *with the schema* $(a_1, a_2, ...)$ *where* a_1 *is the primary key for* R_1 *and* R_2*, and* a_2 *is the aggregation attribute for the query.* $\dot{-}$ *is defined as a projection of the full outer join on equality of* $R_1.a_1 = R_2.a_1$*:*

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

Null values $\emptyset$ *are represented as zero.*

Using this operator, we can define a new intermediate result $diff$:

$$diff := trans(\widehat{S}')\dot{-}trans(\widehat{S})$$

SVC+CORR: Then, as in SVC+AQP, we bound the result using the CLT:

```
SELECT γ·stdev(trans_attr)/sqrt(count(1)) FROM diff
```

5.2.2 AQP vs. CORR For Sample Means

In terms of these bounds, we can analyze how SVC+AQP compares to SVC+CORR for a fixed sample size k. SVC+AQP gives an estimate that is proportional to the variance of the clean sample view: $\frac{\sigma^2_{S'}}{k}$. SVC+CORR to the variance of the *differences*: $\frac{\sigma^2_c}{k}$. Since the change is the difference between the stale and up-to-date view, this can be rewritten as

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

Therefore, a correction will have less variance when:

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

As we saw in the previous section, correspondence correlates the samples. If the difference is small, i.e., S is nearly identical to S', then $cov(S, S') \approx \sigma^2_S$. This result also shows that there is a point when updates to the stale MV are significant enough that direct estimates are more accurate. When we cross the break-even point we can switch from using SVC+CORR to SVC+AQP. SVC+AQP does not depend on $cov(S, S')$ which is a measure of how much the data has changed. Thus, we guarantee an approximation error of at most $\frac{\sigma^2_{S'}}{k}$. In our experiments (Figure 6(b)), we evaluate this break even point empirically.

5.2.3 Selectivity For Sample Means

Let p be the selectivity of the query and k be the sample size; that is, a fraction p records from the relation satisfy the predicate. For these queries, we can model selectivity as a reduction of effective sample size $k \cdot p$ making the estimate variance: $O(\frac{1}{k*p})$. Thus, the confidence interval's size is scaled up by $\frac{1}{\sqrt{p}}$. Just like there is a tradeoff between accuracy and maintenance cost, for a fixed accuracy, there is also a tradeoff between answering more selective queries and maintenance cost.

5.2.4 Optimality For Sample Means

Optimality in unbiased estimation theory is defined in terms of the variance of the estimate [13].

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

A sampled relation R defines a discrete distribution. It is important to note that this distribution is different from the data generating distribution, since even if R has continuous valued attributes R still defines a discrete distribution. Our population is finite and we take a finite sample thus every sample takes on only a discrete set of values. In the general case, this distribution is only described by the set of all of its values (i.e., no smaller parametrized representation). In this setting, the sample mean is an MVUE. In other words, if we make no assumptions about the underlying distribution of values in R, SVC+AQP and SVC+CORR are optimal for their respective estimates ($q(S')$ and c). Since they estimate different variables, even with optimality SVC+CORR might be more accurate than SVC+AQP and vice versa.

There are, however, some cases when the assumptions of this optimality do not hold. The intuitive problem is that if there are a small number of parameters that completely describe the discrete distribution there might be a way to reconstruct the distribution from those parameters rather than estimating the mean. As a simple counter example, if we knew our data were exactly on a line, a sample size of two is sufficient to answer any aggregate query. However, even for many parametric distributions, the sample mean estimators are still MVUEs, e.g., poisson, bernouilli, binomial, normal, and exponential. It is often difficult and unknown in many cases to derive an MVUE other than a sample mean. Furthermore, the sample mean is unbiased for any distribution, but it is often the case that alternative MVUEs are biased when the data is not exactly from the correct model family (such as our example of the line). Our approach is valid for any choice of estimator if one exists, even though we do the analysis for sample mean estimators and this is the setting in which that estimator is optimal.

5.2.5 General Estimators

The theory for bounding general estimators outside of the sample mean family is more complex. We may not get analytic confidence intervals on our results, nor is it guaranteed that our estimates are optimal. In AQP, the commonly used technique is called a statistical bootstrap [4] to empirically bound the results. In this approach, we repeatedly subsample with replacement from our sample and apply the query to the sample. This gives us a technique to bound SVC+AQP the details of which can be found in [3,4,46]. For SVC+CORR, we have to propose a variant of bootstrap to bound the estimate of c. In this variant, repeatedly estimate c from subsamples and build an empirical distribution for c.

SVC+CORR: To use bootstrap to find a 95% confidence interval:

1. Subsample $\widehat{S}'_{sub}$ and $\widehat{S}_{sub}$ with replacement from $\widehat{S}'$ and $\widehat{S}$ respectively
2. Apply SVC+AQP to $\widehat{S}'_{sub}$ and $\widehat{S}_{sub}$
3. Record the difference $s \cdot (q(\widehat{S}'_{sub}) - q(\widehat{S}_{sub}))$, note that for some queries such as `median` $s = 1$.
4. Return to 1, for k iterations.
5. Return the 97.5% and the 2.5% percentile of the distribution of results.

6. OUTLIER INDEXING

Sampling is known to be sensitive to outliers [7,10]. Power-laws and other long-tailed distributions are common in practice [10]. We address this problem using a technique called outlier indexing which has been applied in AQP [7]. The basic idea is that we create an index of outlier records (records whose attributes deviate from the mean value greatly) and ensure that these records are included in the sample, since these records greatly increase the variance of the data. However, as this has not been explored in the materialized view setting there are new challenges in using this index for improved result accuracy.

6.1 Indices on the Base Relations

In [7], the authors applied outlier indexing to improve the accuracy of AQP on base relations. In our problem, we issue queries to materialized views. We need to define how to propagate information from an outlier index on the base relation to a materialized view.

The first step is that the user selects an attribute of any base relation to index and specifies a threshold t and a size limit k. In a single pass of updates (without maintaining the view), the index is built storing references to the records with attributes greater than t. If the size limit is reached, the incoming record is compared to the smallest indexed record and if it is greater then we evict the smallest record. The same approach can be extended to attributes that have tails in both directions by making the threshold t a range, which takes the highest and the lowest values. However, in this section, we present the technique as a threshold for clarity.

There are many approaches to select a threshold. 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, for a given attribute we can select the the top-k records with that attributes. Then, we can use that top-k list to set a threshold for our index. Then, the attribute value of the lowest record becomes the threshold t. Alternatively, we can calculate the variance of the attribute and set the threshold to represent c standard deviations above the mean.

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

6.2 Adding Outliers to the Sample

We need to propagate the indices upwards through the expression tree. We add the condition that the only eligible indices are ones on base relations that are being sampled (i.e., we can push the hash operator down to that relation). Therefore, in the same iteration as sampling, we can also test the index threshold and add records to the outlier index. We formalize the propagation property recursively. Every relation can have an outlier index which is a set of attributes and a set of records that exceed the threshold value on those attributes. The main idea is to treat the indexed records as a sub-relation that gets propagated upwards with the maintenance strategy.

DEFINITION 5 (OUTLIER INDEX PUSHUP). *Define an outlier index to be a tuple of a set of indexed attributes, and a set of records (I, O). The outlier index propagates upwards with the following rules:*

- *Base Relations: Outlier indices on base relations are pushed up only if that relation is being sampled, i.e., if the sampling operator can be pushed down to that relation.*
- $\sigma_\phi(R)$*: Push up with a new outlier index and apply the selection to the outliers* $(I, \sigma_\phi(O))$

- $\Pi_{(a_1,...,a_k)}(R)$*: Push upwards with new outlier index* $(I \cap (a_1, ..., a_k), O)$.
- $\bowtie_{\phi(r1,r2)} (R_1, R_2)$*: Push upwards with new outlier index* $(I_1 \cup I_2, O_1 \bowtie O_2)$.
- $\gamma_{f,A}(R)$*: For group-by aggregates, we set* I *to be the aggregation attribute. For the outlier index, we do the following steps. (1) Apply the aggregation to the outlier index* $\gamma_{f,A}(O)$*, (2) for all distinct* A *in* O *select the row in* $\gamma_{f,A}(R)$ *with the same* A*, and (3) this selection is the new set of outliers* O*.*
- $R_1 \cup R_2$*: Push up with a new outlier index* $(I_1 \cap I_2, O_1 \cup O_2)$*. The set of index attributes is combined with an intersection to avoid missed outliers.*
- $R_1 \cap R_2$*: Push up with a new outlier index* $(I_1 \cap I_2, O_1 \cap O_2)$.
- $R_1 - R_2$*: Push up with a new outlier index* $(I_1 \cup I_2, O_1 - O_2)$.

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

6.3 Query Processing

For result estimation, we can think of our sample $\hat{S}'$ and our outlier index O as two distinct parts. Since $O \subset S'$, and we give membership in our outlier index precedence, our sample is actually a sample restricted to the set $(\widehat{S' - O})$. The outlier index has two uses: (1) we can query all the rows that correspond to outlier rows, and (2) we can improve the accuracy of our *aggregation* queries. To query the outlier rows, we can select all of the rows in the materialized view that are flagged as outliers, and these rows are guaranteed to be up-to-date.

For (2), we can also incorporate the outliers into our correction estimates. For a given query, let c_{reg} be the correction calculated on $(\widehat{S' - O})$ using the technique proposed in the previous section and adjusting the sampling ratio m to account for outliers removed from the sample. We can also apply the technique to the outlier set O since this set is deterministic the sampling ratio for this set is $m = 1$, and we call this result c_{out}. Let N be the count of records that satisfy the query's condition and l be the number of outliers that satisfy the condition. Then, we can merge these two corrections in the following way: $v = \frac{N-l}{N} c_{reg} + \frac{l}{N} c_{out}$. For the queries in the previous section that are unbiased, this approach preserves unbiasedness. Since we are averaging two unbiased estimates c_{reg} and c_{out}, the linearity of the expectation operator preserves this property. Furthermore, since c_{out} is deterministic (and in fact its bias/variance is 0), c_{reg} and c_{out} are uncorrelated making the bounds described in the previous section applicable as well.

EXAMPLE 7. *Suppose, we want to use outlier indexing to process the query in the previous section on* visitView*. We chose an attribute in the base data to index, for example* `duration`*, and an example threshold of 1.5 hours. We first push the index through the join of* Log *and* Video*. Then, we reach the group by aggregate, where we select all the distinct groups (videos) for which the duration is longer than 1.5 hours. This materializes the entire set of rows whose duration is longer than 1.5 hours. For SVC+AQP, we run the query on the set of clean rows with durations longer than 1.5 hours. Then, we use the update rule in Section 6.3 to update the result based on the number of records in the index and the total size of the view. For SVC+CORR, we additionally run the query on the set of dirty rows with durations longer than 1.5 hours and take the difference between SVC+AQP. As in SVC+AQP, we use the update rule in Section 6.3 to update the result based on the number of records in the index and the total size of the view.*

7. RESULTS

We evaluate SVC first on a single node MySQL database to evaluate its accuracy, performance, and efficiency in a variety of materialized view scenarios. 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 application of server log analysis with a dataset from a video streaming company, Conviva.

7.1 Experimental Setup

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 [8] 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 [34] is a long-tailed distribution where 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. The incremental maintenance algorithm used in our experiments is the "change-table" or "delta-table" method used in numerous works in incremental maintenance [19,20,24]. 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[5]:

Join View: In the TPCD specification, two tables receive insertions and updates: lineitem and orders. Out of 22 parametrized 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.

Complex Views: Our goal is to demonstrate the applicability of SVC outside of simple materialized views that include nested queries and other more complex relational algebra. We take the TPCD schema and denormalize the database, and treat each of the 22 TPCD queries as views on this denormalized schema. The 22 TPCD queries are actually parametrized 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 each random instance as a materialized view. 10 out of the 22 sets of views can benefit from SVC. For the 12 excluded views, 3 were static (i.e, this means that there are no updates to the view based on the TPCD workload), and the remaining 9 views have a small cardinality not making them suitable for sampling.

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.

[5]Refer to our extended paper on more details about the experimental setup [26].

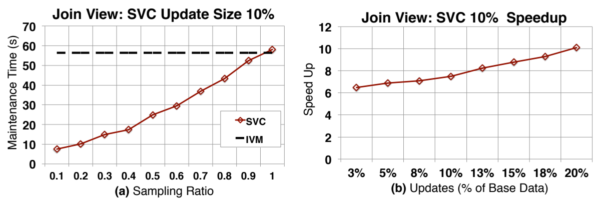

Figure 4: (a) On a 10GB view with 1GB of insertions and updates, we vary the sampling ratio and measure the maintenance time of SVC. (b) For a fixed sampling ratio of 10%, we vary the update size and plot the speedup compared to full incremental maintenance.

For each attribute a, the domain is specified in the TPCD standard. We select a random subset of this domain, e.g., if the attribute is country then the predicate can be countryCode > 50 and countryCode < 100. We generated 100 random `sum`, `avg`, and `count` queries for each view.

Distributed Experimental Setup: We evaluate SVC on Apache Spark 1.1.0 with 1TB of logs from a video streaming company, Conviva [1]. This is a denormalized user activity log corresponding to video views and various metrics such as data transfer rates, and latencies. Accompanying this data is a four month trace of analyst queries in SQL. We identified 8 common summary statistics-type queries that calculated engagement and error-diagnosis metrics. These 8 queries defined the views in our experiments. We populated these view definitions using the first 800GB of user activity log records. We then applied the remaining 200GB of user activity log records as the updates (i.e., in the order they arrived) in our experiments. We generated aggregate random queries over this view by taking either random time ranges or random subsets of customers.

7.1.1 Metrics and Evaluation

We will illustrate that SVC is more accurate than the stale query result (No Maintenance); but is less computationally intensive than full IVM. 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 (change-table based maintenance [19,20,24]) to the full view.

SVC+AQP: We maintain a sample of the materialized view using SVC and estimate the result with AQP-style estimation technique.

SVC+CORR: We maintain a sample of the materialized view using SVC and process queries on the view using the correction which applies the AQP to both the clean and dirty samples, and uses both estimates to correct a stale query result.

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

Relative Error: For a query result r and an incorrect result r', the relative error is $\frac{|r-r'|}{r}$. When a query has multiple results (a group-by query), then, unless otherwise noted, relative error is defined as the median over all the errors.

Maintenance Time: We define the maintenance time as the time needed to produce the up-to-date view for incremental view maintenance, and the time needed to produce the up-to-date sample in SVC.

7.2 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 $z = 2$, and derive the view

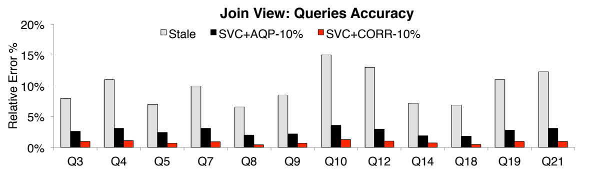

Figure 5: For a fixed sampling ratio of 10% and update size of 10% (1GB), we generate 100 of each TPCD parameterized queries and answer the queries using the stale materialized view, SVC+CORR, and SVC+AQP. We plot the median relative error for each query.

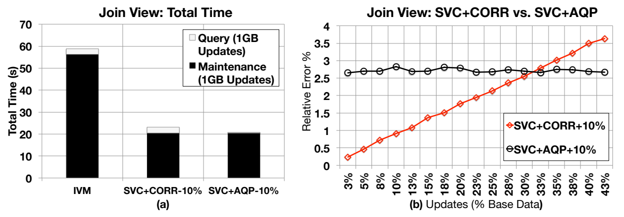

Figure 6: (a) For a fixed sampling ratio of 10% and update size of 10% (1GB), we measure the total time incremental maintenance + query time. (b) SVC+CORR is more accurate than SVC+AQP until a break even point.

from this dataset. We first generate 1GB (10% of the base data) of updates (insertions and updates to existing records), and vary the sample size.

Performance: Figure 4(a) shows the maintenance time of SVC as a function of sample size. With the bolded dashed line, we note the time for full IVM. For this materialized view, sampling allows for significant savings in maintenance time; albeit for approximate answers. While full incremental maintenance takes 56 seconds, SVC with a 10% sample can complete in 7.5 seconds.

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

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

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

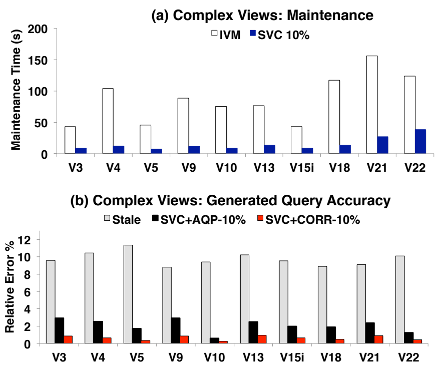

Figure 7: (a) For 1GB update size, we compare maintenance time and accuracy of SVC with a 10% sample on different views. V21 and V22 do not benefit as much from SVC due to nested query structures. (b) For a 10% sample size and 10% update size, SVC+CORR is more accurate than SVC+AQP and No Maintenance.

overhead to the savings in maintenance time it is small.

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

7.3 Complex Views

In this experiment, we demonstrate the breadth of views supported by SVC by using the TPCD queries as materialized views. We generate a 10GB base TPCD dataset with skew $z = 2$, and derive the views 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 7 shows the maintenance time for a 10% sample compared to the full view. This experiment illustrates how the view definitions plays a role in the efficiency of our approach. For the last two views, V21 and V22, we see that sampling does not lead to as large of speedup indicated in our previous experiments. This is because both of those views contain nested structures which block the pushdown of hashing. V21 contains a subquery in its predicate that does not involve the primary key, but still requires a scan of the base relation to evaluate. V22 contains a string transformation of a key blocking the push down. These results are consistent with our previous experiments showing that SVC is faster than IVM and more accurate than SVC+AQP and no maintenance.

7.4 Outlier Indexing

In our next experiment, we evaluate our outlier indexing with the top-k strategy described in Section 6. In this setting, outlier indexing significantly helps for both SVC+AQP and SVC+CORR. We index the l_extendedprice attribute in the lineitem table. We evaluate the outlier index on the complex TPCD views. We find that four views: V3, V5, V10, V15, can benefit from this index with our push-up rules. These are four views dependent on l_extendedprice that were also in the set of "Complex" views chosen before.

In our first outlier indexing experiment (Figure 8(a)), we analyze V3. We set an index of 100 records, and applied SVC+CORR and SVC+AQP to views derived from a dataset with a skew parameter $z = \{1, 2, 3, 4\}$. We run the same queries as before, but this time we measure the error at the 75% quartile. We find in the most skewed data SVC with outlier indexing reduces query error by a factor of 2. Next, in (Figure 8 (b)), we plot the overhead for outlier indexing for V3 with an index size of 0, 10, 100, and 1000. While there is an overhead, it is still small compared to the gains made by sampling the maintenance strategy. We note that none of the prior experiments used an outlier index. The caveat is that these experiments were done with moderately skewed data with Zipfian parameter = 2, if this parameter is set to 4 then the 75% quartile query estimation error is nearly 20% (Figure 8a). Outlier indexing always improves query results as we are reducing the variance of the estimation set, however, this reduction in variance is largest when there is a longer tail.

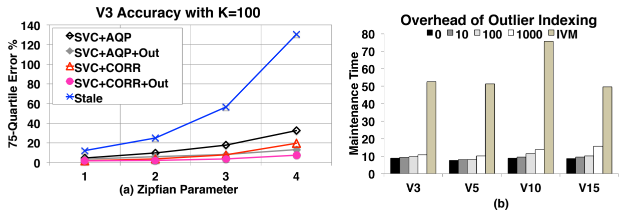

Figure 8: (a) For one view V3 and 1GB of updates, we plot the 75% quartile error with different techniques as we vary the skewness of the data. (b) While the outlier index adds an overhead this is small relative to the total maintenance time.

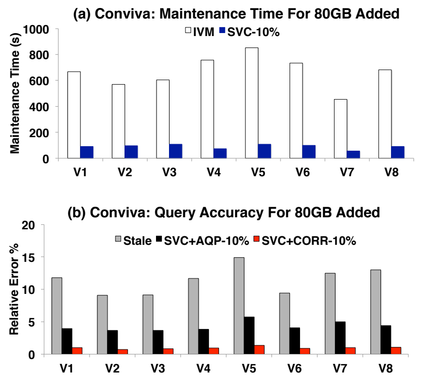

Figure 9: (a) We compare the maintenance time of SVC with a 10% sample and full incremental maintenance, and find that as with TPCD SVC saves significant maintenance time. (b) We also evaluate the accuracy of the estimation techniques.

7.5 Conviva

We derive the views from 800GB of base data and add 80GB of updates, these views are stored and maintained using Apache Spark in a distributed environment. The goal of this experiment is to evaluate how SVC performs in a real world scenario with a real dataset and a distributed architecture. In Figure 9(a), we show that on average over all the views, SVC-10% gives a 7.5x speedup. For one of the views full incremental maintenance takes nearly 800 seconds, even on a 10-node cluster, which is a very significant cost. In Figure 9(b), we show that SVC also gives highly accurate results with an average error of 0.98%. These results show consistency with our results on the synthetic datasets. This experiment highlights a few salient benefits of SVC: (1) sampling is a relatively cheap operation and the relative speedups in a single node and distributed environment are similar, (2) for analytic workloads like Conviva (i.e., user engagement analysis) a 10% sample gives results with 99% accuracy, and (3) the cost of incremental view maintenance is very significant systems like Spark for large views.

8. 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 [35], transactions [5], real-time analytics [31], and physical design [30]. The streaming community has also studied the view maintenance problem [2,16,18,21,25]. SVC proposes an alternative model where we allow approximation error (with guarantees) for queries on materialized views for vastly reduced maintenance time.

Sampling has been well studied in the context of query processing [4,15,37]. Both the problems of efficiently sampling relations [37] 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 look the problem of approximate query processing [3,4] from a different perspective by estimating a "correction" rather than estimating query results. Srinivasan and Carey studied a problem related to query correction which they called compensation-based query processing [41] for concurrency control but did not study this for sampled estimates.

Sampling has also been studied from the perspective of maintaining samples [39]. In [23], Joshi and Jermaine studied indexed materialized views that are amenable to random sampling. While similar in spirit (queries on the view are approximate), the goal of this work was to optimize query processing not address the cost of incremental maintenance. There has been work using sampled views in a limited context of cardinality estimation [27], which is the special case of our framework, namely, the `count` query. Nirkhiwale et al. [36], studied an algebra for estimating confidence intervals in aggregate queries. The objective of this work is not sampling efficiency, as in SVC, but estimation. As a special case, where we consider only views constructed from select and project operators, SVC's hash pushdown will yield the same results as their model. There has been theoretical work on the maintenance of approximate histograms, synopses, and sketches [12,17], which closely resemble aggregate materialized views. The objectives of this work (including techniques such as sketching and approximate counting) has been to reduce the required storage, not to reduce the required update time.

Meliou et al. [33] proposed a technique to trace errors in an MV to base data and find responsible erroneous tuples. They do not, however, propose a technique to correct the errors as in SVC. Correcting general errors as in Meliou et al. is a hard constraint satisfaction problem. However, in SVC, through our formalization of staleness, we have a model of how updates to the base data (modeled as errors) affect MVs, which allows us to both trace errors and clean them. Wu and Madden [44] did propose a model to correct "outliers" in an MV through deletion of records in the base data. This is a more restricted model of data cleaning than SVC, where the authors only consider changes to existing rows in an MV (no insertion or deletion) and does not handle the same generality of relational expressions (e.g., nested aggregates). Challamalla et al. [6] proposed an approximate technique for specifying errors as constraints on a materialized view and proposing changes to the base data such that these constraints can be satisfied. While complementary, one major difference between the three works [6,33,44] and SVC is that they require an explicit specification of erroneous rows in a materialized view. Identifying whether a row is erroneous requires materialization and thus specifying the errors is equivalent to full incremental maintenance. We use the formalism of a "maintenance strategy", the relational expression that updates the view, to allow us to sample rows that are not yet materialized. However, while not directly applicable for staleness, we see SVC as complementary to these works in the dirty data setting. The sampling technique proposed in Section 4 of our paper could be used to approximate the data cleaning techniques in [6,33,44] and this is an exciting avenue of future work.

9. LIMITATIONS AND OPPORTUNITIES

While our experiments show that SVC works for a variety of applications, there are a few limitations which we summarize in this section. There are two primary limitations for SVC: class of queries and types of materialized views. In this work, we primarily focused on aggregate queries and showed that accuracy decreases as the selectivity of the query increases. Sampled-based methods are fundamentally limited in the way they can support "point lookup" queries that select a single row. This is predicted by our theoretical result that accuracy decreases with $\frac{1}{p}$ where p is the fraction of rows that satisfy the predicate. In terms of more view definitions, SVC does not support views with ordering or "top-k" clauses, as our sampling assumes no ordering on the rows of the MV and it is not clear how sampling commutes with general ordering operations. In the future, we will explore maintenance optimizations proposed in recent work. For example, DBToaster has two main components, higher-order delta processing and a SQL query compiler, both of which are complementary to SVC.

10. CONCLUSION

Materialized view maintenance is often expensive, and in practice, eager view maintenance is often avoided due to its costs. This leads to stale materialized views which have incorrect, missing, and superfluous rows. In this work, we formalize the problem of staleness and view maintenance as a data cleaning problem. SVC uses a sample-based data cleaning approach to get accurate query results that reflect the most recent data for a greatly reduced computational cost. To achieve this, we significantly extended our prior work in data cleaning, SampleClean [42], for efficient cleaning of stale MVs. This included processing a wider set of aggregate queries, handling missing data errors, and proving for which queries optimality of the estimates hold. We presented both empirical and theoretical results showing that our sample data cleaning approach is significantly less expensive than full view maintenance for a large class of materialized views, while still providing accurate aggregate query answers that reflect the most recent data.

This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, C3Energy, Cisco, Cray, Cloudera, EMC2, Ericsson, Facebook, Guavus, HP, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Schlumberger, Splunk, Virdata and VMware.

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] A. Chalamalla, I. F. Ilyas, M. Ouzzani, and P. Papotti. Descriptive and prescriptive data cleaning. In *SIGMOD Conference*, pages 445–456, 2014.

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

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

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

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms.

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

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

[16] 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.

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

[18] 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.

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

[20] H. Gupta and I. S. Mumick. Incremental maintenance of aggregate and outerjoin expressions. *Information Systems*, 31(6):435–464, 2006.

[21] 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.

[22] C. Henke, C. Schmoll, and T. Zseby. Empirical evaluation of hash functions for packetid generation in sampled multipoint measurements. In *Passive and Active Network Measurement*, pages 197–206. Springer, 2009.

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

[24] 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.

[25] 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.

[26] S. Krishnan, J. Wang, M. J. Franklin, K. Goldberg, and T. Kraska. Stale view cleaning: Getting fresh answers from stale materialized views. `http://www.ocf.berkeley.edu/~sanjayk/pubs/svc-2014.pdf`, 2014.

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

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

[29] P. L'Ecuyer and R. Simard. Testu01: Ac library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS)*, 33(4):22, 2007.

[30] 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.

[31] 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.

[32] Z. Liu, B. Jiang, and J. Heer. *imMens*: Real-time visual querying of big data. *Comput. Graph. Forum*, 32(3):421–430, 2013.

[33] A. Meliou, W. Gatterbauer, S. Nath, and D. Suciu. Tracing data errors with view-conditioned causality. In *SIGMOD Conference*, pages 505–516, 2011.

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

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

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

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

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

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

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

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

[42] 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.

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

[44] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 6(8):553–564, 2013.

[45] 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.

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

12. APPENDIX

12.1 Extensions

12.1.1 MIN and MAX

`min` and `max` fall into their own category since this is a canonical case where bootstrap fails. We devise an estimation procedure that corrects these queries. However, we can only achieve bound that has a slightly different interpretation than the confidence intervals seen before. We can calculate the probability that a larger (or smaller) element exists in the unsampled view.

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

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

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

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

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

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

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

12.1.2 Select Queries

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

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

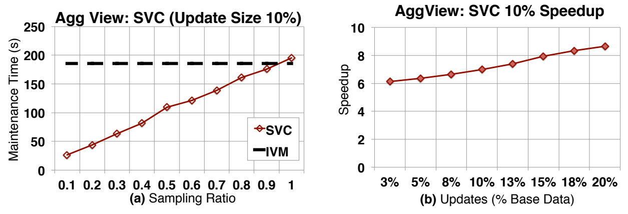

Figure 10: (a) In the aggregate view case, sampling can save significant maintenance time. (b) As the update size grows SVC tends towards an ideal speedup of 10x.

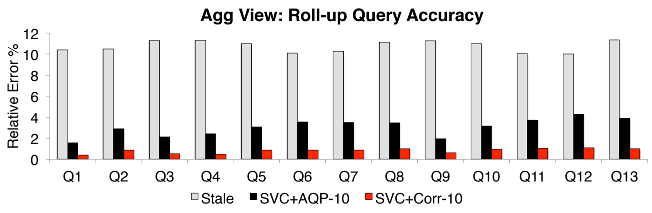

Figure 11: We measure the accuracy of each of the roll-up aggregate queries on this view. For a 10% sample size and 10% update size, we find that SVC+Corr is more accurate than SVC+AQP and No Maintenance.

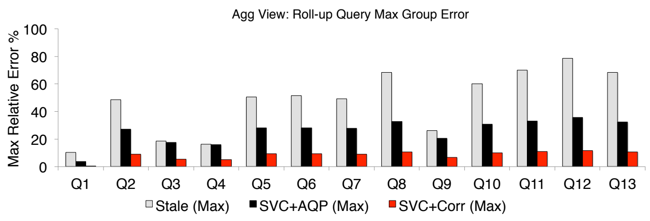

Figure 12: For 1GB of updates, we plot the max error as opposed to the median error in the previous experiments. Even though updates are 10% of the dataset size, some queries are nearly 80% incorrect. SVC helps significantly mitigate this error.

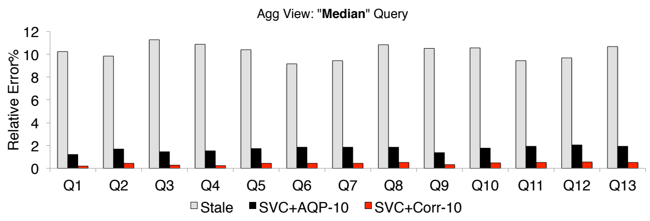

Figure 13: We run the same experiment but replace the `sum` query with a median query. We find that similarly SVC is more accurate.

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

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

12.2 Additional Experiments

12.2.1 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 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.

12.2.2 Mini-batch Experiments

We devised an end-to-end experiment simulating a real integration with periodic maintenance. However, unlike the MySQL case, Apache Spark does not support selective updates and insertions as the "views" are immutable. A further point is that the immutability of these views and Spark's fault-tolerance requires that the "views" are maintained synchronously. Thus, to avoid these significant overheads, we have to update these views in batches. Spark does have a streaming variant [45], 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.

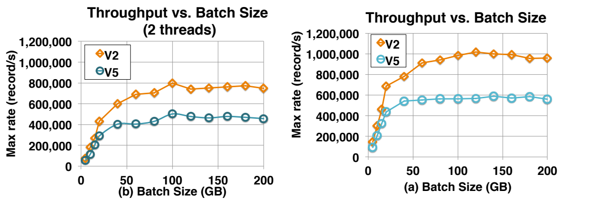

Figure 14: (a) Spark RDDs are most efficient when updated in batches. As batch sizes increase the system throughput increases. (b) When running multiple threads, the throughput reduces. However, larger batches are less affected by this reduction.

Batch sizes: In Spark, larger batch sizes amortize overheads better. In Figure 14(a), we show a trade-off between batch size and throughput of Spark for V2 and V5. Throughputs for small batches are nearly 10x smaller than the throughputs for the larger batches.

Concurrent SVC and IVM: Next, we measure the reduction in throughput when running multiple threads. We run SVC-10 in loop in one thread and IVM in another. We measure the reduction in throughput for the cluster from the previous batch size experiment. In Figure 14(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 16), larger batch sizes are more amenable to parallel computation since there was more idle CPU time.

Choosing a Batch Size: The results in Figure 14(a) and Figure 14(b) show that larger batch sizes are more efficient, however, larger batch sizes also lead to more staleness. Combining the results in Figure 14(a) and Figure 14(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 15). 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 16, 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.

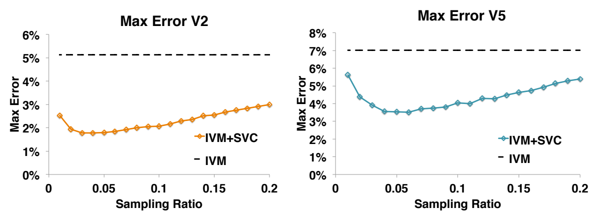

Figure 15: For a fixed throughput, SVC+Periodic Maintenance gives more accurate results for V2 and V5.

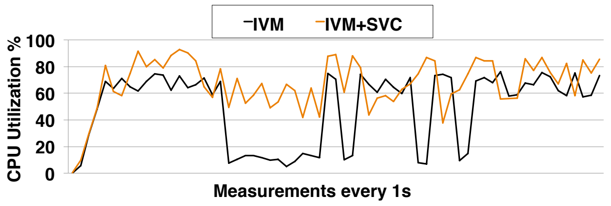

Figure 16: SVC better utilizes idle times in the cluster by maintaining the sample.

12.3 Extended Proofs

12.4 Is Hashing Equivalent To RNG?

In this work, we argue that hashing can be used for "sampling" a relational expression. However, from a complexity theory perspective, hashing is not equivalent to random number generation (RNG). The existence of true one-way hash functions is a conjecture that would imply $P \neq NP$. This conjecture is often taken as an assumption in Cryptography. Of course, the ideal one-way hash functions required by the theory do not exist in practice. However, we find that existing hashes (e.g., linear hashes and SHA1) are sufficiently close to ideal that they can still take advantage of this theory. On the other hand, a SHA1 hash is nearly an order of magnitude slower but is much more uniform. This assumption is called the Simple Uniform Hashing Assumption (SUHA) [11], and is widely used to analyze the performance of hash tables and hash partitioning. There is an interesting tradeoff between the latency in computing a hash compared to its uniformity. For example, a linear hash stored procedure in MySQL is nearly as fast pseudorandom number generation that would be used in a TABLESAMPLE operator, however this hash exhibits some non-uniformity.

12.4.1 Hashing and Correspondence

A benefit of deterministic hashing is that when applied in conjunction to the primary keys of a view, we get the Correspondence Property (Definition 1) for free.

PROPOSITION 2 (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. There are four conditions for correspondence:

- (1) Uniformity: $\widehat{S}'$ and $\widehat{S}$ are uniform random samples of S' and S respectively with a sampling ratio of m
- (2) Removal of Superfluous Rows: $D = \{\forall s \in \widehat{S} \nexists s' \in S' : s(u) = s'(u)\}, D \cap \widehat{S}' = \emptyset$
- (3) Sampling of Missing Rows: $I = \{\forall s' \in \widehat{S}' \nexists s \in S : s(u) = s'(u)\}, \mathbb{E}(\mid I \cap \widehat{S}' \mid) = m \mid I \mid$
- (4) Key Preservation for Updated Rows: For all $s \in \widehat{S}$ and not in D or I, $s' \in \widehat{S}' : s'(u) = s(u)$.

Uniformity is satisfied under by definition under SUHA (Simple Uniform Hashing Assumption). Condition 2 is satisfied since if r is deleted, then $r \notin S'$ which implies that $r \notin \hat{S}'$. Condition 3 is just the converse of 2 so it is satisfied. Condition 4 is 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 $\hat{S}'$. □

12.5 Theorem 1 Proof

THEOREM 2. *Given a derived relation R, primary key a, and the sample $\eta_{a,m}(R)$. Let S be the sample created by applying $\eta_{a,m}$ without push down and S' be the sample created by applying the push down rules to $\eta_{a,m}(R)$. S and S' are identical samples with sampling ratio m.*

PROOF. We can prove this by induction. The base case is where the expression tree is only one node, trivially making this true. Then, we can induct considering one level of operators in the tree. $\sigma, \cup, \cap, -$ clearly commute with hashing the key a allowing for push down. Π commutes only if a is in the projection. For $\bowtie$, a sampling operator on Q can be pushed down if a is in either k_r or k_s, or if there is a constraint that links k_r to k_s. There are two cases in which this happens a foreign-key relationship or an equality join on the same key. For group by aggregates, if a is in the group clause (i.e., it is in the aggregate) then a hash of the operand filters all rows that have a which is sufficient to materialize the derived row. It is provably NP-Hard to pushdown through a nested group by aggregate such as:

```
SELECT c, count(1)
FROM (
        SELECT videoId, sum(1) as c FROM Log
        GROUP BY videoId
     )
GROUP BY c
```

by reduction to a SUBSET-SUM problem. □

12.6 More about the 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.

12.7 Experimental Details

12.7.1 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.

12.7.2 Conviva Views

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.

12.7.3 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.

12.7.4 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