A Data Cleaning Approach for Approximately Up-To-Date Results on Stale Materialized Views

ABSTRACT

TODO

1. INTRODUCTION

Database systems increasingly use materialization, caching a computed query result, to speed up queries on large datasets [?]. However, as the base tables update, materialized views derived from these tables become increasingly stale. Incrementally keeping the views up-to-date, also called incremental maintenance, has been well studied [?] including a variety of techniques such as batch maintenance [?] and lazy maintenance [?].

Unfortunately, in many desired applications, incremental view maintenance can be very costly. This cost breaks down into two components: applying the view definition to the updates, and then writing the "delta" view to the out-of-date view. These two pieces can be costly in different applications. (1) In distributed environments where the view is partitioned over a cluster, incremental view maintenance often neccesitates communicating the delta view. (2) Systems such as Apache Spark, Cloudera Impala, and Apache Tez [?] offer materialized view support, however, are not optimized for selective updates nor have native support for indices. This can lead to high maintenance costs in applications where the views are derived from joins that are not aligned with the partitioning of the base tables. (3) Base data is often raw requiring pre-processing such as string processing, deserialization, and formatting; all of which can can be expensive to run on a large number of updates. Consequently, a commonly applied approach is to extend the maintenance period and schedule maintenance at less active times eg. nightly; while accepting that in the interim results will be stale. This approach avoids placing an undue bottleneck on updates to the base tables and reduces contention on hot data; however a user querying the system can get results that are arbitrarily stale.

Querying a stale view is similar to problems studied in data cleaning[?]. SampleClean is a query processing framework that answers aggregate queries on dirty datasets by applying potentially expensive cleaning techniques to just a sample. The results, while approximate, are bounded with respect to the clean data and the system offers a flexible tradeoff between cleaning cost and result accuracy. Similarly, a stale row and an expensive incremental maintenance scheme, mirrors the problem setting studied in SampleClean.

In this paper, we propose a data cleaning approach for approximate, bounded aggregation queries on stale views. Instead of maintaining the entire view, we maintain only a small sample of the view. Then given an aggregation query on this view, from this small sample, we can estimate how the updates affect the query result. We apply this estimate to correct the dirty aggregation query result on the stale data. Our results are provably bounded, as opposed to unbounded stalness, and the sampling gives a flexible tradeoff to meet performance constraints. Sampling helps reduces both bottlenecks in view maintenance, delta view calculation and view updating, as it reduces the number of updates that need to processed and then written.

Another relevant concept from data cleaning is outlier detection [?]. Sampling has the potential to mask outliers and in fact it is known heavy-tailed distributions are poorly approximated from samples [?]. Recent work has shown that outlier indexing, ie. separating the values from the tail, can improve sample estimates in such distributions. Coupling outlier detection with sampled views has an interesting implication; not only are the outliers themselves interesting for analysis, but the information from the outliers can potentially improve query accuracy or likewise reduce the number of needed samples.

In summary, our contributions are as follows:

- We present a query processing framework that gives bounded error for aggregation queries on stale views, while providing a tradeoff between error and performance.
- We show how coupling concepts from data cleaning such as correction estimates and outlier detection can give increased accuracy (and performance).
- We evaluate our approach on two systems, Apache SparkSQL and MySQL, and discuss how the different systems affect performance performance parameters.

2. BACKGROUND

Materialized views are stored query results that are used to optimize query processing [?]. Due to the decreasing cost of memory, in-memory materialization has had much interest in recent research [?] and materialization research has expanded beyond the SQL setting [?].

However, pre-computed query results face the obvious challenge of *staleness* when applied in a setting where the underlying tables are updating. One commonly applied solution is to recompute the materialized views when the table has been updated. This can be very expensive in the presence of small updates that hardly change the derived views. Consequently, incremental maintenance of materialized views is well studied see [?] for a survey of the approaches. A simple model of incremental maintenance consists of two steps: calculating a "delta" view, and "refreshing" the materialized view with the delta. More formally, given a base relation T, a set of updates U, and a view $\mathbf{V}_T$:

Calculate the Delta View- In this step, we apply the view definition to the updates and we call the intermediate result a "delta" view.

$$\Delta\mathbf{V} = \mathbf{V}_U$$

This is also called a *change propagation formula* in some literature, especially on algebraic representations of incremental view maintenance.

Refresh View- Given the "delta" view, we merge the results with the existing view:

$$\mathbf{V}_T^{'} = refresh(\mathbf{V}_T, \Delta\mathbf{V})$$

The details of the refresh operation depend on the view definition. Refer to [?] for details.

In this work, we address three types of materialized views: Select-Project, Aggregation, and Foreign-Key Join views; and four common aggregation queries on these views: SUM, COUNT, AVG, and VAR. We further analyze only insertions into the database and defer analysis of updates and deletions for future work.

2.1 Scheduling Maintenance

While often less expensive than recomputing a materialized view, incremental maintenance can still be computationally expensive. Materialized views are growing larger and are more frequently implemented in distributed systems. Due to this cost, which we will refer to as the "maintenance" cost, scheduling the refresh operation has been an important topic of research.

There are two principle types of scheduling strategies: immediate and deferred. In immediate maintenance, as soon as a record is updated, the change is propagated to any derived materialized view. Immediate maintenance has an advantage that materialized view is always up-to-date, however it can be very expensive. This scheduling strategy places a bottleneck when records are written reducing the available write-throughput for the database. Furthermore, especially in a distributed setting, record-by-record maintenance cannot take advantage of the benefits of consolidating overheads by batching. To address these challenges, deferred maintenance is alternative solution. In deferred maintenance, the user often accepts some degree of staleness in the materialized view for additional flexibility for scheduling refresh operations. For example, a user can update the materialized view nightly during times of low-activity in the system. More sophisticated deferred scheduling schemes are also possible, refer to [?] for a full survey.

2.2 A Data Cleaning Approach

Without consistency guarantees, in many deferred incremental maintenance approaches, queries on the materialized view are stale until the refresh. In fact, in general, the relative error between the stale result and the up-to-date result is unbounded. But the tradeoff is that deferred maintainence allows flexibility in scheduling to meet the performance constraints of the system and workload.

To similar effect, sampling has been applied to scale expensive query processing operations in streaming [?], aggregate query processing [?], and data cleaning [?]. In particular, the problem of staleness closely resembles a data cleaning problem. There are rows in materialized view that are incorrect, and there is a procedure to "clean" these rows. In the SampleClean project [?], the authors suggested that to process aggregate queries on dirty datasets, one need not clean the entire dataset but rather a random sample to get an approximate answer. Rather than an uncleaned result that can be arbitrarily dirty, you acheive an approximate result with bounded sampling error.

One of key algorithms in SampleClean, NormalizedSC, takes a dirty dataset and query and using a sample of clean data learns how to correct the query so the result is "clean". This approach gives highly accurate answers in datasets where errors are sparse since the correction will be small. In context of materialized views, we explore whether we can apply a similar algorithm to acheive approximately up-to-date results; where we take a sample of up-to-date data and estimate how much we have to correct a stale query result.

However, NormalizedSC alone is not suited for "staleness" data errors. First, the algorithm is designed for sampling base relations and not views. Without views, NormalizedSC does not address the challenge of correcting nested queries so unmodified it would not work. Provided with a way to uniformly sample the view from the base relation, NormalizedSC still has limitations. The SampleClean model looks at errors that can corrected with a transformation of records. Corrections to stale materialized views can imply insertions of new rows in addition to transformations of existing rows.

In this work, we show how ideas from NormalizedSC can be extended to the setting of materialized views. And the implication is that we can use sampling to infer up-to-date results for aggregation queries on stale materialized views. We can apply incremental maintenance to just a sample of the view, thus greatly reducing the cost needed to maintain the view (Figure ?). With the sampling ratio, the user has a knob to tradeoff performance (in terms of throughput and refresh latency) and accuracy.

We envision that this work is complementary to incremental maintenance. In the periods between full materialized view refreshes, the user maintains a sample of the view. The user specifies a sampling ratio that meets the systems performance constraints. With this, the user gets guarantees that for the aggregate queries the results have bounded error and the queries return with error bars.

3. SYSTEM ARCHITECTURE

The architecture of our proposed solution is shown in Figure 3. In this section, we will give an overview of each of the blocks of the system.

To illustrate our approach, we use the following running example which is a simplified schema of one of our experi-

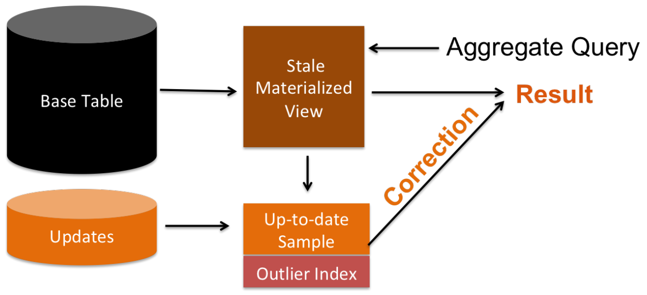
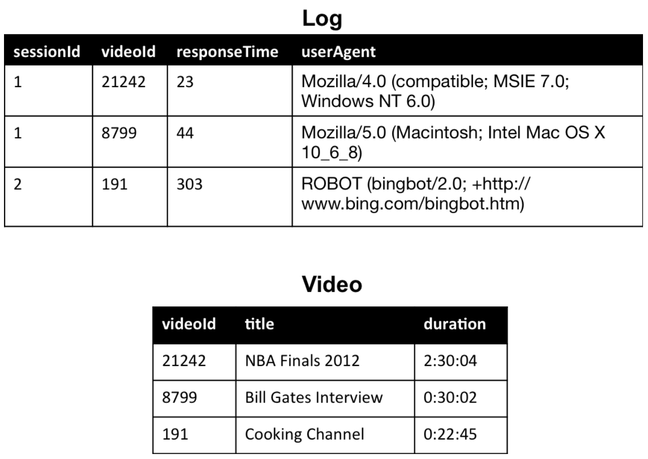


Figure 1: TODO

mental datasets (Figure 3). Imagine, we are querying logs from a video streaming company. These logs record visits from users as they happen and grow over time. We have two table: Log and Video, with the following schema:

```
Log(sessionID, videoID, responseTime, userAgent)
Video(videoID, title, duration)
```

These tables are related with a foreign-key relationship between Log and Video, and there is an integrity constraint that every log record must link to one video in the video table.



Log

sessionId	videoId	responseTime	userAgent
1	21242	23	Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0)
1	8799	44	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_6_8)
2	191	303	ROBOT (bingbot/2.0; +http:// www.bing.com/bingbot.htm)

Video

videoId	title	duration
21242	NBA Finals 2012	2:30:04
8799	Bill Gates Interview	0:30:02
191	Cooking Channel	0:22:45

Figure 2: TODO

3.1 Supported Materialized Views

We will first introduce the taxonomy of materialized views that can benefit from our approach. In particular, we provide example situations when view maintenance can be costly.

Select-Project Views

One type of view that we consider are views generated from Select-Project expressions of the following form:

```
SELECT [col1,col2,...]
FROM table
WHERE condition([col1,col2,...])
```

There are situations when such views are expensive to maintain. For example, as often the case with activity logs, the base table may contain semi-structured data that requires parsing or preprocessing as a part of the view definition. Consider the following example, a typical column in online activity logs is the User-Agent String (Figure ?). When a user accesses a webpage the browser reports this string to identify the browser type, operating system, and layout engine. Suppose, we wanted to create the following view:

```
SELECT * FROM Log
WHERE userAgent
LIKE '%Mozilla%'
```

This involves evaluating regular expression on the string to see if it matches a criteria. Testing a complex regular expression will be far more expensive than numerical comparisions or equality testing. In more extreme examples, the columns may be serialized objects (eg. represented in JSON) which need to be deserialized before evaluating a predicate.

Aggregation Views

We also consider aggregation views of the following form:

```
SELECT [f1(col1),f2(col2),...]
FROM table
WHERE condition([col1,col2,...])
GROUP BY [col1,col2,...]
```

While the same costs that Select-Project views can incur due to pre-process apply as well, aggregation views pose additional challenges to incremental view maintenance. Aggregation views can be costly to maintain when the cardinality of the result is large; that is when there group by clause is very selective. Consider the following example:

```
SELECT videoID,
max(responseTime) AS maxResponseTime
FROM Log
GROUP BY videoID;
```

The cardinality of the delta view is the total number of videos in the log which can be very large. Thus it will be costly to propagate this result with the existing view, potentially updating a large number of rows. These costs increase in a distributed environment where a larger delta view means that more data has to be communicated through a shuffle operation.

Foreign-Key Join Views

The third type of view we consider are Foreign-Key Join views:

```
SELECT table1.[col1,col2,...],
 table2.[col1,col2,...]
FROM table1, table2
WHERE table1.fk = table2.fk
AND condition([col1,col2,...])
```

Such views are ubquitious in star schemas [?] and can be particularly costly to maintain in distributed environments. Consider the following example:

```
SELECT *
FROM Log, Video
WHERE Log.videoID = Video.videoID;
```

Suppose new recrods have been inserted into Log. Calculating the delta view involves joining the new records with the entire table Video. While indexing is the prefered strategy to optimize such joins, many distributed systems, such as Apache Spark, Cloudera Impala, and Apache Tez, lack native support for join indices. To avoid scanning the entire table, these systems rely on partitioned joins where records linked by foreign keys are stored on the same partition. However, when these join keys cross partition lines this operation can become increasingly expensive.

3.2 Reducing Maintenance Cost With Sampling

We presented examples where these views can be expensive to maintain. In this work, we address the question of whether we need to maintain the entire view to answer aggregate queries on these views. Our proposed solution is to sample the delta view $\Delta\mathbf{V}$, and incrementally maintain just a sample of $\mathbf{V}_T$. For example, in our Select-Project view example application, we would have to parse only a sample of the inserted records. Similarly, for the Aggregation view, sampling $\Delta\mathbf{V}$ reduces the cardinality of the result and consequently communication/merging costs. And finally, for the Join views, we would only have to join a sample of the inserted records.

3.3 Outlier Indexing

We are often interested in records that outliers, which we define in this work as records with abnormally large attribute values. Outliers and power-law distributions are a common property in web-scale datasets. Often the queries of interest involve the outlier records, however sampling does have the potential to mask outliers in the updates. If we have a small sampling ratio, more likely than not, outliers will be missed.

Therefore, we propose coupling sampling with outlier indexing. That is, we guarantee that records (or rows in the view derived from those records) with abnormally large attribute values are included in the sample. What is particularly interesting is that these records give information about the distribution and can be used to reduce variance in our estimates.

3.4 Relationship to SAQP

Estimating the results of aggregate queries from samples has been well studied in a field called Sample-based Approximate Query Processing (SAQP). While the concept of estimating a correction from a sample is similar to SAQP it differs in a few critical ways. Traditional SAQP techniques apply their sampling directly to base tables and not on views. The SAQP approach to this problem, would be to treat aggregate queries on views as nested queries and then apply them to a sample of the base data [?]. Another potential technique would be to estimate the result directly from the maintained sample; a sort of SAQP scheme on the sample of the view. We found that empricially estimating a correction and leveraging an existing deterministic result lead to lower variance results on real datasets (see Section ?). We analyze the tradeoffs of these techniques in the following sections.

4. CORRECTION QUERY PROCESSING

In this section, we will describe the how to calculate a correction for stale queries. Suppose we have an aggregation query f and let $\mathbf{V}_T^{'}$ be the up-to-date view and $\mathbf{V}_T$ be the old view. The query f is stale with error ϵ if:

$$f(\mathbf{V}_T^{'}) = f(\mathbf{V}_T) + \epsilon$$

Before we discuss how sampling can scale this process, we will discuss how to correct queries on the full data.

4.1 Model for Aggregate Queries on Views

What is special about SUM, COUNT, AVG, and VAR aggregate functions is that upto proportionality constant, they can be expressed as a mean-value calculation. For example, a SUM is just the mean value times the dataset size, a COUNT is the mean occurance rate times the dataset size, and VAR is mean squared deviation. Let N be the number of tuples in the view V. These queries can also have predicates so we have to incorporate that as an indicator function (true = 1, false = 0) that skips a tuple in the aggregation if the predicate is false. When we couple these queries with predicates, we can express them in the following way:

$$\forall v_i \in V : f(V) = \frac{1}{N}\sum_{i}^{N} \phi(v_i) \cdot predicate(i)$$

We define ϕ in the following way:

Aggregation Query	$\phi(v_i)$
SUM	$N \cdot v_i$
COUNT	N
AVG	$\frac{N}{\sum_i^N predicate(i)} \cdot v_i$

4.2 Corrections For Select-Project and Foreign-Key Join Views

With the model for aggregate queries described above, we can first derive the exact value for ϵ without sampling. Since we only consider a model were records are inserted into the base tables, for these two categories of views $\mathbf{V}_T \subseteq \mathbf{V}_T^{'}$. The row differences between $\mathbf{V}_T$ and $\mathbf{V}_T^{'}$ are completely represented by the delta table $\Delta\mathbf{V}$; that is rows will only be inserted into the views. Since the aggregate queries are in the form of means, we notice that we can exploit the associativity of summations:

$$f(\mathbf{V}_T^{'}) = f(\mathbf{V}_T) + \epsilon$$

$$f(\mathbf{V}_T^{'}) - f(\mathbf{V}_T) = \epsilon$$

Upto a scaling constant c, ϵ is the aggregation function applied to the delta table.

$$c \cdot f(\Delta\mathbf{V}) = \epsilon$$

Aggregation Query	Scaling Constant c
SUM	1
COUNT	1
AVG	$\frac{\lvert\Delta V\rvert}{\lvert\Delta V\rvert + \lvert V\rvert}$

4.2.1 *Example Query Processing*

Recall, our example dataset of video streaming logs and the example selection view:

```
View1 := SELECT * FROM Log
WHERE userAgent
LIKE '%Mozilla%'
```

This query filters out the log records that came from browsers with the Mozilla tag. Let us assume that our old materialized view has 1M rows, and we receive an update of 500,000 new rows with the mozilla tag. Now we want to answer the following query:

```
SELECT avg(responseTime)
FROM View1
```

On the inserted 500,000 rows, we can run the query and call the result r_{delta}. Then, we apply the scaling constant c to the result to convert this into an ϵ and get $r_{delta} \frac{500000}{500000+1000000}$, which equals $\frac{r_{delta}}{3}$. Therefore, the up-to-date result is:

$$r_{old} + \epsilon = r_{old} + \frac{r_{delta}}{3}$$

.

4.3 Aggregation Views

The delta view is not enough information to calculate ϵ in aggregation views. Consider the following example view which we described in the last section:

```
View2 := SELECT videoID ,
max(responseTime) AS maxResponseTime
FROM Log
GROUP BY videoID;
```

Now we want to correct the following stale query.

```
SELECT COUNT(1)
FROM View2
WHERE maxResponseTime > 100ms;
```

Now suppose, View2 looks like this:

videoId	maxResponseTime
125	99
6212	160
222	145

We may get a delta table for this view of the form:

videoId	maxResponseTime_max
125	96

However, we see that when we perform the refresh operation, the updated View2 remains the same since $96 < 99$. Thus the ϵ for query is 0, even though the delta table has non-zero rows. The key point is that the refresh operation depends on the values in the view, and we need to know how these aggregates change after the refresh to estimate ϵ. Let $\mathbf{W}$ be the join of up-to-date view $\mathbf{V}_T^{'}$ and the old view $\mathbf{V}_T$ on the group-by key:

videoId	maxResponseTime_new	maxResponseTime_old
125	99	99
6212	160	160
222	145	145

However, an interesting about aggregation views is that they do not require scaling constant c as in the the other two categories of views. This is because we refresh the delta view; inferring a correction from the entire view rather than just the updates.

4.3.1 *Example Query Processing*

We can make the example in the previous section more interesting to illustrate the query processing steps. Suppose our delta table was the following:

videoId	maxResponseTime_max
125	96
1336	214

Then the joined result would be:

videoId	maxResponseTime_new	maxResponseTime_old
125	99	99
6212	160	160
222	145	145
1336	214	NULL

We can transform the example:

```
SELECT COUNT(1)
FROM View2
WHERE maxResponseTime > 100ms;
```

in terms of SQL case statements that evaluate a boolean to 1 or 0 if true or false/NULL.

```
SELECT (maxResponseTime_new > 100ms)
- (maxResponseTime_old > 100ms)

FROM Joined_View2
```

The result of this query on the example is 1 which is a correction to the stale count of videos with a max response time of greater than 100ms.

5. SAMPLING FOR APPROXIMATE CORRECTIONS

In the previous section, we described how to do the correction using the full delta view. However, this is just as expensive as incremental view maintenance. In this section, we describe how we can couple sampling with the correction calculation to save on I/O and communication costs.

5.1 Sampling for Select-Project and Foreign-Key Join Views

We can extend our deterministic corrections to approximate corrections from a simple random sample $S_{\Delta V}$ of the delta view $S_{\Delta V} \subseteq \Delta\mathbf{V}$. Recall that a simple random sample is uniform sample where every row $r \in \Delta V$ is in $S_{\Delta V}$ with equal probability p. For Select-Project and Foreign-Key Join Views, this means we have to take a sample of the updates and then apply the view definition to the sample of the updates. Formally, for every record u inserted into the table, with probability p, we include it in the sample S. Then, we take the sample updates S and apply the view definition forming $S_{\Delta V}$. Therefore,

$$c \cdot f(S_{\Delta V}) \approx \epsilon$$

The scaling constant c for the SUM and COUNT queries depends on the sample size and is $c = \frac{K}{N}$. This estimate is guaranteed to be unbiased, that is, in expectation the answer is ϵ.

5.2 Cost Analysis for Select-Project and Foreign-Key Join Views

Let n be the number of inserted records, v be the cardinality of the old view, v' be the cardinality of the new view, δ_v be the cardinality of the delta view, and p be the sampling ratio.

Scan of Updates: Both incremental maintenance and our proposed solution require at least one scan of the inserted records, and in both solutions we can load the updates into memory once and amortize that I/O cost over all views.

Delta View: For Select-project and foreign-key join views, incremental maintenance has processing cost of n records where the predicate or the join has to be evaluated for each inserted record. Our approach has a cost of np as we have to evaluate this only on our sample.

Refresh: For Select-project and foreign-key join views, incremental maintenance has to insert δ_v rows while we have to insert only $p\delta_v$ records. If there is an outlier index, this cost increases to $p\delta_v + l$.

Query: As incremental maintenance completely refreshes the view, the cost of processing a query on the view is at most v'. For Select-project and foreign-key join views, the processing cost is $v + p\delta_v$ and this is guaranteed to be less than v'.

5.3 Accuracy Analysis For Select-Project and Foreign-Key Join Views

By the Central Limit Theorem, means of independent random variables converge to a Gaussian distribution. We can apply this theorem to bound the approximation error in the correction ϵ. For $\epsilon = c \cdot f(S_{\Delta V})$, and a sampling ratio of p. Recall, from the previous section that f can be expressed as a sum of the function ϕ applied to each row. Let $\phi(S_{\Delta V})$ be set of ϕ applied to each of rows in the sample delta view. Then variance of the estimate is:

$$c^2 \frac{var(\phi(S_{\Delta V}))}{p|\Delta V|}$$

Since there is asymptotic convergence to a normal distribution, we can use this variance to bound the distance the true ϵ:

$$\epsilon \pm \gamma \cdot c \frac{std(\phi(S_{\Delta V}))}{p|\Delta V|}$$

We can set γ to our desired normal confidence level (eg. 1.96 for 95% confidence). In other words, the accuracy of the estimate depends on the variance of the inserted records.

5.4 Sampling for Aggregation Views

In the previous section, we discussed how the delta view did not contain enough information to calculate a correction. Similarly, sampling to estimate the correction for queries on Aggregation Views is more challenging. We notice that in the refreshed view each GROUP BY key is unique, and thus, to sample the refreshed view we have to sample by GROUP BY keys in the inserted records. For each inserted record we apply a hash to the cols in the GROUP BY clause, and then we take the result of the hash modulo a sampling ratio to sample the table. The result is that we ensure that every record with the same group by key is either fully in the sample or not, thus none of the rows in the delta view are approximate. Then, we refresh this sample delta view instead of the full view. We can then join this delta view with the old view to approximate ϵ.

Unlike before this is not a sample of a delta view, it is a sample of the entire view joined with the stale resuts. Thus, we denote this sample as S_V. Also, recall that unlike the other views, Aggregation Views did not have an additional proportionality constant for a full correction.

$$f(S_V) \approx \epsilon$$

However, when we introduce sampling a scaling constant is now neccessary. The scaling constant c for the SUM and COUNT queries depends on the sample size and is $c = \frac{K}{N}$, but $c = 1$ for the AVG query. For SUM, COUNT, AVG, and VAR, this estimate of ϵ is unbiased as before.

5.5 Cost Analysis for Aggregation Views

Let n be the number of inserted records, v be the cardinality of the old view, v' be the cardinality of the new view, δ_v be the cardinality of the delta view, and p be the sampling ratio.

Scan of Updates: As in the Select-Project and Foreign-Key Join views, both incremental maintenance and our proposed solution require at least one scan of the inserted records, and in both solutions we can load the updates into memory once and amortize that I/O cost over all views.

Delta View: For aggregation views, incremental maintenance has the processing cost of n and in addition an aggregation cost of δ_v where aggregates for each of the groups have to be maintained. In contrast, for aggregation views, our approach has a cost of $p\delta_v$ as we sample the group by keys and an expected processing cost of np. For aggregation views, in a distributed environment, there are potentially additional communication costs as the updates may not be partitioned by the group by key.

Refresh: For aggregation views, the cost is a little bit more complicated as we have a combination of insertions into the view and updates to the view. Incremental maintenance has to refresh δ_v rows while we have to refresh $p\delta_v$ rows. If there is a join index, in constant time, we can determine which rows are new insertions and which correspond to rows already in the view.

The costs become higher in a distributed environment as we need to consider communication and query processing engines that rely on partitioned joins rather than indices. For aggregation views, we want to partition the data by the group by key. This allows a paritioned join which only requires communication (a shuffle operation) of the delta table. Therefore, in incremental maintenance we have to communicate δ_v rows while our solution requires $p\delta_v$ rows.

Query: For aggregation views, the cost is $v + pv'$ where we calculate a correction by processing pv' rows and correct an existing aggregation of v records.

5.6 Accuracy Analysis For Aggregation Views

As in the Select-Project and Foreign-Key views the Central Limit Theorem can be used to bound the approximation. However, there are two cases with aggregation views: (1) updates to existing rows and (2) insertions into the view. Let $S_V^{(i)}$ be the set differences for those rows in the sample materialized view that are updated; that is, represents how much each attribute changes, and let $S_V^{(i)}$ be the sample materialized view that correspond to inserted records. Let w_u by the fraction of the delta view which are updates to existing rows and w_i be the fraction insertions into the view.

Recall, from the previous section that an aggregation function on the view f can be expressed as mean of the function ϕ applied to each row. Let $\phi(S_V^{(i)})$ be set of ϕ applied to each of rows that inserted, $\phi(S_V^{(i)})$ be the set of ϕ applied to the update differences, and $\phi(S_V)$ be the union of the two sets. As before, we can use the Central Limit Theorem to calculate the variance of the estimate and bound the approximation error. The variance of a mixture of two distributions is:

$$\sigma^2 = w_1((\mu_1 - \mu)^2 + \sigma_1^2) + w_2((\mu_2 - \mu)^2 + \sigma_2^2)$$

Applying this in our setting, let us define the following values:

$$\mu_{diff}^{(i)} = mean(\phi(S_V^{(i)})) - mean(\phi(S_V))$$

$$\mu_{diff}^{(u)} = mean(\phi(S_V^{(i)})) - mean(\phi(S_V))$$

$$\sigma_{(i)}^2 = var(\phi(S_V^{(i)}))$$

$$\sigma_{(u)}^2 = var(\phi(S_V^{(u)}))$$

$$\sigma_{diff}^2 = w_i((\mu_{diff}^{(i)})^2 + \sigma_{(i)}) + w_u((\mu_{diff}^{(u)})^2 + \sigma_{(u)})$$

Therefore, the estimate variance is:

$$c^2 \frac{\sigma_{diff}^2}{|S_V|}$$

And as before there is asymptotic convergence to a normal distribution, we can use this variance to bound the distance the true ϵ:

$$\epsilon \pm \gamma \cdot c \frac{\sigma_{diff}^2}{|S_V|}$$

We can set γ to our desired normal confidence level (eg. 1.96 for 95% confidence).

To interpret σ_{diff}^2, it not only has dependence on the variance of the inserted records, but also on the variance of the updates and the relative difference between the updates and inserted records. A worst case, would be if there is a 50-50 split between updates and insertions and changes to existing rows are in distribution very different than the new inserted rows in the materialized view.

6. OUTLIER INDEXING

One feature of many large datasets is long-tailed distributions[?]. Such distributions pose a challenge to sample-based approaches as the distributions often have variances that are orders of magnitude larger than the mean. As a result, answering queries such as SUM, COUNT, AVG may require a large sample size for sufficient accuracy.

Furthermore, the outliers themselves are interesting, and often these are the most frequently queried records. An application of particular interest for up-to-date query results is outlier detection. When we have growing datasets, for example activity logs, we may want to know which records correspond to abnormal activity.

The key question is how can we incorporate guarantees about outliers and efficiency guarantees on long tails in our estimation framework. The approach is to build an outlier index similar to that proposed in prior work [?]. The user specifies an attribute in the base table and a percentage $o\%$, and we identify the records whose attribute value is in the top o percentile. We then construct an index to ensure that rows in the materialized view that are derived from the top-o percentile records are guaranteed to be in the sample.

Recall, our example base table:

```
Log(sessionID, videoID, responseTime, userAgent)
```

and the selection view:

```
View1 := SELECT * FROM Log
WHERE userAgent
LIKE '%Mozilla%'
```

If the user creates an outlier index on "responseTime" with $o = 1\%$, then the records containing the top 1% of response times are guaranteed to be in the sample of the delta view regardless of the sample size.

6.1 A Framework For Outlier Indexing

A key aspect of our outlier index is that it is derived on the base table, yet can still give benefits to queries on the view. This has a few advantages: (1) the outlier index does not require materialization of the full delta table, (2) the cost of building the index amortizes over all the materialized views, and (3) it can be built with a single pass of the updates. In principle, other outlier detection schemes [?] can be applied to build the index, but in this work, we implement and analyze an index of top o%. In attributes with both positive and negative values, this can also be the records with top o% in absolute value terms.

6.2 Building The Outlier Index

The first step is that the user selects an attribute to index and specifies the parameter $o\%$. In a single pass of only the inserted records, the index is built storing references to the records in the top o%. For Select-Project and Foreign-Key Join views, this is sufficient as we can additionally iterate through the index when sampling and mark which records satisfy the view definition and perform any neccessary joins. This adds an additional scanning cost of $o\%$ to the creation of each view, but we envision that in most datasets the outlier index will be very small. In our experiments, we show that even a very small outlier index (less than .01% of records) is sufficent to greatly improve the accuracy of correction estimates.

For aggregation views, at sample creation time, we must ensure that all rows in the materialized view that are derived from a record in the outlier index are added to the view. This means that we have to run a COUNT(DISTINCT) for the group by key of the view on the outlier index and then ensure that all of those keys are added to the view (that is ensure they are sampled).

6.3 Query Processing with the Outlier Index

The outlier index has two uses: (1) we can answer SELECTION queries on the outliers, and (2) we can improve the accuracy of our AGGREGATION queries. For selection queries, we must simply check to see if the record satisfying the query is in the outlier index. While the outlier index is small, outliers (and their derived materialized view rows) are often the records that we want to query. To see a more general handling of Selection Queries, see Section [?].

We can also incorporate the outliers into our estimates of the correction ϵ for aggregation queries. By guaranteeing that certain rows are in the index, we have to merge a deterministic result (set of outlier rows) with the estimate. One way to think of this is that we have ϵ is calculated from the set of records that are not outliers. Let $|V_T'|$ be the size of the updated view, l be the number of rows in the outlier index, and let D_o be the set of differences (as defined in Section ?) for the rows in the outlier index. We can update ϵ with the outlier information by:

$$\frac{|V_T'|}{|V_T'| - l}\epsilon + c \cdot f(D_o)$$

6.4 Increased Accuracy For Long-Tailed Distributions

This outlier indexing procedure can greatly increase the accuracy of estimates where the set of difference is heavy tailed. This approach has been well studied in AQP [?] and is called truncation in Statistics [?]. The intuition is that by removing the tail, you are reducing the variance of the distribution, and thus, making it easier to estimate an aggregate from a sample.

7. EXTENSIONS

7.1 Proof of Optimality of Correction

7.2 Selection Queries

7.3 Different Types of Aggregation Views

8. RELATED WORK

9. RESULTS

9.1 Experimental Setting

We evaluated our approach on two different datasets and workloads: (1) TPCD-Skew and (2) Conviva. The TPCD-Skew dataset (1) has the same schema as the TPCD benchmark dataset but sets attribute values drawn from a Zipfian powerlaw distribution instead of uniformly. The Zipfian distribution [?] is a long-tailed distribution with a single parameter $z = \{0, 1, 2, 3, 4\}$ which a larger value means a more extreme tail. The dataset is in the form of a generation program which can generate both the base tables and a set of updates. For this dataset, we applied our approach to three materialized views, each of a different type:

Select-Project View

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

Aggregation View

```
SELECT l_orderkey,
       l_shipdate,
       sum(l_quantity) as quantity_sum,
       sum(l_extendedprice) as extendedprice_sum,
       max(l_receiptdate) as receiptdate_max,
       count(*) as group_count
FROM LINEITEM
GROUP BY l_orderkey, l_shipdate
```

Foreign-Key Join View

```
SELECT supplier.*,
          customer.*
FROM   customer,
       orders,
       lineitem,
       supplier,
       partsupp
WHERE  c_custkey = o_custkey
   AND o_orderkey = l_orderkey
   AND l_suppkey = ps_suppkey
   AND l_partkey = ps_partkey
   AND ps_suppkey = s_suppkey
   AND s_nationkey <> c_nationkey
```

For these three views, we randomly generated aggregation queries and evaluated them for different sample sizes, different update rates, and different parameter settings for the Zipfian distribution. As a baseline, we evaluate against the following techniques: SAQP, Full Incremental Maintenance, and Recalculation. In SAQP, rather than estimating a query correction, we incrementally maintain a sample of the view and estimate aggregate queries from only the sample.

9.2 Query Correction

9.2.1 Sample Size and Accuracy

In this experiment, we illustrate the tradeoff between sample size and accuracy in the TPCD Skew dataset with $z = 2$. For each of the three views above, we apply the views to a 10GB dataset. Then, we simulate 5000000 inserted records, corresponding to about 800MB. We then vary the sampling ratio for the three types of views and show how much sampling is needed to acheive a given query accuracy. For 10,000 randomly generated aggregation queries on each view, we compare the stale query error to SAQP and our approach. For SAQP and our approach, we keep the same sampling ratio. This does, however, mean that SAQP uses significantly more storage than our approach Select-Project and Foreign-Key Join Views.

In Figure 9.2.1, we show the accuracy as a function of sampling ratio for each of the views. For both SAQP and our approach, there is a break-even point at which the approximation error is less than the staleness error. At sampling ratios beyond this point, our query approximated queries are on average more accurate than queries on the stale view. For all three of the views, the average query error is significantly less than SAQP as the break-even point happens earlier in our approach.

9.2.2 Update Rate and Accuracy

In a second experiment, we evaluated the same views and queries but varied the update rate for a fixed sample size. We set the sample size to 5% and then vary the number of inserted records by increments of 500000 records to a final count of 8000000 records (1.3GB). Like before, we evalute SAQP and our approach against a baseline stale result. In Figure **??**, we show the results of the experiment.

The results highlight an important point comparing the accuracy of SAQP to our approach. The accuracy our approach is proportional to the amount of correction needed, while SAQP keeps a roughly constant accuracy. As more records are inserted the approximation error in our approache increases. However, we find that even for very large amounts of inserted records (>10% of dataset size), our approach gives significantly more accurate results than SAQP. The gain is most pronounced in aggregation views where there are a mixture of updated and inserted rows into the view. Compensationg for a correction to an existing row is often much smaller than doing so for a new inserted record.

9.2.3 Distribution of Query Error

In the previous two experiments, we looked at the average error for the queries on the views. In this experiments, we looked at the distribution of query errors and compared that to their staleness. For each of the three views, we derive the views from a 10GB dataset. Then, as before, we simulate 5000000 inserted records and a 5% sample. We evaluated the relative error for each query.

In Figure 9.2.4, we present staleness vs. estimation accuracy as a scatter plot. For all three of the views, the median error is greater than 10 times less than the staleness. However, we do see that there are outliers. Outliers can occur for a variety of reasons. First, some of the generated queries are highly selective and a uniform sampling approach may not sample enough records to answer it accurately. Next, outliers can be caused by large updates to the data that are missed by our sampling. In Section ?, we show how outlier indexing can solve the latter problem and, in fact, after outlier indexing all of our queries on these three views are estimates more accurately than the pre-existing staleness error.

9.2.4 *Computational Efficiency*

We have looked at how small samples can still give highly accurate results. In this section, we evaluate the efficiency of sampling in terms of computation. We evaluate our approach on a single r3.large Amazon EC2 node with a MySQL database. For a batch of updates, we evaluate how long it takes to incrementally maintain a view compared to maintaing a sample. For, the join view, we build an index on the foreign key of the view. As before, we derived the views from a 10GB base dataset, and inserted records in increments of 500000.

We find that maintaining a sample is far cheaper than the entire view. In this experiment, keeping a view exactly up-to date required computation on the order of minutes. However, we found that with a small sample we could acheive this in a few seconds and still acheive accurate results.

9.3 Outlier Indexing

TODO

9.4 Application: Conviva Dataset

Conviva is a video streaming company that logs activity for each of their videos such as who is watching the video, who owns the video, browser, and network latency etc. Analysts from the company answer analytics queries on this dataset such as how many times is a certain video watched. We experimented with a 1TB dataset of these logs and the corresponding queries on this dataset. The dataset is a single relation with 103 attributes. We used the query logs to generate three materialized views.

View 1

```
SELECT clientId,
       customerId,
       sessionTimeMs,
       count(1)
         as group_count,
       max(sessionTimeMs)
         as sessionTimeMs_max,
       avg(lastBufferLengthMs)
         as lastBufferLengthMs_avg,
       avg(lLifePausedTimeMs)
         as lLifePausedTimeMs_avg
FROM anon_sdm2_ss
GROUP BY clientId, customerId
```

View 2

```
SELECT customerId,
       sessionType,
       count(1) as group_count,
       max(estBwCount)
            as estBwCount_max,
       sum(buffTimeMs)
            as buffTimeMs_sum,
       sum(estBwCount)
            as estBwCount_sum
FROM anon_sdm2_ss
GROUP BY
customerId, sessionType
```

View 3

```
SELECT justStarted,
       sessionType,
       startResourceState,
       state,
       count(1)
            as group_count,
       max(playTimeMs)
            as playTimeMs_max,
       min(stoppedTimeAtJoinMs)
            as stoppedTimeAtJoinMs_min,
       max(lifeAverageBitrateKbps)
            as lifeAverageBitrateKbps_max
FROM anon_sdm2_ss
GROUP BY justStarted,
         sessionType,
         startResourceState,
         state
```

All three of these views are aggregation views. View 1 has the most selective group by clause and was chosen to a be a large view where most of the maintenance is in the form of new rows to insert. View 2 is a medium size view where maintenance is a mix of both updates and insertions. View 3 is a small view where most of the maintenance is updates to existing rows. For these views, we randomly generated aggregation queries.

9.4.1 *Accuracy in Conviva*

We evaluated the average query accuracy for different sample sizes and numbers of records inserted. Figure 9.4.1, compares this accuracy to the staleness of the query. We find that even a 0.1% sample gives significantly more accurate results for View 1 and View 2. Even in the situation where the view is small, sampling can still have benefits as seen in View 3.

9.4.2 *Performance in Conviva*

We evaluated performance on Apache Spark, on a 20 node r3.large Amazon EC2 cluster. This cluster had enough memory to hold all of the materialized views in memory. Spark supports materialized views through a distributed data structure called an RDD [?]. The RDD's are immutable, thus requiring significant overhead to maintain. We compare maintenance to recalculation of the view when the data is in memory and when only the updates are in memory. We derived the views from a 120GB base relation and inserted records in 10GB increments. As Spark does not have support for indices, we rely on partitioned joins for incremental maintenance of the aggregation views. Each view is partitioned by the group by key, and thus only the sampled delta view has to be shuffled.

In Figure 9.4.2, we illustrate the maintenance time as a function of the number of inserted records. We find that we acheive good scalability on view 1 and view 2, and still have some performance gains on view 3. In view 1 and view 2, our gains are more pronounced due to the savings on communication in a distributed environment. As view 3 is smaller, the communcation gains are less and the only savings are in computation.

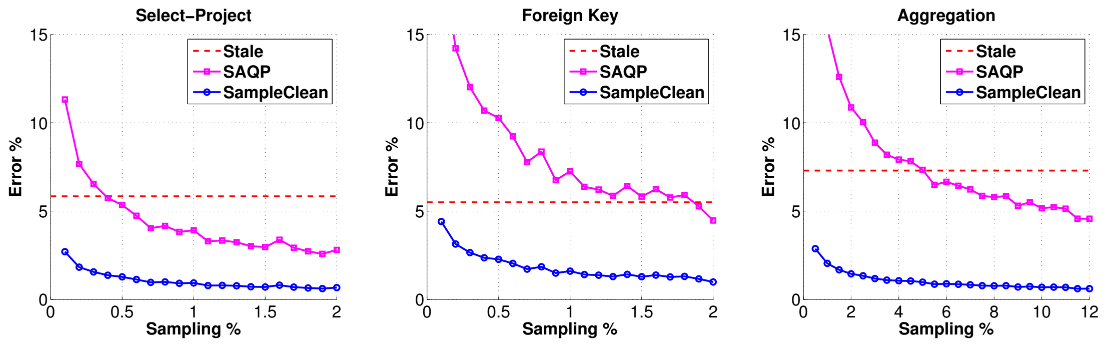


Figure 3: TODO

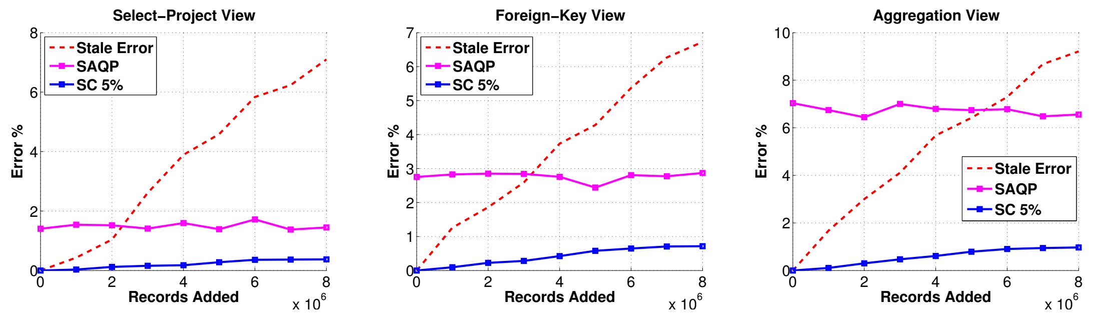


Figure 4: TODO

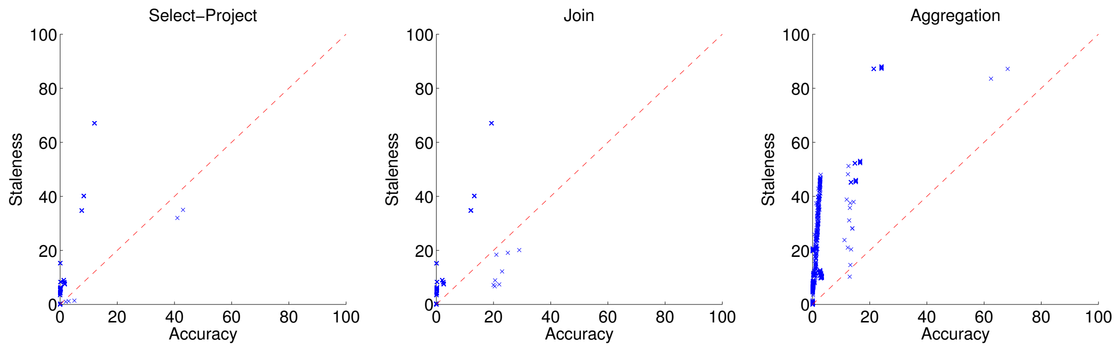


Figure 5: TODO

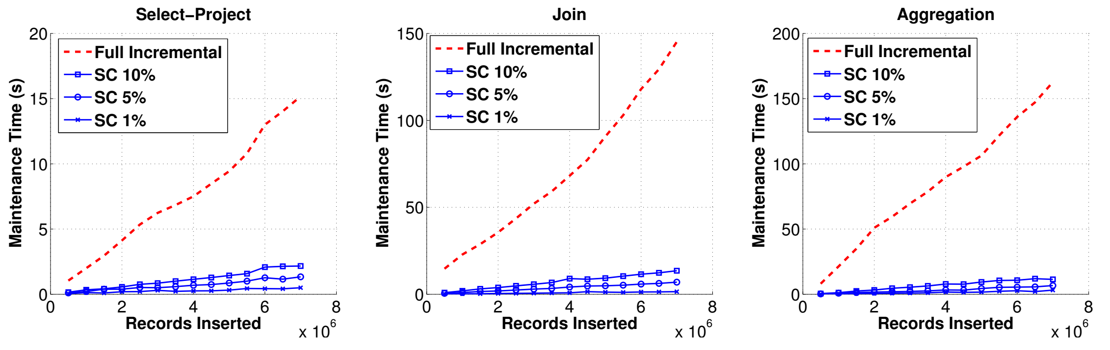


Figure 6: (placeholder) TODO

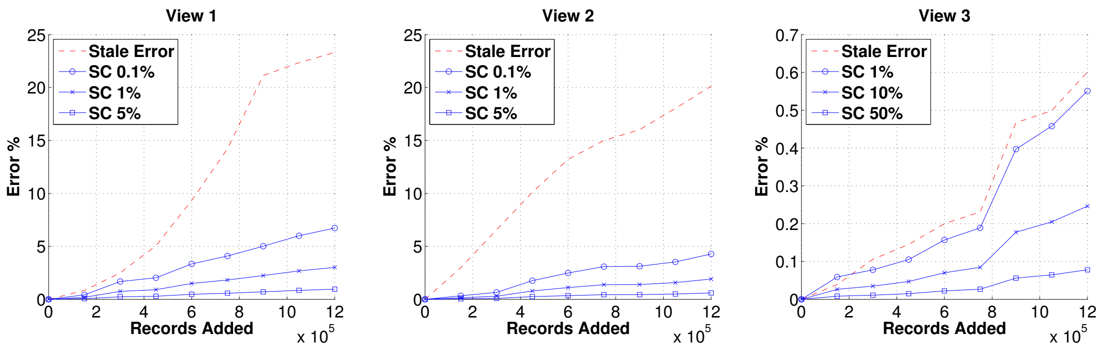


Figure 7: TODO

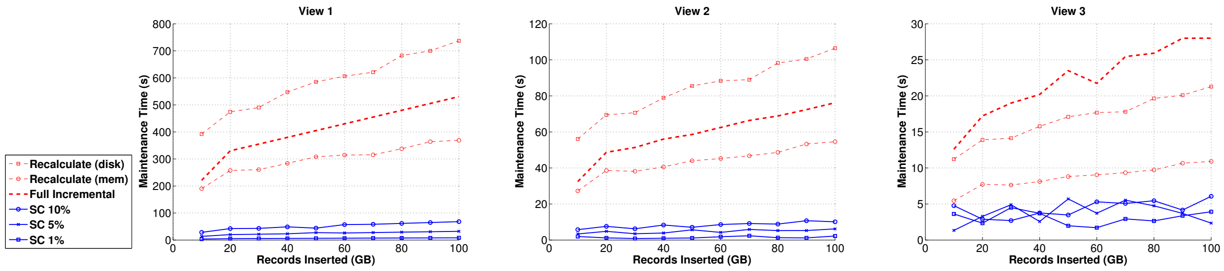


Figure 8: (placeholder) TODO