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breaks down into two components: applying the view definition to the updates, and then writing the "delta" view to the out-of-date view.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 *stalness* 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.

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 importanttopic 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 Problem Statement

Deferred maintenance

In this work, we use sampling infer up-to-date results for aggregation queries on stale materialized views. The key idea is that we can apply incremental maintenance to just a sample of the view, thus greatly reducing the cost needed to maintain the view (Figure ?). 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 restrict our analysis to only insertions into the database and defer analysis of updates and deletions for future work.

2.3 Taxonomy of Supported Materialized Views

In this section, we will introduce the taxonomy of materialized views that can benefit from our approach. In particular, we provide example situations when view maintenance can be costly. First, we will illustrate a simple model of incremental view maintenance, see [?] for a detailed survey. For example, if the view only consists of selections and projections then the merge operation is simply a union of $\mathbf{V}_T$ and $\Delta\mathbf{V}$. On the other hand, if the view is an aggregation view then it may involve updating records in $\mathbf{V}_T$ as records containing an existing GROUP BY clause may appear in the updates. Refer to [?] for the query processing details.

2.3.1 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 1). 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 a view of this dataset filtering records to those that correspond users who used MacOS X and Safari. This involves evaluating regular expression on the string to see if it matches a criteria (eg. contains "Mac OS X" and contains "Safari"). 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.

2.3.2 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. If the cardinality of the delta view is large then it will be more costly to merge this result with the existing view. These costs increase in a distributed environment where a larger delta view means that more data has to be communicated through a shuffle operation.

2.3.3 *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. In the example above, suppose new recrods have been inserted into table1. Calculating the delta view involves joining the new records with the entire table2. 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.

2.4 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.

2.5 Staleness and Correcting Staleness

Given the three categories of views and the SUM, COUNT, AVG, and VAR queries on these views, we can formalize the concept of staleness. Let $\mathbf{V}_T$ be the old view, and $\mathbf{V}_T^{'}$ be the up-to-date view. If f is an aggregation function, then we call the staleness error of the query ϵ if :

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

Since we already have the out-of-date view, we can easily compute $f(\mathbf{V}_T)$. However, to get an up-to-date result, we need an estimate of ϵ. In [?], the authors proposed an algorithm called "NormalizedSC" which estimated the error term ϵ from a sample of the *difference set*. The difference set is defined in the following way: for a set of tuples $v_i^{'} \in \mathbf{V}_T^{'}$ and $v_j \in \mathbf{V}_T$ such that if tuple i is in both $\mathbf{V}_T^{'}$ and $\mathbf{V}_T$ then value $v_i^{'} - v_i$ is included and if not the value v_i' is included. If we denote this set as $\mathbf{V}_T^{'} - \mathbf{V}_T$, we can take a sample $S \subseteq \mathbf{V}_T^{'} - \mathbf{V}_T$, and then we can apply the query to the sample $f(S) \approx \epsilon$. The key result from [?] is that for a large enough sample size $f(S) \sim N(\epsilon, \frac{\sigma_{diff}^2}{k})$; that is the estimate is centered around the true value with variance proportional to the differences and inversely proportional to the sample size. With the an estimate $f(S)$, we can then correct a stale query $f(\mathbf{V}_T) - f(S) \approx f(\mathbf{V}_T^{'})$. This estimate is unbiased and probabilistically bounded.

We address a few new challenges in this work. First, NormalizedSC was designed in the context of static tables and data errors that can be modeled record transformations. In this context, it is clear how to sample the set of differences. We can sample the dirty table and then transform the records, and then take the differences. In the following sections, we will describe how address the new challenges of insertions. A further addition, is we consider the effects of outliers on the estimate of NormalizedSC and characterize the optimality of the NormalizedSC algorithm.

2.6 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.

3. CORRECTION APPROXIMATE QUERY PROCESSING

In this section, we will extend the NormalizedSC algorithm to estimate corrections for queries on stale views. The key challenge is to estimate ϵ such that:

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

3.1 Select-Project and Foreign-Key Join Views

We will 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. The aggregation functions SUM, COUNT, AVG, and VAR are special as they can be expressed as summations. The summative forms lead to the following insight:

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

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

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

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

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

3.1.1 Sampling the Delta View

We can extend this theory to 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,

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

Due to the summative forms and uniform sampling, we can apply the Central Limit Theorem to bound the approximation error. Sums of independent random variables converge to a normal distribution, and further more the expected value of $c \cdot f(S_{\Delta V})$ is ϵ.

$$c \cdot f(S_{\Delta V}) \sim N(\epsilon, \frac{\sigma^2_{diff}}{k})$$

$$f(\mathbf{V}_T) + c \cdot f(S_{\Delta V}) \sim N(f(\mathbf{V}'_T), \frac{\sigma^2_{diff}}{k})$$

σ^2_{diff} is an interesting parameter as it quantifies the variance of the delta view. In Section ?, we will analyze this parameter and describe how the statistics of the updates affect the estimate accuracy.

3.2 Aggregation Views

Applying the view definition to the updates is not enough information to calculate ϵ in aggregation views. Consider the following example view and query pair:

```
View1 := SELECT col2 , MAX(col1) as col1_max
FROM table
GROUP BY col2

Query1 := SELECT AVG(col1_max)
FROM View1
```

col1	col2
3	1
6	1
2	2

col2	col1_max
1	6
2	2

Suppose records are inserted into **table** and we can apply the the definition View1 to set of inserted records:

col1	col2
3	1
6	1
2	2
2	**1**

col2	col1_max
2	1

However, we see that when we perform the merge operation, the updated View1 remains the same, thus the ϵ for Query1 is 0, even though the delta table has non-zero rows:

col2	col1_max
1	6
2	2

The key point is that the merge operation depends on the aggregations in view definition, and we need to know how these aggregates change after the merge 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. To make the example above more interesting we can insert a few more records, $\mathbf{W}$ would be:

col1	col2
3	1
6	1
2	2
2	**1**
4	**2**

col2	col1_max_new	col1_max_old
1	6	6
2	4	2

The value of ϵ for Query1 would be the avg query applied to the difference:

```
QueryEpsilon1 := SELECT AVG(col1_max_new−col1_max_old)
FROM W
```

3.2.1 Sampling the Merged View

We realize that in the merged view each GROUP BY key is unique, and thus, to sample the merged 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 merge this sample delta view with the old view to get the set of differences.

4. OUTLIER INDEXING

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. Up-to-date query results, as opposed to long periods of staleness, have to potential to detect these outliers quickly. However, as we use sampling to processes aggregate queries on the views, this has a potential to mask outliers. Our framework can be extended to guarantee that outliers records will be incorporated into the sample. In this section, we discuss how not only are these outliers themselves of interest but that these outliers give information about the distribution of an column, and can greatly improve the accuracy of estimates.

4.1 A Model For Outlier Indexing

We propose the following model for detecting and indexing outliers. When creating a view, the user specifies an attribute in the base relation to "outlier index". What this means is the that the records with the l largest attribute values are guaranteed to be included in the sample view. In this outlier model, we detect outliers in updates with a single pass and without having to build the entire delta table. For aggregation views, for the records that are indexed as outliers, we simply add those group by keys to the sample.

4.2 Query Processing with the Outlier Index

We can incorporate the outliers into our estimates of the correction ϵ. 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:

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

4.3 Increased Accuracy For Heavy-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.

5. ANALYSIS

5.1 Cost Analysis

Scan of Updates.

Delta View.

Merge.

Query.

5.2 Variance Analysis

5.3 Optimality of Corrections