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breaks down into two components: applying the view definition to the updates, and then writing the "delta" view to the out-of-date view.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

Similarly, in context of materialized views, we address the key question of whether we have to maintain the entire view to answer aggregate queries with bounded error. In this work, we 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.

2.3 Extending SampleClean For 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 $f(\mathbf{V}_T)$. In this work, we look at correcting staleness by estimating ϵ from the sample. In other words, we use sampling to learn how updates affect the aggregation query on the view and then correct the query results.

In [?], the authors proposed an algorithm called "NormalizedSC" which estimated the error term ϵ from a sample of the *difference set*. We address a few new challenges in this work. First, NormalizedSC was designed in the context of static tables and data dirtiness that can be modeled as record transformations. In this work, we notice that correcting staleness can imply inserting new rows into the view as well as transforming old ones. A further addition, is we consider the effects of outliers on the estimate of NormalizedSC and characterize the optimality of the NormalizedSC algorithm.

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.

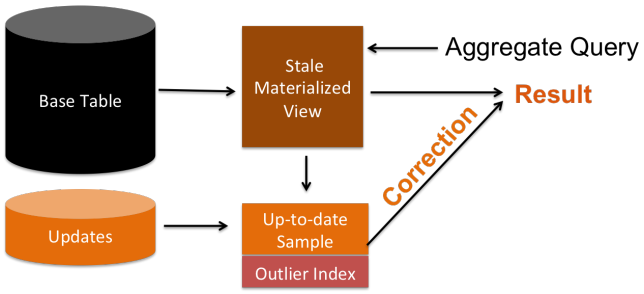

Figure 1: TODO

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

5.1 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.

5.2 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:

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

5.3 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.

6. ANALYSIS

6.1 Cost Analysis

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, p be the sampling ratio, and l be the size of the outlier index.

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. If a user chooses to add an outlier index on an attribute in the updates building this index also requires a single scan of the inserted records. However, as the index is based on the base relation and not the delta view this can be populated when the record is inserted.

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. 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. If there is an outlier index, the cost increases additively to $p\delta_v + l$. 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: Let us first analyze the refresh step in a single node system with a join index. 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 $p\delta_v + l$. 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: As incremental maintenance completely refreshes the view, the cost of processing a query on the view is at most v'. In our approach, we use the old view, sample, and outlier index to process a query. For Select-project and foreign-key join views, the processing cost is $v + p\delta_v + l$ and this is guaranteed to be less than v'. For aggregation views, the cost is $v + pv' + l$ where we calculate a correction by processing $pv' + l$ rows and correct an existing aggregation of v records.

6.2 Variance Analysis

We will first analyze the variance of the estimate ϵ without the outlier index. In Section ?, we discussed how the variance of ϵ is the variance of the difference set. For Select-project and foreign-key join views, this variance can be analytically derived from the variance formula for statistical mixtures [?].

$$f(\Delta V)\frac{\delta_v}{v'}(1 - \frac{\delta_v}{v'}) + var(\Delta V)\frac{\delta_v}{v'} \quad (1)$$

For Aggregation views, the accuracy depends on both the variance of the inserted rows and the updates to existing rows. <TODO>

6.3 Optimality of Corrections

<TODO>

7. RESULTS

7.1 Experimental Setting

7.2 Query Correction

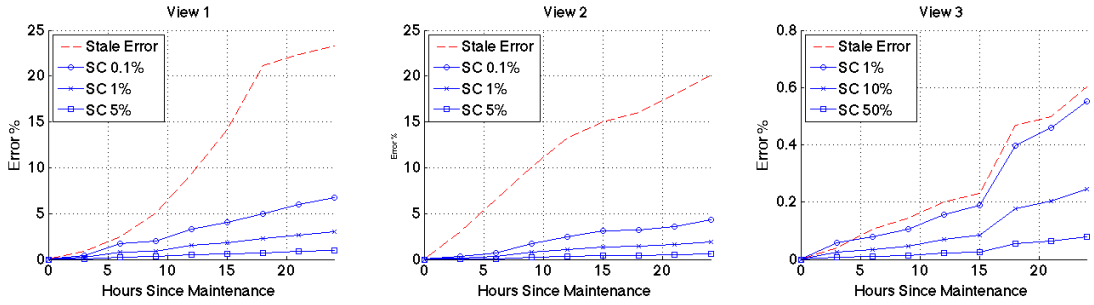

Figure 3: TODO