Overcoming Limitations of Sampling for Aggregation Queries

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Gautam Das
Microsoft Research
gautamd@microsoft.com

Mayur Datar *
Department of Computer Science
Stanford University
datar@cs.stanford.edu

Rajeev Motwani
Department of Computer Science
Stanford University
rajeev@cs.stanford.edu

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

Abstract

We study the problem of approximately answering aggregation queries using sampling. We observe that uniform sampling performs poorly when the distribution of the aggregated attribute is skewed. To address this issue, we introduce a technique called outlier-indexing. *Uniform sampling is also ineffective for queries with low selectivity. We rely on weighted sampling based on* workload information *to overcome this shortcoming. We demonstrate that a combination of outlier-indexing with weighted sampling can be used to answer aggregation queries with significantly reduced approximation error compared to either uniform sampling or weighted sampling alone. We discuss the implementation of these techniques on Microsoft's SQL Server, and present experimental results that demonstrate the merits of our techniques.*

1 Introduction

Decision support applications such as On Line Analytical Processing (OLAP) and data mining tools for analyzing large databases are gaining popularity. Executing such applications on large volumes of data can be resource-intensive. Fortunately, small *samples* of the data can be used by data mining and statistical techniques effectively without significantly compromising the accuracy of their analysis. Likewise, OLAP servers that answer queries involving aggregation can potentially benefit from the ability to use sampling.

There are at least two factors why significant errors may be introduced if uniform random samples are used for approximating results of aggregation queries over relational databases. These two factors are presence of *data skew* and *low selectivity* of queries. A skewed database is characterized by the presence of outlier values that are significantly different from the rest in terms of their contribution to the aggregate. Unless special care is taken to handle the effect of these outliers, uniform sampling over a skewed database is susceptible to significant error. Uniform sampling is also hard to exploit for selection queries with aggregates. Very few or no tuples in the sample may satisfy the query predicate. Extrapolating the aggregate from such a small set of tuples to the entire data set can can lead to error in estimating the aggregates.

In this paper, we suggest techniques to overcome these limitations of uniform random sampling. First, in order to address problems arising out of skew in data, we isolate values in the data set that could contribute heavily to the error in sampling. We refer to this technique as *outlier-indexing*. Next, we exploit *workload information* to overcome limitations in answering queries with low selectivity. This is motivated by the observation that in general no single uniform sample can answer low selectivity queries with sufficient accuracy. However, by using a representative workload, we can tune our selection of samples[1] (as well as outlier-indexes) such that for queries in the workload, we can significantly improve accuracy.

For much of this paper we illustrate our techniques for the class of single table queries involving selection and group-by queries with the *sum* aggregate. However, our techniques naturally extend to a broader class of queries, containing foreign key joins as well as other aggregation functions. We discuss these extensions in Section 4.3.

*This work was done while the author was visiting Microsoft Research.

[1]Most of the techniques that we propose can either use online samples (i.e., computed during query processing), or precomputed samples of the base relations (see also Section 5).

A key contribution of the paper is the experimental evaluation of the proposed techniques based on an implementation on Microsoft SQL Server. Our technique demonstrates that the combination of outlier-indexing and weighted sampling, based on workload information, results in significant reduction in error compared to uniform or weighted sampling alone.

The rest of the paper is organized as follows. In Section 2 we discuss related work. In Section 3 we point out the limitations in using uniform sampling for aggregation queries. In Sections 4 and 5, we describe outlier-indexing and workload-sensitive sampling respectively. We present an experimental evaluation in Section 6 and conclude in Section 7.

2 Related work

Sampling based methods have been used in a wide variety of scenarios in databases, such as query selectivity estimation in query optimization, providing sampling as a relational operation, and approximate query answering [18]. We now discuss related research on the problem of approximately answering *aggregation queries* which is the focus of our paper.

The work of Hellerstein, Haas, and Wang [12] uses an *online* sampling technique for answering aggregation queries that provides the user with a time-accuracy tradeoff. The paper by Hellerstein and Haas [10] introduces novel join methods that are best suited for such an interactive architecture. Our techniques can be integrated with this architecture to derive the benefits that the architecture provides, while at the same time addressing some of its limitations. One of the important limitations addressed in our work is their assumption that there is little variability in the data.

Acharya, Gibbons, Poosala, and Ramaswamy [2] proposed the use of *synopses* (i.e., precomputed samples of relations) for answering aggregation queries. Gibbons and Matias [9] developed techniques for the fast incremental maintenance of summary statistics, and considered their application to providing approximate query answers.

A key technique introduced in our paper is *weighted sampling* of the data by exploiting *workload information*. Recently, Ganti, Lee, and Ramakrishnan have independently developed a weighted sampling scheme that also exploits workload information to continuously tune a representative sample of the data [8]. Although the work by Acharya, Gibbons and Poosala [1] proposes a weighted sampling scheme, they do not explicitly leverage workload information. Instead, their sampling scheme tries to accommodate all possible group-by queries.

None of the above papers address the problem of *data skew* that we have addressed in this paper through the technique of outlier-indexing. Outlier detection has a long history in statistics [3, 4, 11] and has also been considered in data mining [14, 15, 16]. In particular, the work in [14] is similar to ours in that it explores the concept of detecting and removing *deviants* in time series data. The differences are that (1) we do not deal with time series data, and (2) our outlier detection algorithm only minimizes data skew of the remaining data, while their algorithm attempts a more complex minimization, i.e., the error of the histogram representation of the remaining data. Consequently, our algorithm is more efficient than the one in [14]. Finally, our technique for outlier indexing is structurally a horizonal partition (subset) of the data. Thus, it is related to *partial indexes* [19].

3 Study of limitations

In this section, we demonstrate the limitations of uniform sampling in answering aggregation queries. We discuss two problems that adversely affect the accuracy of sampling-based estimations: (1) presence of skew in aggregate values, and, (2) the effect of low selectivity in selection queries as well as the presence of small groups in group-by queries.

3.1 Effect of data skew

The following example demonstrates the adverse impact of skew on the applicability of uniform sampling.

Example 1 *Consider a relation R with $10,000$ tuples of which 99% have value 1 in the aggregate column C, while the remaining 1% of the tuples have value 1000 in C. Thus the sum over all tuples of R is $109,900$.*

Consider using a 1% uniform random sample of R (i.e., 100 tuples) to estimate this sum. The idea is to compute the sum of values in the sample, and multiply the result by 100 (multiplying by the inverse of the sampling fraction is necessary since each tuple in the sample "represents" 100 tuples of R).

It is quite likely that the sample would not include any tuple of value 1000, leading to an estimate of $10,000$ for the sum of R. On the other hand, if perchance two or more tuples of value 1000 were to be included in the sample, then our estimate of the sum of R would be more than $209,800$. In either case, the estimate would be far from the true value which is $109,900$. Only in the event where we get exactly one tuple of value 1000 in the sample, we would obtain a reasonable estimate of the average value. But this event has probability only 0.37. Therefore, with probability of 0.63, we would get a large error in the estimate. ∎

Although the above example demonstrated the limitations for the *sum* aggregate, similar arguments also hold for the aggregate *avg* (average). Thus a skewed database

is characterized by the existence of certain tuples that are deviant from the rest in terms of their contribution to the aggregate value. We refer to these tuples as *outliers*. The following theorem from [7] quantifies the contribution of these outliers to the error in estimating the *sum* via uniform sampling (a similar theorem also holds for the error in estimating the average via uniform sampling).

Theorem 1 *Consider a relation R of size N and let $\{y_1, y_2, \ldots, y_N\}$ be the set of values associated with the tuples in the relation. Let U be a uniform random sample of the y_i's of size n. Then $Y_e = (N/n) \sum_{y_i \in U} y_i$ is an unbiased estimator of the actual sum $Y = \sum_{i=1}^{N} y_i$ with a standard error (i.e., standard deviation) of*

$$\epsilon = \frac{NS}{\sqrt{n}} \sqrt{1 - \frac{n}{N}} \tag{1}$$

where S is the standard deviation of the values in the relation, defined as

$$S = \sqrt{\frac{\sum_{i=1}^{N} (y_i - \overline{Y})^2}{N-1}}$$

If there are outliers in the data then S could be very large. In such a case, for a given error bound, we will need to increase the sample size n. In the work of Hellerstein et al. [12], they assume that the aggregate attributes are not skewed. Therefore, the confidence intervals provided with their estimate could be severely affected by the presence of skew. Finally, note that although histograms are widely used as statistical summaries in databases, it remains non-trivial to leverage histograms to alleviate the resulting error in sampling.

3.2 Effect of low selectivity and small groups

Since most queries involve selection conditions and/or group-by's, it is important to study their interaction with sampling. We observe that if the selectivity of a query is low, then it adversely impacts the accuracy of sampling-based estimation. A selection query partitions the relation into two sub-relations: tuples that satisfy the condition (*relevant* sub-relation) and tuples that do not. If we sample uniformly from the relation, the number of tuples that are sampled from the relevant sub-relations will be proportional to its size. If this relevant sample size is small due to low selectivity of the query, it may lead to large error. The same is true for group-by queries which partition the relation into numerous sub-relations (tuples that belong to specific groups). Thus for uniform sampling to perform well, the relevant sub-relation should be large in size, which is not the case in general (see also [1] and [8]).

In the next two sections we propose solutions to the two problems of data skew and low selectivity of queries.

4 Handling data skew: Outlier-indexes

As seen earlier from Example 1 and Theorem 1 presented in Section 3, we know that a large variance in the aggregate column could lead to unacceptably high errors. The large variance is primarily due to the presence of certain outliers or deviants in the data. Thus, a natural idea would be to deal with outliers separately, and sample from the rest of the relation.

This leads to our idea of *outlier-indexing*, whereby we identify the tuples with outlier values and store them in a separate sub-relation.[2] The basic insight is that we can now use the following efficient scheme for estimating a query's result. Consider a selection query with the *sum* aggregate. First, apply the query to the *outlier values* and thus determine the true result of the query on the *part of the table which only includes the outlier values*; next, pick a uniform random sample from the *part of the table that does not include the outlier values* (i.e., the *non-outliers*), and estimate from the sample (as in Theorem 1) an approximation to the true result of the query if applied to the non-outlier tuples; finally, combine the two results to obtain an overall estimate of the query's true result. It is clear that for Example 1 (presented in Section 3), the proposed technique would result in a very accurate estimate of the aggregate. In the rest of this section we formalize and expand on this idea.

4.1 Using outlier-indexes to approximate aggregation queries

Given an aggregation query Q which aggregates over column C of relation R, we describe a technique which uses an outlier-index for C along with a uniform sample of R to approximately answer Q. While we will give a precise definition of the outlier-index in the next sub-section, for the time being it is sufficient to assume that an outlier-index R_O is a sub-relation of the original relation R.

We view the relation R as being partitioned into two sub-relations R_O (outliers) and R_{NO} (non-outliers). The query Q can now be considered as the "union" of two sub-queries, the first of which is Q applied to R_O, while the second is Q applied to R_{NO}. This leads us to the following scheme for approximately answering an aggregation query for a given choice of the outliers R_O. To illustrate the scheme, we use the following example query:

Select sum(sales) from lineitem

[2] We use the word "index" here to simply indicate a distinct physical sub-relation. Such an index is best considered as a materialized view. Of course, a materialized view may be physically implemented as a heap or as a B+-tree index.

Preprocessing steps.

1. **Determine outliers** — *specify the sub-relation R_O of the relation R deemed to be the set of outliers.* For the example query above, we will create a view called *lineitem_outlier* which will be appropriately indexed.

2. **Sample non-outliers** — *select a uniform random sample T of the relation R_{NO}.* For the example query, we will obtain a uniform sample from the *lineitem* table (ensuring that no tuples that appear in *lineitem_outlier* are sampled) and materialize the sample in a table called *lineitem_samp*.

Query processing steps.

1. **Aggregate outliers** — *apply the query to the outliers in R_O accessed via the outlier-index.* For the example query, this corresponds to computing *sum(sales)* for the view *lineitem_outlier*.

2. **Aggregate non-outliers** — *Apply the query to the sample T and extrapolate to obtain an estimate of the query result for R_{NO}.* For the example query, this corresponds to computing *sum(sales)* for the table *lineitem_samp* and then multiplying the result by the inverse of the sampling fraction (extrapolation).

3. **Combine aggregates** — *combine the approximate result for R_{NO} with the exact result for R_O to obtain an approximate result for R.* For the example query, this means adding *sum(sales)* for the view *lineitem_outlier* and the extrapolated *sum(sales)* for *lineitem_samp*.

Since the database content changes over time, this requires selection of outlier indexes and samples to be refreshed appropriately. The samples need to be refreshed periodically as precomputed samples can become stale with use. An alternative is to do the sampling completely online, i.e., make it a part of query processing.

4.2 Selection of outliers

There are two points to be observed for our scheme: (1) the query error is solely due to the error in estimating the aggregate of the non-outliers from their sample, and (2) unlike sampling, in our scheme, there is an additional overhead of maintaining and accessing the outlier-index. Let τ be the memory (i.e., the number of tuples) allocated for the outlier-index. We would like to select the outlier set R_O so as to minimize the approximation error of our scheme for a class of queries subject to the constraint that R_O contains at most τ tuples from the relation R. The following definition identifies an optimal choice of the outlier set R_O. The reader should relate the error ϵ in the definition to the error introduced in Theorem 1.

Definition 1 *For any sub-relation $R' \subset R$, let $\epsilon(R')$ be the standard error in estimating the sum of the values in R' using uniform random sampling followed by extrapolation. An* optimal outlier-index *$R_O(R, C, \tau)$ for a column C in a relation R with a threshold τ is defined as a sub-relation $R_O \subset R$ such that*

- $|R_O| \leq \tau$, *and*
- $\epsilon(R \setminus R_O) = \min_{R' \subset R, |R'| \leq \tau} \{\epsilon(R \setminus R')\}$

Essentially, we have defined the outlier-index as an optimal sub-relation R_O that leads to the minimum possible sampling error, subject to the constraint that R_O has at most τ tuples in the relation R. Recall that the sampling error is the error in estimating the aggregate value over the tuples not included in R_O using the standard sample-and-extrapolate strategy. We know from Theorem 1 that the error is directly proportional to the standard deviation S. Let $S(R')$ denote the standard deviation for a sub-relation $R' \subset R$. Then, as per our definition, an outlier index R_O is a sub-relation of size at most τ such that the complement sub-relation $R \setminus R_O$ has minimum standard deviation $S(R \setminus R_O)$. The following theorem assists in choosing such a sub-relation efficiently.

Theorem 2 *Consider a multiset $R = \{y_1, y_2, \ldots, y_N\}$ where the y_i's are in sorted order. Let $R_O \subset R$ be the subset such that*

- $|R_O| \leq \tau$, *and*
- $S(R \setminus R_O) = min_{R' \subset R, |R'| \leq \tau} \{S(R \setminus R')\}$

Then, there exists some $0 \leq \tau' \leq \tau$ such that $R_O = \{y_i | 1 \leq i \leq \tau'\} \cup \{y_i | (N + \tau' + 1 - \tau) \leq i \leq N\}$

The theorem states that the subset that minimizes the standard deviation over the remaining set consists of the leftmost τ' elements (for some $0 \leq \tau' \leq \tau$) and the rightmost $\tau - \tau'$ elements from the multiset R, when the elements are arranged in a sorted order. Thus, the selection of an outlier-index reduces to determining the value τ'. This gives rise to the following algorithm for outlier-index selection.

Algorithm Outlier-Index(R, C, τ):

1. Read the values in column C of the relation R. Let $y_1, y_2, \ldots, y_N$ be the sorted order of the values appearing in C. Each value corresponds to a tuple.
2. For i = 1 to $\tau + 1$, compute
   $E(i) = S(\{y_i, y_{i+1}, \ldots, y_{N-\tau+i-1}\})$.
3. Let i' be the value of i where $E(i)$ takes its minimum value. Then the outlier-index is the tuples that correspond to the set of values $\{y_j | 1 \leq j \leq \tau'\} \cup \{y_j | (N+\tau'+1-\tau) \leq j \leq N\}$ where $\tau' = i' - 1$.

The efficiency of the algorithm crucially depends on the ability to compute standard deviations efficiently. It is well known that quantities such as sum, mean, variance, standard deviation can be efficiently maintained for a dynamic set of numbers subject to insertions and deletions. In particular, $E(i+1)$ can be computed from $E(i)$, y_i, and $y_{N-\tau+i}$ in $O(1)$ time.

We start the algorithm by performing the sort in Step 1. In Step 2, we first scan the sorted data to compute $E(1)$. We also make memory-resident copies of the first τ and the last τ values of $\{y_1, y_2, \ldots, y_N\}$. If they cannot fit into memory then we spool this subset of tuples W to disk. Since τ is usually small, the latter situation is unlikely. After this, we *do not* need access to data that is not in the set W. This is because $E(i+1)$ can be incrementally computed from $E(i)$ since the value to be deleted (y_i) and the value to be inserted ($y_{N-\tau+i}$) can be looked up from W. Thus, the running time of the algorithm is dominated by the sorting in Step 1 [3].

4.3 Discussion

Error guarantees. Like other sampling schemes, it is possible to give a per-query standard error (probabilistic) guarantee of our estimated answer. The standard error is estimated using Theorem 1. Estimating the standard error requires estimators for the count as well as the standard deviation of the non-outlier samples. The details of these computations are omitted due to lack of space.

Storage allocation. Thus far we have looked at the problem of allocating storage separately between outliers and samples. But, we can also ask the question: given sufficient space to store m tuples, how do we allocate storage between samples and outlier-index in order to minimize the error? This question is important when we use precomputed samples instead of online samples. Let $S(\tau)$ denote the standard deviation in the non-outliers for an optimal outlier-index of size τ. If we allocate the space such that we have τ tuples in the outlier-index and $m-\tau$ tuples in the sample, then the error as given by Equation 1 is proportional to $S(\tau)/\sqrt{m-\tau}$. Identifying an optimal allocation requires finding the value of τ for which $S(\tau)/\sqrt{m-\tau}$ is minimized. Due to lack of space, we omit details of our techniques that address such optimization.

Extensions to other aggregates. For the case of the *count* aggregate, outlier-indexing is not particularly beneficial since there is no variance among the data values. For the case of the aggregate average (*avg*), our preprocessing steps (i.e., selecting outliers and samples) does not change. During query processing, an *avg* query is estimated as *sum*/*count*. In general, our techniques extend to a class of aggregates that satisfy certain algebraic properties, the details of which are deferred due to lack of space.

Suppose we want to aggregate (*sum* or *avg*) $f(t)$, where f is a real valued function defined over the tuples, e.g., $sum(price * quantity)$. Then, we can use the function values instead of y_i's in our algorithm and can determine the sub-relation that minimizes the standard deviation over the remaining set. Of course, the number of real-valued functions f is potentially infinite and one cannot build an index for each one. As future work, we are investigating whether one can exploit *workload information* and build outlier-indexes only for the frequently occurring functions. In such cases it is also likely that the different outlier-indexes are correlated and as a result have a lot of tuples in common. Hence an optimization heuristic is to maintain the union of these indexes, instead of building a separate outlier-index for each.

Outlier-indexing does not appear to be useful for aggregates that depend on the *rank* order of the tuples rather than their actual values (i.e., aggregates such as *min*, *max*, and *median*). For a discussion of sampling-based techniques for computing order statistics, the reader should refer to the work by Manku, Rajagopalan, and Lindsay [17].

Extensions to foreign-key joins. Thus far, we have been only concerned with queries over a single relation. Consider foreign-key join queries involving a fact table R and a dimension table D, where the aggregation column is in R. Our techniques will work if the outlier-index and sample are computed over R, and these are joined with D at query processing time. This example generalizes to a wider class of queries in which multiple dimension tables are joined to the same fact table which contains the aggregate column. It is known that sampling-based methods (including ours) do not work well for more general join queries [5].

5 Handling low selectivity and small groups: Exploiting workload information

In this section, we examine the problem of low selectivity queries and small groups in group-by queries. Our approach to this problem is to use *weighted sampling* (instead of uniform sampling) by leveraging information about the *workload* while drawing the sample. The essential idea behind our weighted sampling scheme is to sample more from subsets of data that are small in size but are important, i.e., have high usage. Thus, our approach is based on the desire to exploit the fact that the usage of a database is typically characterized by considerable locality in the access pattern, i.e., queries against the database access certain parts of the

[3] For small values of τ, smallest and the largest τ values may be determined without requiring a complete sorting of the column.

data more than others. Therefore, by tuning the sample to a *representative* workload (i.e., set of queries) faced by the system, we can hope to answer queries posed to the database system more accurately.

The technique presented in this paper for weighted sampling is based on using precomputed samples. As part of our ongoing work, we are investigating how to leverage the idea of weighted sampling in the context of online sampling. As mentioned earlier, our technique of outlier-indexing presented in Section 4.2 can either use online or precomputed samples.

The rest of this section is organized as follows. First, we describe the framework for exploiting workload information. Second, we discuss the details of our weighted sampling scheme based on workload information.

5.1 Exploiting workload information

Our use of workload information for sampling and outlier-indexing involves the following steps:

1. *Workload Collection:* We obtain a workload consisting of representative queries gainst the database. Modern database systems provide tools to log queries posed against the server (e.g., the Profiler component of Microsoft SQL Sever).

2. *Trace Query Patterns:* The workload can be analyzed to obtain parsed information, e.g., the set of selection conditions that are posed.

3. *Trace Tuple Usage:* The execution of the workload reveals additional information on usage of specific tuples, e.g., frequency of access to each tuple, the number of queries in the workload for which it passes the selection condition of the query. Since tracking this information at the level of tuples can be expensive, it can be kept at coarser granularity, e.g., on page-level. Alternatively, the techniques presented in [8] such as batching of updates can be used to lower this overhead. For our experiments, we have assumed that a tuple t_i has weight w_i if the tuple t_i is required to answer w_i of the queries in the workload. These weights are subsequently normalized (See Section 5.2).

4. *Weighted Sampling:* Perform sampling by taking into account weights of tuples (from Step 3).

5.2 Details of Weighted sampling

In this section, we describe how we can leverage the weights derived from the tuple usage (see Step 3 above) to draw a weighted sample and how to use the weighted sample to answer an aggregation query.

Let the weight of the tuple t_i in the relation be w_i (see Step 3 above). Let the *normalized* weight be w_i' defined as $w_i / \sum_{j=1}^{N} w_j$. In our weighted sampling scheme, this tuple is accepted in the sample with probability $p_i = n \cdot w_i'$. Thus, while the expected sample size is n, the probability with which each tuple is accepted in the sample varies from tuple to tuple.

Given such a sample, we now address the question of how to answer aggregation queries approximately. With each tuple that is included in the sample, we store the probability p_i with which it was accepted in the sample. The inverse of this probability is the *multiplication factor* associated with the tuple used while answering the query. Each aggregate computed over this tuple gets multiplied by this multiplication factor. In the (degenerate) case of uniform sampling, since the probability is same for each tuple we do not have to store it and the multiplication factor is the same (N/n) for all tuples.

Weighted sampling works well if (1) the access pattern of queries is local (most of the queries access a small part of the relation) and (2) we have a workload which is a good representative of the actual queries which will be posed in the future.

Finally, note that just as we modified sampling by exploiting workload information, it is possible to *also tune the outlier-index* based on workload information. That is, we can modify algorithm **Outlier-Index(**R, C, τ**)** presented in Section 4.2 so that only outliers "relevant" to the queries in the workload are selected as part of the outlier-index. This is part of our ongoing work.

6 Implementation and experimental results

We have implemented the techniques described in this paper on Microsoft SQL Server 2000 and experimentally evaluated their effectiveness. The goals of the experiments were to compare the quality and performance of uniform sampling, weighted sampling, and weighted sampling + outlier-indexing. We begin by describing the implementation and the experimental setup. We then discuss our experimental results.

6.1 Implementation

Outlier-indexing. We leveraged the support of *materialized views* in Microsoft SQL Server 2000[4] to implement outlier-indexing. For example, if *l_extendedprice* is the column on which we want to construct an outlier-index, the following view returns all tuples in the *lineitem* table satisfying a certain predicate on (l_extendedprice):

[4]Materialized views are referred to as *indexed views* in SQL Server 2000.

CREATE VIEW l_extendedprice_otl_idx AS SELECT * from lineitem
WHERE (l_extendedprice $\leq$ 54819.46) OR (l_extendedprice $\geq$ 71442.88)

By materializing the above view we effectively index all tuples in *lineitem* satisfying the predicate (i.e., it is a partial index). The predicate in the view is determined using the algorithm described in Section 4 and depends on the storage allocated for the outlier-index. We also implemented a module that automatically rewrites a query to use the outlier-index and the sample rather than the fact table.

Uniform and weighted sampling. We modified Microsoft SQL Server to support uniform and weighted sampling. Specifically, we modified the execution tree generated by the SQL Server optimizer by adding a new operator as the root of the tree. For uniform sampling this operator simply accepts tuples with the specified probability (i.e., the sampling fraction) and stores the accepted tuples in a table. For weighted sampling, the probability of accepting a tuple is proportional to the weight associated with the tuple. In our experiments we calculated exact weights for each tuple for a given workload. We did this by including an additional column in the fact table (*lineitem*) to hold the weight of the tuple. It is also possible to maintain this column in a separate (*weights*) table. The trade-off is that while the cost of updating the weight associated with the tuple is reduced, the time to pick a weighted sample goes up since the *weights* table has to be "joined" in. We converted a SELECT query in the workload into the corresponding UPDATE statement that increments the weight of all the tuples satisfying the selection predicates. Finally, we implemented a module that automatically substitutes the sample table for the fact table of an incoming query.

6.2 Experimental setup

Platform. All experiments were run on a Dell Precision 610 system with a Pentium III Xeon 450 Mhz processor with 128 MB RAM and an external 23GB hard drive.

Databases. We used the well-known TPC-R benchmark for our experiments. One of the requirements of the benchmark however, is that the data is generated from a *uniform* distribution. Since we were interested in comparing the alternatives across different data distributions, we modified the TPC-R data generation program to generate data with varying degree of *skew*. The modified program generates data for each column in the schema from a Zipfian [20] distribution determined by the Zipfian parameter **z**[5]. For our experiments we generated 100MB TPC-R databases by varying **z** over values 1, 1.5, 2, 2.5, and 3. The ratio of the maximum value to the minimum value of the aggregation column varied between 76 and 106 for the different databases. The values in the aggregation column are chosen independently of their frequencies.

[5]We have made this program (which runs on x86/Windows NT platform) available for public download from [6].

Workloads. For our experiments on outlier-indexing, we generated several workloads over the TPC-R schema using a random query generation program. The program generates queries with (1) foreign key joins between tables, (2) aggregations on the fact table (*lineitem*), and optionally (3) grouping, and (4) selection. We used the *sum* aggregation function. Due to lack of space, we present results only for a few of the workloads that we experimented with.

Parameters. We varied the following parameters in our experiments: (1) skew of the data (**z**) was varied over 1, 1.5, 2, 2.5, and 3 (2) the sampling fraction (**f**) was varied over a wide range from 1% to 100%, and (3) the storage for the outlier-index was varied over 1%, 5%, 10%, and 20%. All numbers reported are the average over 3 runs.

Error metric. For each query in the workload we computed the *relative error* by dividing the difference between the approximate estimate for the aggregate (sum) and the accurate value of the aggregate by the latter. For queries with a group-by clause, we use the following error metric. Consider a query that has k groups in the answer obtained from actually executing the query. We build a k dimensional vector where the ith dimension contains the relative error in the aggregate expression for that ith group. The error is the mean of all the points in the vector. Thus, we report the average relative error over all groups. The error metric for a workload is average error over all queries in the workload.

6.3 Comparison of uniform sampling, weighted sampling and weighted sampling + outlier-indexing

We experimentally compared (1) uniform sampling (**USAMP**), (2) weighted sampling (**WSAMP**) and (3) weighted sampling + outlier-indexing (**WSAMP+OTLIDX**). Due to lack of space we only report results on the *quality* of these approaches, i.e., our experimental comparsions are focussed on the the accuracy of the estimates. Also, due to lack of space, we only report experiments for which the storage for outlier-indexing was 10% of the size of the data set. We plan to make additional experimental results available at *http://research.microsoft.com/dmx*.

To test the effectiveness of weighted sampling and outlier-indexing when the queries in the workload have low

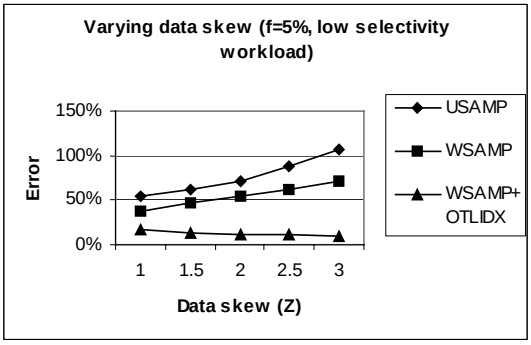

Figure 1. Error versus data skew

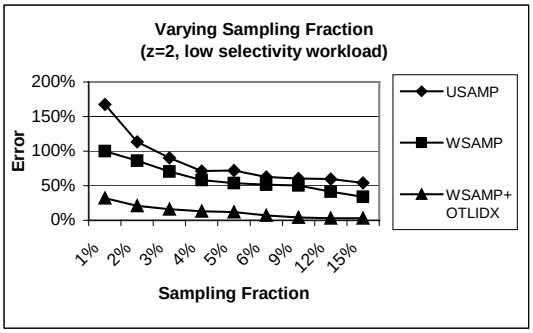

Figure 2. Error versus sampling fraction

selectivity, we generated a workload where each query consisted of a fact and dimension table, and all selection conditions on the dimension table were picked at random from a fixed range of the dimension table to simulate locality in workload. The "width" of the range was fixed at 20% of the size of the dimension table.

Varying the data skew. In this experiment, we vary the data skew, while keeping the sampling fraction constant (5%). Figure 1 shows that weighted sampling does consistently better than uniform sampling across all data skews, and that weighted sampling + outlier-indexing performs significantly better than weighted sampling alone.

Varying the sampling fraction. In our next experiment, we varied the sampling fraction while fixing the data skew (z=2). From Figure 2 we once again observe that weighted sampling gives lower error than uniform sampling across all sampling fractions. As expected, the greatest benefit occurs at low sampling fractions (e.g., 1%). Once again, use of outlier-indexing in addition to weighted sampling further improves accuracy significantly.

Varying the selectivity of queries. In our third experiment, we vary selectivity of queries between 1% and 100%.

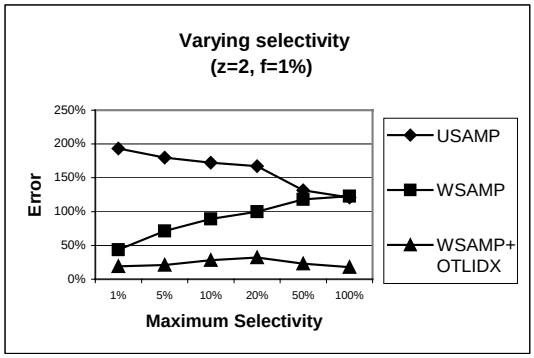

Figure 3. Error versus selectivity of queries

We fixed data skew (z=2) and the sampling fraction (f=1%). Figure 3 shows that at very low selectivity, weighted sampling noticeably alleviates the drawback of uniform sampling. However, as expected, when the workload references large portions of the data (e.g., at 100% selectivity) uniform and weighted sampling are not significantly different. Moreover, we see that weighted sampling + outlier-indexing performs consistently well across different selectivities, although as expected, its relative improvement over weighted sampling is smaller at lower selectivities.

6.4 Results on a real data set: MS Sales

We tested the effectiveness of outlier-indexing and weighted sampling on a real data set. We used an internal database in Microsoft called **MS Sales** which tracks the sales of products by the company over the fiscal year. We performed our analysis on a subset of about 1 GB of this data (fact + dimension tables). We used 25 representative queries involving aggregation, grouping, and selection which were picked from the log of queries executed against the database. We varied the sampling fraction over a range of values from 1% to 90% and compared uniform sampling and uniform sampling + outlier-indexing, where the size of the outlier-index was restricted to be no more than 10% of the data set. We found that compared to uniform sampling at 1%, uniform sampling + outlier-indexing significantly reduced the error (by 70% to 140%). We also found that using weighted sampling further reduced the error by about 20%. The maximum benefits of outlier-indexes appeared to be at the lowest sampling fraction (1%), although the same trends appeared to hold at higher sampling fractions as well.

7 Conclusion and future work

We explored some of the problems encountered when using uniform sampling as a means for approximate query answering. We observe that skew in the aggregation attribute

can lead to large errors. We proposed outlier-indexing to improve the accuracy in such cases. Our experiments demonstrated that our technique indeed improves accuracy significantly, at only a small additional cost. Another important issue examined in this paper is the problem of low selectivity of queries, and we outlined approaches based on workload information. Combination of outlier-indexing and weighted sampling based on workload information has proved to be a significant step forward.

We are currently investigating the problem of building a single outlier-index for different aggregates and aggregate expressions. As mentioned in this paper, tuning the selection of the outlier-index using the workload information is another interesting issue. We are also investigating extensions of our techniques to a wider class of join queries.

Acknowledgement. We are grateful to Venkatesh Ganti and the annonymous reviewers for their thoughtful comments on the draft.

References

[1] Acharya S., Gibbons P., and Poosala V. Congressional samples for approximate answering of group-by queries. In *Proceedings of the ACM SIGMOD Conference*, 487-498, 2000.

[2] Acharya S., Gibbons P., Poosala V., and Ramaswamy S. Join synopses for approximate query answering. In *Proceedings of the ACM SIGMOD Conference*, 1999.

[3] Barnett V. and Lewis T. *Outliers in Statistical Data*. John Wiley, 3rd edition, 1994.

[4] Chatfield C. *The Analysis of Time Series*. Chapman and Hall, 1984.

[5] Chaudhuri S., Motwani R., and Narasayya V. On random sampling over joins. In *Proceedings of the ACM SIGMOD Conference*, 1998.

[6] Chaudhuri S. and Narasayya V. *Program for TPC-D data generation with skew.* ftp.research.microsoft.com/pub/users/viveknar/tpcdskew.

[7] Cochran W. G. *Sampling Techniques*. John Wiley & Sons, New York, third edition, 1977.

[8] Ganti V., Lee M. L., and Ramakrishnan R. ICICLES: self-tuning samples for approximate query answering. In *Proceedings of 26th International Conference Very Large Data Bases*, 2000.

[9] Gibbons P., and Matias Y. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the ACM SIGMOD Conference*, 331-342, 1998.

[10] Haas P. and Hellerstein J. Ripple joins for online aggregation. In *Proceedings of the ACM SIGMOD Conference*, 287-298, 1999.

[11] Hawkins D. *Identification of Outliers*. Chapman and Hall, london, 1980.

[12] Hellerstein J., Haas P., and Wang H. Online aggregation. In *Proceedings of the ACM SIGMOD Conference*, 1997.

[13] Hou W., Ozsoyoglu G., and Dogdu E. Error-constrained COUNT query evaluation in relational databases. In *Proceedings of the ACM SIGMOD Conference*, 278-287, 1991.

[14] Jagdish H., Koudas N. and Muthukrishnan S. Mining deviants in times series database. In *Proceedings of 25th International Conference Very Large Data Bases*, 102-113, 1999.

[15] Knorr E. and Ng R. Algorithms for mining distance-based outliers in large datasets. In *Proceedings of 24th International Conference Very Large Data Bases*, 392-403, 1998.

[16] Ramaswamy S., Rastogi R., and Shim K. Efficient algorithms for mining outliers from large data sets. In *Proceedings of the ACM SIGMOD Conference*, 427-438, 2000.

[17] Manku G., Rajagopalan S., and Lindsay B. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *Proceedings of the ACM SIGMOD Conference*, 251-262, 1999.

[18] Olken F. *Random sampling from databases - bibliography.* http://pueblo.lbl.gov/ olken/mendel/sampling/bibliography.html.

[19] Seshadri P. and Swami A. Generalized partial indexes. In *Proceedings of the 11th International Conference on Data Engineering*, 420–427, 1995.

[20] Zipf G. E. *Human Behavior and the Principle of Least Effort.* Addison-Wesley Press, Inc, 1949.