An Experimental Study of Time-Constrained Aggregate Queries

Ying Hu[1] Wen-Chi Hou[2] Seema Sundara[1] Jagannathan Srinivasan[1]
[1] {ying.hu, seema.sundara, jagannathan.srinivasan}@oracle.com, [2] hou@cs.siu.edu

ABSTRACT
Although the notion of time-constrained query was first introduced two decades ago to address the problem of long running SQL queries, none of the commercial database systems support such a feature. This is rather surprising given the fact that database systems are beginning to accommodate large datasets in the order of terabytes to petabytes. Thus, the long running SQL query problem needs to be addressed. Recently, at Oracle we investigated and proposed a mechanism of supporting time-constrained queries to provide quick approximate answers by use of sampling for such long running SQL queries. This we followed up by coming up with error estimates as a measure of goodness for the approximation. To further validate our time-constrained query work, in this paper we present an experimental study conducted on our time-constrained query prototype built on the Oracle Database. It is our hope that this work will revive interest in time-constrained queries.

1. INTRODUCTION
The idea of time-constrained query was introduced in [11] two decades ago to address the problem of long running SQL queries. In the meantime, two developments have occurred that are worth examining. On one hand, databases have grown to terabytes and are now reaching petabytes (for example, Web Analytics Databases of Yahoo [17], data warehouses of eBay, Wal-Mart, etc. [18], and Oracle Exadata Storage Server [19]), making the problem of long running SQL queries even more significant. On the other hand, there have not really been any significant features in commercial database systems to address this problem. The onus is on users to limit the query execution time either by formulating a top-k query, or by using sampling on the tables involved. All these approaches are difficult in practice as translating the time requirement to corresponding result size limit (k) or sample sizes on tables involved is not an easy task. The database system is far better suited to perform such a translation.

Recently, we at Oracle have started to look at supporting the notion of time-constrained queries. The basic idea is that users should only be required to specify the time constraint and the database system should be able to do the rest, namely, augment the query to return top-k rows, or provide quick approximate answers by sampling. The latter approach of sampling is very promising, as there have been numerous papers published on this topic [1, 2, 4, 6, 9, 11, 12, 13, 14].

EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.

However, the shift from accurate to approximate answers immediately raises the question of how good those answers are. To address this, error estimates must be provided. In our recent work [13], we provided both point and interval estimators especially for the case of join queries involving aggregates that are processed using cross-product sampling. Even though the use of cross-product sampling is shown to have issues [2], we feel our approach of maximizing join result size warrants further study. In this paper, we present a series of experiments, which bring out the merits as well as shortcomings of this approach. Furthermore, these experiments validate the goodness of the estimators proposed in [13] where an experimental evaluation was omitted due to space limitations.

Our approach to the experimental study has been as follows. We focus on queries involving aggregates because those form a significant portion of the class of long running SQL queries. We study both single table queries and join queries involving aggregates (`SUM`, `COUNT`, `AVG`, and `MEDIAN`), which are processed either by sampling a single table or by cross-product sampling. The experiments are conducted on both synthetic dataset TPC-H [3] (generated with different degrees of skewness) and a real world dataset (Lahman Baseball Database [15]) to see the accuracy of our estimators. In addition, we present an interesting use case for progressively estimating the aggregates through multiple runs of time-constrained queries. Lastly, we present an experiment that demonstrates the trade-offs between the execution time and accuracy of query results.

We plan to continue this work and hope that the research community participates in advancing the state-of-the-art work in time-constrained queries so that such a feature becomes available in commercial database systems, at least for a useful subset of long running SQL queries.

Note that to deal with long-running queries, parallel database features, and more recently cloud computing using the MapReduce paradigm [20] have been explored. However these mechanisms are orthogonal to our time-constrained query mechanism, i.e., the time-constrained query mechanism can be used in conjunction with parallel query option to further reduce the query time. Also, parallel database features and the MapReduce paradigm only split a big computing job into smaller ones, which does not reduce the total computing resources (CPU or I/O) as is the case for the time-constrained query mechanism.

The rest of the paper is organized as follows. We briefly summarize previous work on time-constrained query (Section 2), and error estimation for aggregates (Section 3). Section 4 discusses the experiments, which is followed by a conclusion in Section 5.

2. TIME-CONSTRAINED QUERIES

In [12], time-constrained extensions were proposed to SQL that can specify limits on the time taken by a query. Based on specification in the SQL clause, a soft or hard time constraint with a time limit can be imposed on a query. Furthermore, the user can specify the acceptable nature of results (partial or approximate).

The basic approach as explained in detail in [12], is to look at the estimated time for query completion in the query execution plan, and then augment the original query with either the ROWNUM clause (for partial results) or the SAMPLE clause (for approximate results) to ensure that the query completes within the specified time constraint. The cardinality of the result set or the sample size is automatically determined iteratively by using a root-finding algorithm and maximizing either the number of returned rows or the number of rows in resultant joins.

For example, consider the following join query over the TPC-H 10 GB dataset that involves the SUM aggregate function:

```
SELECT SUM(CASE WHEN o_orderpriority =
  '1-URGENT' OR o_orderpriority = '2-HIGH'
  THEN 1 ELSE 0 END) AS cnt
FROM lineitem, orders
WHERE l_orderkey = o_orderkey AND
 l_receiptdate >= date '1994-01-01' AND
 l_receiptdate < date '1994-01-01' +
 interval '1' year;
```

Such queries are common in applications like OLAP and data warehouse. They tend to be long running as the aggregations are computed over large datasets. The actual run of this query could easily take more than 5 minutes. Instead of waiting for it to complete, users on data exploration tasks would probably prefer approximate but fast answers, say within one minute. Given this time constraint, the database system will determine which table(s) to sample and how much to sample in order to return an approximate result within one minute.

To maximize the number of rows in resultant joins, our time constraint mechanism relies on the following two heuristics: 1) for a nested loop join, the SAMPLE clause should be put into the outer relation, or the driver node; 2) for a normal hash join, when the time to process a sampled relation is equal to the time to process another sampled relation, the product of their sample sizes is maximal. Note that the second heuristic is also used for a sort-merge join. Based on these, our algorithm iteratively determines which table(s) to sample and the corresponding sample sizes.

3. ESTIMATION OF AGGREGATES

In this section, we briefly describe prior work of using cross-product sampling [6] to estimate SUM, COUNT, AVG, and MEDIAN, and then discuss how our time-constrained query mechanism can be used to progressively estimate aggregate values.

3.1 Estimated SUM, COUNT, AVG, and MEDIAN

The cross-product sampling scheme takes samples independently from different tables and then joins them. While its point estimates are straightforward, its interval estimates are fairly complex. For example, the AVG over the joins of samples can be simply taken as the estimated AVG. But its interval estimate has to be done in two steps: 1) its estimated variance is computed from some fairly complex formulas [13]; 2) the 100(1-α)% confidence interval is computed using a normal approximation, which is based on the finite-population Central Limit Theorem. The normal approximation is used for SUM, COUNT, and AVG when the sample size is large. For MEDIAN, we can use the MEDIAN obtained from the cross-product sample as an estimate for MEDIAN. To obtain a confidence interval for the estimated MEDIAN, we need to do the following: 1) get a new aggregate that counts the number of values that are less than the estimated MEDIAN; 2) estimate an approximate variance for the new aggregate (which is similar to COUNT); 3) from the estimated approximate variance and the normal approximation, i.e. the finite-population Central Limit Theorem, obtain both the lower bound and upper bound for MEDIAN with a given confidence level [13]. Note that this approach is different from the approach of sampling only in the final stage [16] in that the variance in the former case has to be computed using the techniques described above.

3.2 Progressive Estimation of Aggregates

Although progressive estimates are provided in current online aggregation systems [9, 5, 14], they rely on special join algorithms (such as ripple join algorithms) or engines (such as DBO query engine) to perform online aggregation. As an alternative, our time-constrained query mechanism can be used to progressively estimate aggregate values. The basic idea is to take advantage of the independence of multiple runs of the same time-constrained query. For example, suppose a long-running query involving SUM, COUNT, or AVG takes 10 minutes to finish. We repeatedly issue this query with a time constraint, say 5 seconds. This is the period at which the estimates are updated. For each run, we keep the estimated SUM, COUNT, or AVG, and their variance. We can use the average of the estimated SUM, COUNT, or AVG values as a new combined estimator. Because these runs are independent of each other, the estimated variance of the new estimator is 1/n times the average of the estimated variance from these runs, where n is the number of runs.

Assume $\hat{A}_i$ denotes the estimated SUM, COUNT, or AVG in the i-th run, and $\hat{A}*_n$ denotes the combined estimator after the n-th run. The new estimator, its variance, and the estimator of its variance are given by:

$$\begin{cases} \hat{A}*_n = (\hat{A}_1 + \hat{A}_2 + ... + \hat{A}_n)/n \\ V(\hat{A}*_n) = (V(\hat{A}_1) + V(\hat{A}_2) + ... + V(\hat{A}_n))/n^2 \\ \hat{V}(\hat{A}*_n) = (\hat{V}(\hat{A}_1) + \hat{V}(\hat{A}_2) + ... + \hat{V}(\hat{A}_n))/n^2. \end{cases}$$

We can also compute $\hat{A}*_n$ and $\hat{V}(\hat{A}*_n)$ recursively:

$$\begin{cases} \hat{A}*_n = ((n-1)\hat{A}*_{n-1} + \hat{A}_n)/n \\ \hat{V}(\hat{A}*_n) = ((n-1)^2\hat{V}(\hat{A}*_{n-1}) + \hat{V}(\hat{A}_n))/n^2. \end{cases}$$

The above arithmetic mean of A_i, i = 1, …, n, is an optimal combined estimator, when $V(\hat{A}_1) = V(\hat{A}_2) = ... = V(\hat{A}_n)$. As the time (or n) increases, the estimated running variance $\hat{V}(\hat{A}*_n)$ decreases in the order of 1/n, and the running confidence interval continues to narrow in the order of $\sqrt{1/n}$. When $V(\hat{A}_1)$, $V(\hat{A}_2)$, …, and

$V(\hat{A}_n)$ are different, the best combined estimator is given by a weighted mean.

Thus, we can provide estimated aggregates progressively by our time-constrained query mechanism easily, without any special query engine. The storage for intermediate results (i.e. $\hat{A}*_i$ and $\hat{V}(\hat{A}*_i)$ at the i-th run) is also very minimal. The effectiveness of our approach is demonstrated by experiments presented in Section 4.2. Similarly, this estimation technique can also be used in error-constrained queries [10, 7].

4. EXPERIMENTS

This section describes the experiments conducted using the prototype built on top of the Oracle Database. We run experimental queries with soft time constraints, that is, the queries are run against a small data set and expected to finish within the specified time limits. The time-constrained queries are translated into queries augmented with SAMPLE clauses that complete sooner due to the reduction in the amount of data blocks scanned and the sizes of intermediate results. However, the focus of these experiments is on the goodness of the estimated aggregate values derived from the approximate queries on both synthetic and real world datasets - TPC-H benchmark and Lahman Baseball Database [15]. In addition, to see how our estimators perform for skewed data, we conducted experiments by varying the degree of skewness of the TPC-H dataset [3]. Note that all TPC-H experiments are conducted under the Bernoulli block sampling scheme. However, since queries against the Lahman Baseball Database are not long running, Bernoulli row sampling is used in these experiments to get a relatively large sample. We also conducted experiments on the tradeoffs between the execution time and accuracy of query results, and then conclude this section with a discussion on several aspects of the experiments.

4.1 Estimating SUM in TPC-H

The biggest two tables (LINEITEM and ORDERS) in TPC-H benchmark are used to run the same join query discussed in Section 2. The scale factor is set to 1, which translates to a database size of about 1GB.

To approximate the result of the above query, we can push the sampling operation to the LINEITEM table only, since the LINEITEM table is the fact table for the TPC-H and has a foreign key reference to o_orderkey, which is the primary key of ORDERS [1]. We select a time constraint of 11s, which is about 16% of the estimated time for running the query to completion (70s). This gives us a sample size of 1% on the LINEITEM table, which constitutes the configuration (a) in our experiments. With the same time constraint and the goal to maximize f_1*f_2 or sample as many rows as possible in resultant joins, our time constraint mechanism [12] can also pick cross-product sampling with a sampling fraction of 10% for the LINEITEM and 40% for the ORDERS tables. This configuration corresponds to configuration (b) in our experiments. A hash join method is used in processing for both configurations.

We start with a uniformly distributed TPC-H database (z=0) and run the query 200 times against it under both configurations.

Figure 1 shows the results of the 200 runs with z=0 and configuration (a). The estimated SUM values are shown in blue, and their 95% upper/lower confidence limits, obtained by adding $\pm 2\sqrt{\hat{V}(\hat{Y})}$ (where $\sqrt{\hat{V}(\hat{Y})}$ is the estimated standard error of the estimated SUM – $\hat{Y}$) to the estimated SUM values, are in red and green, respectively. The actual SUM = 365666 is also shown as a straight line in the figure. In 189 of 200 runs (94.5%), the actual SUM = 365666 lies inside the 95% confidence interval, whereas in other 11 runs (5.5%), it lies outside the 95% confidence intervals. The average and standard deviation of the estimated SUM from the 200 runs are 364703 and 12353. The average of the estimated standard error from 200 runs is 12132, and the standard error is 12222.

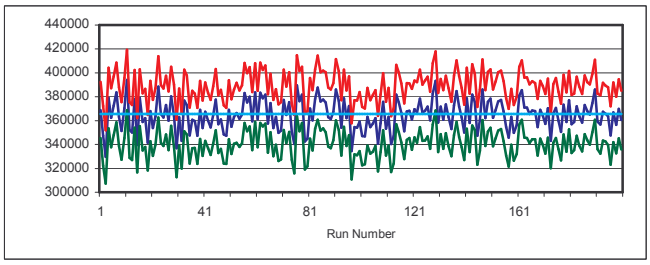


Figure 1. Estimated SUM and confidence interval in TPC-H (z=0 and configuration (a))

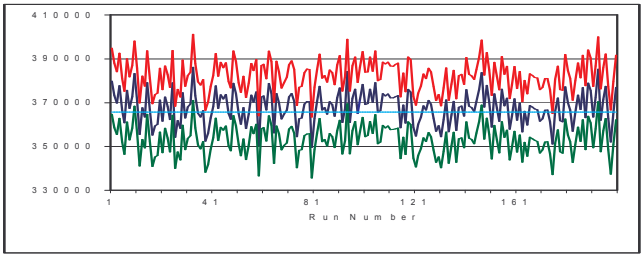


Figure 2. Estimated SUM and confidence interval in TPC-H (z=0 and configuration (b))

Figure 2 shows the results with z=0 and configuration (b). In 192 of 200 runs (96%), the actual value lies in the 95% confidence intervals, whereas in other 8 runs (4%), it lies outside the 95% confidence intervals. The average and standard deviation of the estimated SUM from the 200 runs are 367729 and 7170. The average of the estimated standard error from 200 runs is 7234, and the standard error is 7167. Comparing Figure 1 and Figure 2, we can see that the estimated standard error $\sqrt{\hat{V}(\hat{Y})}$ in configuration (b) is smaller than, or almost half of that in configuration (a). Thus maximizing f_1*f_2 can help reduce $\sqrt{\hat{V}(\hat{Y})}$. In addition, both experiments show that the estimated standard error is close to the standard error ($\sqrt{V(\hat{Y})}$), and they are also close to the standard deviation (STDDEV) of the estimated SUM from the 200 runs. All these results confirm the accuracy of these estimators.

We also ran the same query with the above two configurations against three skewed TPC-H databases (z=1, 2, and 4) 200 times. The results are similar to those shown in Figure 1 and Figure 2, and the statistics of these runs (including $\sqrt{\hat{V}(\hat{Y})} \doteq \sqrt{V(\hat{Y})} \doteq STDDEV$ of the estimated SUM from the 200 runs) are listed in Table 1.

We can see that although the data distribution becomes more skewed as z increases, the standard error in configuration (b) is almost half of that in configuration (a), which indicates that cross-

product sampling can do better than single table sampling under certain time constraints.

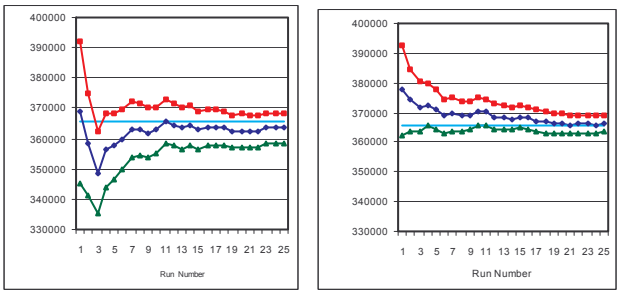
Table 1. Statistics for estimated SUM in TPC-H (z = 1, 2, and 4; configurations (a) and (b))[1]

	$z=1$ (a)	$z=1$ (b)	$z=2$ (a)	$z=2$ (b)	$z=4$ (a)	$z=4$ (b)
SUM	398264		109596		4499	
AVG ($\hat{Y}$)	399906	401848	110058	112469	5290	4616
STDDEV	24074	13871	16043	8669	3376	1651
$\sqrt{\hat{V}(\hat{Y})}$	24969	13732	16046	8782	3360	1772
$STDDEV_1$	1067	391	1346	441	1313	429
$\sqrt{V(\hat{Y})}$	25228	13485	16091	8438	3341	1738
% in CI	97%	95.5%	95%	94.5%	94%	94%

4.2 Estimating SUM Progressively

To return results progressively like in an online aggregation system, we combine the latest result with the previous results to calculate the combined aggregates and confidence intervals, as described in Section 3.2. Figure 3 shows the new combined results using the first 25 runs from the TPC-H database (z = 0), which are obtained from previous experiments under the same two configurations. The running confidence intervals continue to narrow and the estimated SUMs do not fluctuate as much as shown in Figures 1 and 2. Because the runs are on random samples, the estimated SUMs also converge randomly from different signs.

For z = 1, 2, 4, we get similar figures. Table 2 lists the estimated running standard errors $\sqrt{\hat{V}(\hat{Y}^*)}$ after 1st, 5th, 10th and 25th runs. For example, the estimated running standard errors of the 25th run are only about 1/5 of the estimated running standard errors of the 1st run except the two cases under z=4, where the STDDEV of estimated standard errors $\sqrt{\hat{V}(\hat{Y})}$ becomes relatively big, as seen in the last two columns of Table 1.



(a) (b)

Figure 3. Estimated running SUM and running confidence interval in TPC-H (z = 0, configurations (a) and (b))

[1] In the first column of Table 1, SUM: the actual SUM; AVG ($\hat{Y}$): the average of 200 estimated SUMs; STDDEV: the STDDEV of 200 estimated SUMs; $\sqrt{\hat{V}(\hat{Y})}$: the average of 200 estimated standard errors; $STDDEV_1$: the STDDEV of 200 estimated standard errors; $\sqrt{V(\hat{Y})}$: the standard error; % in CI: the percentage of the actual SUM lying in the 95% confidence interval.

Table 2. Estimated running standard error for estimated SUM in TPC-H (z = 1, 2, and 4; and configurations (a) and (b))

	$z=1$ (a)	$z=1$ (b)	$z=2$ (a)	$z=2$ (b)	$z=4$ (a)	$z=4$ (b)
1st	25185	14315	17027	9094	3064	2237
5th	11164	6254	7604	3958	1259	790
10th	8060	4377	5248	2832	1105	567
25th	4992	2754	3196	1768	705	383

4.3 Estimating AVG in the Lahman Baseball Database

The following query is executed, to get the average number of hits by a Hall of Fame player in a stint:

```
SELECT AVG(h) AS R
FROM master ma, batting ba
WHERE ma.playerID = ba.playerID AND
  ma.hofID IS NOT NULL;
```

The above query would normally finish in less than 1 second, which is the minimal time unit. We run the experiments with the following two configurations: in configuration (a), only the BATTING table is sampled at 1%; in configuration (b), the BATTING table is sampled at 10% and the MASTER table is sampled at 40%. Queries in both configurations use a hash join method and are completed in about 50% of the execution time (0.33s) for the original query.

Table 3 shows the statistics from 200 experiments. It is clear that the variances in configuration (b) are smaller than those in configuration (a). Like in Section 4.2, AVG can also be estimated progressively by combining the latest result and its previous results, as described in Section 3.2.

Table 3. Statistics for estimated AVG in the Lahman Baseball Database

R = 75.6081061	Configuration (a)	Configuration (b)
AVG ($\hat{R}$)	76.03	75.44
STDDEV ($\hat{R}$)	5.903	3.452
AVG ($\sqrt{\hat{V}(\hat{R})}$)	5.827	3.603
STDDEV($\sqrt{\hat{V}(\hat{R})}$)	0.3411	0.1340
$\sqrt{V(\hat{R})}$	5.838	3.482
% in confidence interval	95.5%	97%

4.4 Estimating MEDIAN

The following query is run against 1G TPC-H databases:

```
SELECT MEDIAN(l_extendedprice)
FROM lineitem, orders
WHERE l_orderkey = o_orderkey AND
  l_receiptdate >= date '1994-01-01' AND
  l_receiptdate < date '1994-01-01' +
  interval '1' year AND
  (o_orderpriority = '1-URGENT' OR
  o_orderpriority = '2-HIGH' );
```

The query run against the Lahman Baseball Database is the same query as in Section 4.3 with AVG being replaced by MEDIAN. Because the TPC-H z = 2, 4 databases either have so many duplicates that $M = \hat{M} = y_{(l)} = y_{(u)}$ (where M, $\hat{M}$, $y_{(l)}$, and $y_{(u)}$ are the actual MEDIAN, the estimated MEDIAN, the lower confidence limit and the upper confidence limit of MEDIAN, respectively), or have so few returned rows that 95% confidence interval cannot be obtained from the sample data (for example, $l <$

1), we list only results from the TPC-H databases (z = 0, 1), as well as those from the Lahman Baseball Databases (LBD) in Table 4. The same configurations (a) and (b) are used.

The 95% confidence intervals for MEDIAN are more conservative because of higher percentages (98-100%) of actual MEDIAN lying in them. This is because several approximations are used to obtain them.

Table 4. Statistics for estimated MEDIAN[2]

	z = 0 (a)	z = 0 (b)	z = 1 (a)	z = 1 (b)	LBD (a)	LBD (b)
MEDIAN	36731		34585		59	
AVG	36754	36712	34890	34723	61.6	58.7
STDDEV	591	115	2539	1663	17.1	11.7
AVG_1	5352	3033	12511	7549	77.5	58.7
$STDDEV_1$	355	66	2369	977	11.5	6.1
% in CI	100%	100%	99.5%	99%	98%	99.5%

4.5 Tradeoffs between Time and Accuracy

Both Q6 and Q5 in TPC-H are used to run against a 10G TPC-H database (z = 0). Q6 is a single table query to forecast revenue change, and Q5 is a 6-table-join query to list the revenue volume done through local suppliers. Q5 returns 5 rows (each for one Asian country), and here we just show the first resulting row (i.e. the row for INDONESIA) since the other four rows have similar results. Figure 4 shows that the estimated standard error becomes smaller as more time is spent on Q6, and Figure 5 shows the same results for Q5.

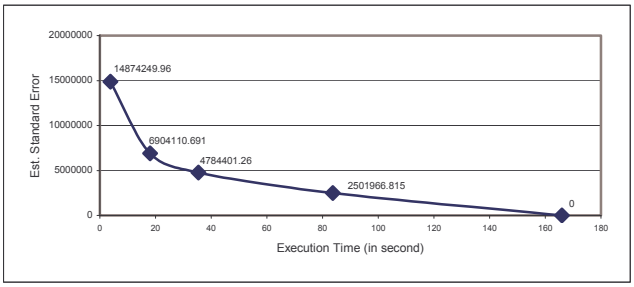


Figure 4. Estimated standard error vs actual execution time for Q6 in TPC-H (z = 0, 10G)

Note that although both queries can be completed in about 3 minutes, it takes only 4 seconds for Q6 to have its 95% confidence interval: ±2*14874250, or ±2.4% relative error of the actual value 1231283271, while it takes more than 70 seconds (but still 37.7% of the original completion time) for Q5 to have its 95% confidence interval: ±2*13072664.5, or ± 4.9% relative error of the actual value 530355705. The main reason is that there are more elements selected in Q6 than Q5 to compute the aggregates; more specifically, 1140640 rows vs. 14571 rows from about 60 million rows in the LINEITEM table meet the selection criteria of Q6 and Q5, respectively. The confidence intervals in time-constrained queries can help users to understand whether the approximate results are already good enough even for a short period of time of processing, whereas other queries may need more time to complete if the accuracy of their results is important.

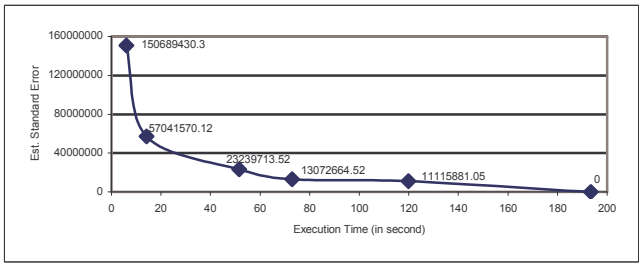


Figure 5. Estimated standard error vs. actual execution time for Q5 in TPC-H (z = 0, 10G)

4.6 Discussion

Although queries against the Lahman Baseball Database are not long running, (for instance, the query in Section 4.3 is completed in 0.33 seconds and the queries augmented with SAMPLE clauses are completed in about half of the time,) these results were presented mainly to confirm the goodness of our estimators on a real world dataset.

Also, the standard error (or variance) computation can be done with minimal overhead by leveraging the original aggregate query processing. For the example of configuration (b) in Section 4.3, the standard error calculation for average number of hits by a hall of fame player in a stint can be conceptually expressed as an aggregate SQL query as follows:

```
WITH
 q0 AS (SELECT h, ba.ROWID AS rwd,
               ma.playerID AS playerID
        FROM master SAMPLE(40) ma,
             batting SAMPLE(10) ba
        WHERE ma.playerID = ba.playerID AND
              ma.hofID IS NOT NULL),
 q1 AS (SELECT AVG(h) AS avg_h, COUNT(h) AS cnt_h
        FROM q0),
 q2 AS (SELECT (SUM(h) - (SELECT avg_h FROM
          q1)*COUNT(h)) AS cnt, rwd, playerID
        FROM q0 GROUP by playerID, rwd),
 q3 AS (SELECT SUM(cnt) AS x0, SUM(cnt*cnt) AS x1
        FROM q2),
 q4 AS (SELECT SUM(cnt1*cnt1) AS x2 FROM
         (SELECT SUM(cnt) AS cnt1
          FROM q2 GROUP BY rwd)),
 q5 AS (SELECT SUM(cnt1*cnt1) AS x3 FROM
         (SELECT SUM(cnt) AS cnt1
          FROM q2 GROUP BY playerID))
SELECT q1.avg_h AS est_avg_h, SQRT(q4.x2*(1-0.1)
 +q5.x3*(1-0.4)-q3.x1*(1-0.1)*(1-0.4))/q1.cnt_h
 AS est_std_error
FROM q1, q3, q4, q5
```

This computation is similar to the data cube computation, which can be optimized by algorithms described in [8]. A key aspect in the data cube computation is the size of the cube, which theoretically would be n_1*…*n_k, where n_1, …, n_k are the sample sizes for each of the joined tables. But in practice, this is much smaller owing to join selectivity and presence of filter predicates (if any) on the participating tables. For example, for the above case, the theoretical size is about 89945*17022*0.1*0.4 or about 60 Million cells, whereas in actuality the size of resulting cells is

[2] In the first column of Table 4, MEDIAN: the actual MEDIAN; AVG: the average of 200 estimated MEDIANs; STDDEV: the STDDEV of 200 estimated MEDIANs; AVG_1: the average of 200 estimated $(y_{(u)} - y_{(l)})$ values; $STDDEV_1$: the STDDEV of 200 estimated $(y_{(u)} - y_{(l)})$ values; % in CI: the percentage of the actual MEDIAN lying in the 95% confidence interval.

about 500. Also, for large tables (such as tables in the TPC-H dataset) our time-constrained query mechanism uses block sampling, so cardinality of each dimension, namely sample size, would be calculated in terms of number of blocks (as opposed to number of rows), which would usually be much smaller.

Furthermore, note that the estimator derived using Bernoulli sampling gives much tighter bounds for the standard error (in the above example, `SQRT(q4.x2*0.9 + q5.x3*0.6 - q3.x1*0.54)/q1.cnt_hr`) when compared to approximating the standard error using sampling with replacement (`SQRT(q4.x2 + q5.x3)/q1.cnt_hr`). For the above example, the average of 200 estimated standard errors returned by the latter is 5.014, or about 44% (= (5.014-3.482)/3.482) higher than the standard error (3.482 as per Table 3). In contrast, the average of estimated standard errors returned by the former is 3.603 or only 3.5% higher.

For the TPC-H dataset, we see that the execution time is reduced significantly by adding SAMPLE clauses. For instances, the original query (i.e. without SAMPLE clause) in Section 4.1 (z=0) is estimated to finish in 70 seconds, and is normally completed in less than 65 seconds. The approximate queries in configurations (a) and (b) are estimated to finish in 11 seconds, and are normally completed in less than 7 seconds. For z=1, 2, and 4, we see similar results because the major part of execution time is spent in scanning the two tables. In summary, although there are differences between actual execution time and estimated execution time, we do observe that sampling speeds up the queries as was also shown in [12].

Note that under configuration (a) in TPC-H dataset, the query result corresponds to the result of a uniform sample from the original join, because only LINEITEM (a fact table) is sampled and it has a foreign key reference to o_orderkey, which is the primary key of ORDERS (dimension table) [1]. Configuration (b) is obtained to maximize f_1*f_2, or sample as many rows as possible in resultant joins. We find that when a fact table after sampling becomes much smaller than a dimension table, using cross-product sampling to maximize f_1*f_2 can be better than the strategy of uniform sampling only on the fact table, because the variance of its estimated aggregates can be smaller, as shown in our experiments. For instance, 1% LINEITEM is much smaller than ORDERS. When a fact table after sampling is still bigger than a dimension table, the approach of maximizing f_1*f_2 can result in the same configuration as the approach of sampling only on a fact table. Thus, when the knowledge of the variances is absent without trials, the objective of achieving a maximal f_1*f_2 (or sampling for as many rows as possible in resultant joins) is justified in practice.

5. CONCLUSION

The paper considered time-constraint aggregate queries where time reduction is obtained by the use of sampling. The experimental study conducted in our time-constrained query prototype, built on top of the Oracle Database, demonstrates the effectiveness of the proposed estimation techniques.

6. REFERENCES

[1] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy, "Join Synopses for Approximate Query Answering," *SIGMOD* 1999, pp. 275-286.

[2] S. Chaudhuri, R. Motwani, V. R. Narasayya, "On Random Sampling over Joins," *SIGMOD* 1999, pp. 263-274.

[3] S. Chaudhuri, V. R. Narasayya, "Program for TPC-D Data Generation with Skew," [Online]. Available: ftp://ftp.research.microsoft.com/users/viveknar/tpcdskew

[4] P. J. Haas, "Large-Sample and Deterministic Confidence Intervals for Online Aggregation," *SSDBM* 1997, pp. 51-63.

[5] P. J. Haas, J. M. Hellerstein, "Ripple Joins for Online Aggregation," *SIGMOD* 1999, pp. 287-298.

[6] P. J. Haas, J. F. Naughton, S. Seshadri, A. N. Swami, "Selectivity and Cost Estimation for Joins Based on Random Sampling," *J. Comput. Syst. Sci.* 52(3), pp. 550-569, 1996.

[7] P. J. Haas, A. N. Swami, "Sequential Sampling Procedures for Query Size Estimation," *SIGMOD* 1992, pp. 341-350.

[8] J. Han, M. Kamber, *Data Mining: Concepts and Techniques,* Morgan Kaufmann, San Francisco, CA, 2006.

[9] J. M. Hellerstein, P. J. Haas, H. J. Wang, "Online Aggregation," *SIGMOD* 1997, pp. 171-182.

[10] W.-C. Hou, G. Özsoyoglu, E. Dogdu, "Error-Constraint COUNT Query Evaluation in Relational Databases," *SIGMOD* 1991, pp. 278-287.

[11] W.-C. Hou, G. Özsoyoglu, B. K. Taneja, "Processing Aggregate Relational Queries with Hard Time Constraints," *SIGMOD* 1989, pp. 68-77.

[12] Y. Hu, S. Sundara, J. Srinivasan, "Supporting Time-Constrained SQL Queries in Oracle," *VLDB* 2007, pp. 1207-1218.

[13] Y. Hu, S. Sundara, J.Srinivasan, "Estimating Aggregates in Time-Constrained Approximate Queries in Oracle," *EDBT* 2009, pp. 1104-1107

[14] C. M. Jermaine, S. Arumugam, A. Pol, A. Dobra, "Scalable Approximate Query Processing with the DBO Engine," *SIGMOD* 2007, pp. 725-736.

[15] S. Lahman, *The Lahman Baseball Database, Version 5.5.* [Online]. Available: http://www.baseball1.com/, Dec. 11, 2007.

[16] G. S. Manku, S. Rajagopalan, B. G. Lindsay, "Approximate Medians and other Quantiles in One Pass and with Limited Memory," *SIGMOD* 1998, pp. 426-435.

[17] http://www.informationweek.com/news/software/database/showArticle.jhtml?articleID=207801436

[18] http://www.computerworld.com/s/article/9117159/Teradata_creates_elite_club_for_petabyte_plus_data_warehouse_customers

[19] http://www.oracle.com/database/exadata.html

[20] J. Dean, S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *OSDI* 2004, pp.137-150.