Supporting Time-Constrained SQL Queries in Oracle

Ying Hu Seema Sundara Jagannathan Srinivasan
Oracle
One Oracle Drive
Nashua, NH 03062, USA
{ying.hu, seema.sundara, jagannathan.srinivasan}@oracle.com

ABSTRACT

The growing nature of databases, and the flexibility inherent in the SQL query language that allows arbitrarily complex formulations, can result in queries that take inordinate amount of time to complete. To mitigate this problem, strategies that are optimized to return the 'first-few rows' or 'top-k rows' (in case of sorted results) are usually employed. However, both these strategies can lead to unpredictable query processing times. Thus, in this paper we propose supporting time-constrained SQL queries. Specifically, a user issues a SQL query as before but additionally provides nature of constraint (soft or hard), an upper bound for query processing time, and acceptable nature of results (partial or approximate). The DBMS takes the criteria (constraint type, time limit, quality of result) into account in generating the query execution plan, which is expected (guaranteed) to complete in the allocated time for soft (hard) time constraint. If partial results are acceptable then the technique of reducing result set cardinality (i.e. returning first few or top-k rows) is used, whereas if approximate results are acceptable then sampling is used, to compute query results within the specified time limit. For the latter case, we argue that trading off quality of results for predictable response time is quite useful. However, for this case, we provide additional aggregate functions to estimate the aggregate values and to compute the associated confidence interval. This paper presents the notion of time-constrained SQL queries, discusses the challenges in supporting such a construct, describes a framework for supporting such queries, and outlines its implementation in Oracle Database by exploiting Oracle's cost-based optimizer and extensibility capabilities.

1. INTRODUCTION

The prolific growth in the amount of information being generated has lead to larger and larger databases. Most commercial databases (including IBM DB2, Microsoft SQL Server, Oracle, MySQL, PostgreSQL) have deployed databases ranging anywhere from Gigabytes to Terabytes (and now even to Petabytes).

Furthermore, use of SQL as the standard query language allows for formulation of arbitrarily complex queries involving joins of many tables, grouping and sorting of results, and the use of expensive user-defined functions as filtering predicates.

The above two trends have introduced the problem of long running SQL queries. This is further worsened by unpredictable query response times. For example, a query which has worked quite well in the past, can take seconds, minutes or even hours to complete when executed with an input value that makes the filter condition unselective.

Prior research has tried to address the problem of long running queries and unpredictable query response time using the following approaches:

- *Optimize for first-few rows*: Users can indicate (for example, by using a hint in Oracle) that they want to optimize the query to return the first few rows as soon as possible.
- *Optimize for top-k rows*: This is another variant of 'first-few rows' optimization, where the user is interested in only the top-k candidates (specifiable, for example, using a ROWNUM clause in Oracle) of the sorted results.
- *Compute approximate results*: Unlike the first two approaches which return accurate results, this approach speeds up query processing by working only on portions of data and hence, returns approximate results. Since the results are approximate, errors (in terms of confidence interval) are usually estimated to give a measure of goodness of results.

These approaches try to address the problem of long running SQL queries by computing either partial or approximate results quickly. However, when a user is constrained for time, the onus of employing these approaches intelligently is left to the user. Given the sophistication of cost-based optimizers that are now common in commercial database systems, it is not an easy task for a user to translate a time constraint to an appropriate top-k or approximate query.

Thus, the paper argues that database systems should support time-constrained SQL queries and examines supporting such a notion in Oracle Database. The key idea is to support an additional clause as part of SQL query specifying *constraint type* (*soft* or *hard*), *time limit (time in seconds)*, and *acceptable nature of results* (*partia*l or *approximate*). For soft (hard) time constraint, the DBMS generates a query execution plan, which is expected (guaranteed) to complete in the allocated time. The acceptable nature of results determines the technique employed by the DBMS, namely, reducing the result set cardinality when partial results are acceptable or using sampling when approximate results are acceptable. For approximate results, we additionally provide two types of aggregate functions, namely, i) functions that estimate the aggregate values for the complete table data even though only a portion of the table is used, and ii) functions that compute

VLDB '07, September 23-28, 2007, Vienna, Austria.

confidence intervals quantifying the uncertainty in the corresponding estimated aggregate values.

Time-constrained SQL queries would be useful both in traditional and emerging database applications:

- *Decision Support Systems (DSS)*: In traditional DSS environments, users have complex aggregate queries with long processing time. Often, the user is not interested in full query results, but only wants partial results to spot trends (e.g. growth rate of sales in different regions). There has been a lot of discussion around this in the context of approximate and top-k queries. By using a time-constrained query, a user can obtain results within a certain time. The results can be complete or approximate as long as they meet the time constraints.
- *Mobile and Sensor Network Applications*: In some mobile applications, the user is interested in complex select queries with top-k results, which are often bound by time. Consider a user who expects to pass a certain location in the next 2 minutes, and is interested in finding out if there are any Starbucks near that location. In this case, any results returned after 2 minutes are useless to the user. In such a case, the user query can be easily modeled with a time constraint. In RFID applications, sensors generate voluminous data tracking objects in containers (shelf, shipment package, etc.) that are part of supply chain. Time-constrained queries can be used to generate approximate statistics regarding objects detected by RFID sensors in a timely manner.
- *Data Center Service Level Management:* Consider a data center that employs a Service Level Agreement (SLA) to ensure quality and codify customer expectations. Such a system may want to flag queries which take an inordinately long amount of time, indicating possible problems with resource availability or poor execution plans, or simply due to use of an unselective filter condition. The system can impose a time constraint on all such queries.

We have prototyped the support for soft time-constrained SQL queries in Oracle Database, using the cost-based optimizer features (EXPLAIN PLAN and system and object level statistics) and the extensibility features of Oracle (table functions). We conducted experiments using a 10GB sized TPC-H data set [26], which validate the feasibility of our approach (see Section 7 for more details).

The key contributions of this paper are as follows:

- A framework for supporting time-constrained SQL queries leveraging features available in a commercial database system,
- An algorithm for translating time constraints on a multi-table query that does not involve joins on foreign keys by choosing the tables to which the sample clause should be added and determining the corresponding sample sizes,
- A scheme for supporting hard time constraints that puts hard limits on the top-level query as well as on each of its blocking operations, and
- Experimental evaluation validating the feasibility of the approach.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces the terminology and the key concept of time-constrained SQL queries. Section 4 discusses relevant features of Oracle Database. Sections 5 and 6 cover the scheme for supporting soft and hard time constraints respectively. Section 7 presents results of an experimental study conducted using a TPC-H Benchmark data set, which is followed by a discussion in Section 8. Section 9 concludes the paper and outlines future work.

2. Related Work

Real-time database systems have been a topic of research for over a decade now. However, as pointed out in [28, 29], the focus has been on transaction processing issues encompassing areas of resource scheduling, concurrency control, and memory management. Also, the bulk of the work in this area is in relation to main-memory based systems.

In contrast, our work focuses on extending a traditional database system with time constraints on long running SQL queries. It leverages research done in the areas of top-k query processing, approximate queries, and error estimations. Here we briefly discuss the relevant work in each of these areas.

The need for a SQL extension to explicitly limit the cardinality of a query result was proposed in [3]. The basic strategy of limiting data that need to be examined by augmenting the query with a range predicate is discussed in [5, 23]. In [4], authors propose a partitioning-based approach to efficiently process top-k queries. For queries involving joins, [24] proposes generating join results ordered on a user-defined scoring function. In general, the need for rank-aware query optimization and possible approaches to supporting it is discussed in [25]. Note that most commercial database systems allow specifying top-k query and its optimization. For example, Oracle Database supports the ROWNUM clause [19], which can be used to specify top-k results, and Oracle cost-based optimizer takes that into account in generating an optimal query execution plan [18].

Approximate query processing focuses on returning approximate results by employing sampling techniques [8, 9, 10]. Online aggregation is another area where approximate query processing techniques are used [6, 7]. The work in this area also includes estimating errors in the reported results, for example, computing confidence interval values for common aggregates such as SUM, AVG, and COUNT [6]. The ability to restrict rows from a table to a sample is supported in most commercial database systems including Microsoft SQL Server [15], IBM DB2 [16], and Oracle [19].

Another somewhat related area of work is progress monitors [11, 12, 13, 14] for long running SQL queries. They try to address the problem of long and unpredictable query processing time by providing a progress bar, which indicates the percentage of query execution completed so far.

Time-constrained query processing is considered in detail in the context of CASE-DB prototype database system [1, 2]. For queries involving non-aggregates, this work assumes partitioning of a relation into a collection of fragments, where each fragment contains semantically related rows. For example, one fragment may contain data that is more recent and hence must be processed first. They propose a scheme of implementing various relational operations by transforming the original query to work over

corresponding table fragments. For queries involving aggregates, a sampling technique is employed. For queries involving multiple tables, the problem is simplified by assuming equal sample sizes from all the participating tables. The work also provides statistical estimators when sampling is employed.

Our work of supporting time-constrained SQL queries in Oracle Database differs from CASE-DB in the following ways: 1) We transform the query and rely on Oracle's cost-based optimizer to estimate the query processing time. 2) For handling queries on non-aggregates, our scheme does not require the user to specify fragments for a relation. 3) For queries involving multi-table joins, our scheme is flexible in that it can identify one or more tables to which the sample clause is added. Furthermore, the sample size for different tables could be different. 4) Our scheme for supporting hard time constraints introduces timers for each blocking operation.

3. KEY CONCEPTS

This section gives the terminology, and describes time-constrained SQL queries and related concepts.

3.1 Terminology

Often, the end user has time constraints and wants the query results within certain time bounds. We introduce the concept of *soft time constraints* where the end user only gives directives on the desired time for the query to return results, and *hard time constraints*, where the query is guaranteed to return results within the specified time.

For a time-constrained query, the query results can be *approximate* or *partial*. The query can *sample* a portion of the data that is queried and return approximate results based on the sample. Alternately, it can choose to return only a partial subset of the results (first few rows or top K rows in case of sorted result) with the guarantee that they are *accurate*.

3.2 Time-Constrained SQL Queries

The syntax of SQL's `SELECT` statement can be extended by adding a `TIME CONSTRAINT (T)` clause (where `T` indicates time in seconds) and additional clauses to specify the result set characteristics (approximate or partial). The resulting query syntax is as follows:

```
SELECT … FROM … WHERE …
GROUP BY … HAVING …
ORDER BY …
[[SOFT|HARD] TIME CONSTRAINT (T)
 [WITH {APPROXIMATE|PARTIAL} RESULT]];
```

If neither `SOFT` nor `HARD` is specified, by default soft time constraint will be assumed. If the `WITH ... RESULT` clause is not specified then the system computes approximate results by default. Although `T` in the `TIME CONSTRAINT (T)` clause can be replaced by an SQL `<value expression>`, which is not correlated with the rest of the query, this paper only focuses on the case of `T` as an integer. In addition, this paper does not address the cases of putting the `TIME CONSTRAINT` clause into sub-queries, DML statements and even DDL statements because the semantics may be unclear. However, the semantics of the `TIME CONSTRAINT` clause in a top-level SQL query is clear, namely, return the results (partial or approximate) for the top SQL query as per the specification, in the specified time limit.

3.3 The Basic Approach

For supporting soft time constraints, the input query is transformed by either

- augmenting it with a ROWNUM clause that reduces the result set size, or
- augmenting it with a SAMPLE clause that reduces the data blocks scanned and the intermediate result size returned from the referenced table(s)

When the user asks for partial results, the query is augmented with the ROWNUM clause. When the user asks for approximate results, the query is transformed by augmenting it with the SAMPLE clause. The transformed query is expected to finish sooner. For example, consider the following time-constrained SQL query where the user is interested in '`APPROXIMATE`' results:

```
Q1: SELECT AVG(salary)
    FROM employees
    SOFT TIME CONSTRAINT (50)
    WITH APPROXIMATE RESULT;
```

The user indicates that the specified query should be completed with approximate results in 50 seconds. The above query after rewrite may be transformed into

```
Q1-T: SELECT AVG(salary)
    FROM employees SAMPLE BLOCK (10);
```

The sample clause specifies the sample size (in percentage). Thus, in the above transformed query, only about 10%[1] of table blocks are accessed in computing the average, which reduces the overall query processing time. Thus, augmenting the SAMPLE BLOCK clause to the query is expected to reduce the time needed to complete the query. The same happens for the case of augmenting the ROWNUM filter condition in the top-k or partial result query.

The challenge is in correctly estimating the result set cardinality in the case of partial results or estimating the sample size in the case of approximate results, which can lead to timely completion of the query.

3.4 Soft Time-Constrained Query Definition and Processing

Definition*: A query Q with a soft time constraint of T seconds* $\Rightarrow t_{estimated\_by\_optimizer}(Q')$ *BETWEEN T-d AND T, where d is a small time unit and Q' is the transformed query.*

That is, the transformed query should have an estimated time within the specified time limit.

To estimate the result set size in the case of partial results, we can proceed as follows. Let the function $f_Q(r)$, which represents that the time to execute query Q depends on result set size r. Thus, $f_Q(r) = T$, where T is the specified time constraint. The desired r is a root of equation $f_Q(r) - T = 0$ and is obtained using a root finding algorithm [20].

Similarly, estimating the sample size for a single table query is straightforward. Let the function $f_Q(s)$, which represents that the time to execute query Q depends on sample size s. Thus, $f_Q(s) =$

[1] (10%: This percentage indicates the probability of each row, or each cluster of rows in the case of block sampling, being selected as part of the sample. It does not mean that the database will retrieve exactly sample percent of the rows of the table.)

T, where T is the specified time constraint. The desired s is a root of equation $f_Q(s) - T = 0$ and is obtained using a root finding algorithm as before. Note that Oracle's EXPLAIN PLAN feature is used to estimate $f_Q(r)$ and $f_Q(s)$.

However, for queries involving multiple tables, there is no easy way to determine the set of tables to which the sampling clauses should be added and to determine the respective sample sizes. For queries involving joins of tables via foreign key, we add the sampling clause only to the fact table as proposed in [8]. For the class of queries that do not involve joins on a foreign key, we propose an algorithm that determines the tables for which the sample clause should be added and computes the corresponding sample sizes (see Section 5.2.3 for details).

Adding the SAMPLE and the ROWNUM clauses to the queries as part of query transformation allows supporting soft time constraints. That is, the system only transforms the query based on the EXPLAIN PLAN time estimates. However, the actual execution may take longer than the specified time constraint.

3.5 Hard Time-Constrained Query Definition and Processing

Definition*: A query Q with a hard time constraint of T seconds* $\Rightarrow t_{elapsed}(Q) <= T$.

That is, the query must complete within the specified time limit. We propose the following scheme for supporting hard time constraints: 1) Transform the input query by treating the specified time limit as a soft constraint. The estimated time for the transformed query meets the specified time limit. 2) Generate execution plan and use the estimated time information for various operations to associate timers as follows:

- A timer for top-level operation with time set to the specified time limit.
- A timer for every blocking operation with time set to the estimated time for corresponding operation in execution plan if `'APPROXIMATE RESULT'` is specified.
- If `'PARTIAL RESULT'` is specified, the "`ROWNUM <= RNO`" condition can be pushed into some blocking operations, the variable `RNO` will be adjusted in each of these blocking operation to meet its time budget during the execution.

3.6 Aggregates in Time-Constrained Queries

When time-constrained queries involve aggregates and the SAMPLE clauses are added, the user would by default get the aggregate values based on the data sampled. We propose three new user-defined aggregate functions – `estimatedSum`, `estimatedCount`, and `estimatedAvg` that can estimate the aggregate values for the complete table data even though only a portion of the table is used to compute the aggregates. Also, we propose three ancillary user-defined aggregate functions – `sumConfidence`, `countConfidence` and `avgConfidence` that can compute confidence intervals quantifying the uncertainty in the corresponding estimated aggregate values. The ancillary aggregate functions take in the expression and the probability with which to compute the estimate. Note that the user-defined aggregate function `estimatedAvg` returns the same value as `AVG` does, but it is needed for computing confidence interval `avgConfidence`. The confidence interval functions are based on Central Limit Theorem, and Hoeffding's inequality, which are well covered in several papers [6, 30, 31].

A query with built-in and user-defined aggregate functions:

```
Q2: SELECT COUNT(*),
           estimatedAvg(salary),
           avgConfidence(salary, 95)
    FROM employees
    SOFT TIME CONSTRAINT (50)
    WITH APPROXIMATE RESULT;
```

The above query returns the exact count for the sampled portion of `employees` table, the estimated average salary for the entire `employees` table, and also returns the confidence interval (say `c`) [6], indicating that the estimated average salary is within ± `c` of the actual computed average salary, with 95 percent probability. Note that when the time limit is large enough to process the entire `employees` table, `estimatedSum` and `estmatedCount` will return the same values as `SUM` and `COUNT` do, and the confidence interval functions will return 0.

Both the estimated aggregates and the confidence interval functions utilize collected statistics (such as histograms) and run-time query results to refine their final values. They can help the user determine the nature of error in the returned results.

3.7 EXPLAIN_TQ_RESULTS View

The characteristics of time-constrained SQL query results are maintained in an in-memory data structure, which is available to the user as EXPLAIN_TQ_RESULTS (RESULT_TYPE, NUMROWS, ESTIMATED_NUMROWS, COMMENT) view. The RESULT_TYPE column displays `'PARTIAL'` or `'APPROXIMATE'`, NUMROWS gives number of rows returned, ESTIMATED_NUMROWS column gives the result set cardinality without time constraint, and COMMENT contains any additional comment regarding the result. For example, when the ROWNUM filter condition is used for the case of partial results, the estimated total number of resultant rows for the following query:

```
Q3: SELECT salary, MAX(tax)
    FROM tax_return
    GROUP BY salary
    ORDER BY salary DESC
    SOFT TIME CONSTRAINT (10)
    WITH PARTIAL RESULT;
```

can be obtained by querying the EXPLAIN_TQ_RESULTS view as follows:

```
SELECT RESULT_TYPE, NUMROWS,ESTIMATED_NUMROWS
FROM EXPLAIN_TQ_RESULTS;

RESULT_TYPE   NUMROWS    ESTIMATED_NUMROWS
-----------   -------    -----------------
    PARTIAL    100000              1000000
```

Only the result characteristics for the most recent query are maintained in this data structure.

4. RELEVANT ORACLE FEATURES

This section describes the relevant Oracle features, and their use in supporting both soft and hard time-constrained SQL queries.

System Statistics and Object-Level Statistics: In the Oracle Database, the package DBMS_STATS is used to collect and modify not only object-level statistics, such as the number of blocks and the number of rows in a table, the height of an index etc., but also system statistics, such as the average number of CPU

cycles per second, average time to read a single block (random read), and average time to read multi-blocks at once (sequential read). Both object-level statistics and system statistics enable the Oracle Database optimizer to calculate CPU and IO costs for each access method in a SQL query and combine them into the total elapsed time.

EXPLAIN PLAN: The EXPLAIN PLAN statement allows users to see the execution plan Oracle follows to execute a specified SQL statement. The execution plan describes each step of the SQL execution plan, such as the join order in the multi-table join query, and how tables are accessed, either via full table scan or via indexes. For example, query 14 in TPC-H can be explained and the execution plan can be shown through the use of "select * from table (dbms_xplan.display)" in Figure 1.

```
| Id  | Operation           | Name     | Rows | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |          |    1|    51 |       | 88300   (9)| 00:04:39 |
|   1 |  SORT AGGREGATE     |          |    1|    51 |       |            |          |
|*  2 |   HASH JOIN         |          |  744K|    36M|    24M| 88300   (9)| 00:04:39 |
|*  3 |    TABLE ACCESS FULL | LINEITEM |  739K|    16M|       | 83047   (9)| 00:04:22 |
|   4 |    TABLE ACCESS FULL |     PART | 2010K|    53M|       |  2840   (8)| 00:00:09 |

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("L_PARTKEY"="P_PARTKEY")
   3 - filter("L_SHIPDATE">=TO_DATE('1995-09-01 00:00:00', 'yyyy-mm-dd hh24:mi:ss') AND
              "L_SHIPDATE"<TO_DATE('1995-10-01 00:00:00', 'yyyy-mm-dd hh24:mi:ss'))
```

Figure 1: A sample EXPLAIN PLAN output

The Oracle optimizer not only chooses the best plan for this query, but also estimates the resulting data set size and the elapsed time in each step.

ROWNUM Pseudo Column Filtering Clause: This clause can be used to limit the result of a query. For example, the query

```
SELECT * FROM
  (SELECT * FROM employees ORDER BY salary)
WHERE ROWNUM <= 10
```

returns the 10 employees with highest salaries. Oracle's cost-based optimizer can generate an optimal execution plan for the query with the ROWNUM predicate.

SAMPLE Clause: Several database systems including IBM DB2, Microsoft SQL server, and Oracle, support random sampling in a SQL query to allow users to find overall patterns quickly from a smaller sized sample. In general, there are two classes of sampling methods: *row sampling* and *block sampling* (or page-level sampling in DB2). If row sampling retrieves every available row and then decides if this row is taken, it may not outperform some queries without sampling (for example, a single table full scan query). However, it can still reduce the data set significantly, which, in turn, can have performance benefits if the results have to be further processed by expensive operations. Moreover in practice, row sampling can be optimized to skip some data blocks once enough sampled rows are obtained. Block sampling is similar to row sampling, except that the blocks are sampled. Since block sampling can reduce the number of disk I/Os, the query with block sampling can run faster than the original query (i.e. without the sample clause). However, if there is a high degree of column value clustering in the blocks, block sampling may not be as accurate as row sampling to represent the entire dataset.

In the presence of indexes, block sampling can be applied on the index blocks only when the optimizer chooses an index fast full scan. For other types of index scans (unique scan, range scan) the optimizer can apply row sampling for the index rows.

Table Functions: We model the time-constrained query as a SQL table function, which takes in the original SQL query to be executed, the time limit in seconds, the nature of time constraint (`'SOFT'` or `'HARD'`), and the nature of result (`'PARTIAL'` or `'APPROXIMATE'`). Our implementation makes use of the rewritable SQL table function feature of Oracle [27], which allows a table function to be replaced by a generated SQL query.

5. SUPPORTING SOFT TIME CONSTRAINTS

5.1 Optimizing for Partial Results

For time-constrained query `'WITH PARTIAL RESULT'`, the input query is augmented with `'ROWNUM <= RNO'` predicate so that the estimated time for the transformed query is within the specified time constraint. The `RNO` is obtained using the algorithm shown in Figure 2, which makes use of a root finding algorithm [20].

```
Q  := Original Query;
T  := Time Constraint;
ER := Estimated # of result rows from Q;
RNO:= The # used in the ROWNUM predicate;
ET := Estimated time from explain plan for Q;
ET(RNO):= Estimated time from explain plan for
          SELECT * FROM (Q) WHERE ROWNUM <= RNO;

  explain plan for Q to obtain ER and ET;
  if (ET <= T) return Q;
  else {
    explain plan for
       SELECT * FROM (Q) WHERE ROWNUM <= 1;
    if (ET(1) > T)
      return 'ERROR: NEED AT LEAST $ET(1) s';
    else {
      RNO = root_finding_algorithm(1, ER,
               (ET(1)-T), (ET-T), T, Q);
      return
       'SELECT * FROM (Q) WHERE ROWNUM <= RNO';
    }
  }
```

Figure 2: Computing estimated rownum

In our implementation, the false position method [21] is used as our root finding algorithm.

As an example, after optimizing for response time, the query Q3 specified in Section 3.7 gets rewritten to

```
Q3-T: SELECT *
    FROM (SELECT salary, MAX(tax)
          FROM tax_return
          GROUP BY salary
          ORDER BY salary DESC)
    WHERE ROWNUM <= RNO;
```

When the time constraint is inadequate to return the first single row, an error is returned to end users with the information such as a lower-bound time limit, above which the time-constrained query would most likely yield the first row.

5.2 Optimizing for Approximate Results

When an end user specifies the `'WITH APPROXIMATE RESULT'` clause, our system will try to transform the query with the time constraint to a query with sampling on some table(s) so that the estimated time for the transformed query is within the specified time constraint. Since a random sample is used, the results for queries involving aggregates may be approximate. This

'APPROXIMATE' mode can significantly reduce the query processing costs by using sampling. Thus, it allows the query to complete within a specified time constraint. However, end users should use estimatedSum/sumConfidence, estimatedCount/countConfidence, and estimatedAvg/avgConfidence to check how well the approximate results are computed. An alternative is to let the system figure out the minimal sample size for each table so that a statistically valid approximate result can be returned. However this problem is beyond the scope of this paper.

5.2.1 Single Table Query

Since long running queries over a single table typically use a full table scan, the query transformation involves augmenting the query with a sample block clause for the table. The sample size is obtained in the same fashion as in Section 5.1, i.e. by using a root finding algorithm via EXPLAIN PLAN trials.

5.2.2 Multi-Table Query with Foreign Key Join

According to [8], a uniform random sample over foreign-key joins of tables can be achieved through a uniform random sampling over the fact table and then joining with other dimension tables. We adopt this approach for transforming a time-constrained query over multiple tables with foreign key joins. Specifically, we analyze the table structures (especially foreign keys and primary keys) and determine which table is the fact table. The sample size for the fact table is obtained in the same fashion as in Section 5.1, i.e. by using a root finding algorithm via EXPLAIN PLAN trials.

Note that the presence of foreign keys and primary keys does not mean that the best execution plan will utilize the index structures associated with these keys. For example, a hash join method could perform much better than an index nested-loop join method as shown in figure 1. Another example is that if there is a selective predicate in a dimension table, other index structures, such as bitmap index and bitmap join index, are better alternatives. This approach requires that only one table is the fact table, and other tables are joined through the foreign key either directly from the fact table or from the already joined tables. Though it appears restrictive, the primary key/foreign key requirements normally hold for many well-defined (normalized) schemas (e.g., TPC-H Benchmark [26]).

5.2.3 Multi-Table Query without Foreign Key Join

When some tables are joined without foreign keys (even in the presence of other tables being joined with foreign keys), we introduce the following algorithm, which tries to achieve the goal of returning as many resulting rows as possible (for queries involving aggregates, the goal is to sample as many rows as possible in the resultant joins).

Let R_1, R_2: The two tables that need to be joined; f_1, f_2: The sample size for the two tables respectively; and f: The sample size corresponding to the join result.

Consider samples S_1 = SAMPLE (R_1, f_1), S_2 = SAMPLE(R_2, f_2), and S = SAMPLE ($R_1 \bowtie R_2$, f). Since S = SAMPLE ($R_1 \bowtie R_2$, f) requires more time than $S_1 \bowtie S_2$ = SAMPLE(R_1, f_1) $\bowtie$ SAMPLE(R_2, f_2), we only need to consider the latter case, that is sampling prior to join. Also, $f_1 * f_2$ is the upper bound for f (from Theorem 12 in [22]). To simplify the calculation of f, we assume that f is a linear function of $f_1 * f_2$. Thus, sampling should maximize $f_1 * f_2$ while meeting the time constraint.

Assume that T_1 is the total time to process R_1 and T_2 is the total time to process R_2. The total query time T can be simplified as:

$$T = T_1 + T_2.$$

Note that T_1 and T_2 do not mean the time to simply scan R_1 and R_2. They can also include the time to sort a table, and to hash-build or probe a table. In the nested loop join case, the total time to process an inner relation includes the time to scan the inner relation as many times as needed. We also specify t_1 and t_2 to be the time to process S_1 and S_2.

Here we discuss the nested loop join and hash join in detail, and the sort-merge join briefly.

5.2.3.1 Nested Loop Join

Considering nested loop join as the join method for $S_1 \bowtie S_2$, and S_1 and S_2 are the outer relation and inner relation that are block-sampled directly from R_1 and R_2, we can simplify the total time (for processing $S_1 \bowtie S_2$) to be:

$$T = t_1(f_1) + t_2(f_1, f_2) = f_1 * T_1 + f_1 * f_2 * T_2.$$

That means that the time to process S_1 is linear to the sample size f_1 while the time to process S_2 is linear to f_1 and f_2. Note that T_2 includes the time to scan S_2 as many times as needed. To make $f_1 * f_2$ maximum, we can prove the following theorem.

Theorem 1:

When $f_2 = 1$, $\max(f_1 * f_2) = f_1 = T/(T_1 + T_2)$.

Proof:

Assume $f_2 < 1$, $f_1 * f_2$ is maximal:

$$T = f_1 * T_1 + f_1 * f_2 * T_2$$
$$> f_1 * f_2 * T_1 + f_1 * f_2 * T_2 = f_1 * f_2 * (T_1 + T_2).$$
$$\Rightarrow \quad f_1 * f_2 < T/(T_1 + T_2).$$

But when $f_2 = 1$, $f_1 * f_2 = f_1 = T/(T_1 + T_2)$.

Therefore, when $f_2 = 1$,

$$\max(f_1 * f_2) = f_1 = T/(T_1 + T_2).$$

Theorem 1 tells us that the sample clause should be put into the outer relation, or the driver node [12]. The same conclusion can be extended to the case of multi-table nested loop joins. The total time to perform *n*-table nested loop joins can be written as follows.

$$T = f_1 * T_1 + f_1 * f_2 * T_2 + f_1 * f_2 * f_3 * T_3 + \ldots + (\prod f_1) * T_n.$$

Theorem 2:

When $f_i = 1$, where $2 \le i \le n$,

$\max(\prod f_i) = f_1 = T/\sum T_i$.

The proof to Theorem 2 is similar to the above proof to Theorem 1, and is ommited due to space limitations. Therefore, to maximize the sample size f, the outer relation should be sampled in a pipelined multi-table nested loop join and other inner relations should be processed as before. In other words, the optimal plan is to apply only the reduction factor of the outer relation (i.e. f_1) to every stage in the pipeline.

The same conclusion can also be made when the inner relation takes an index unique scan, or the index nested loop join is performed. In this case, only row sampling method can be applied to the inner relation. But because the reduction factors $f_2, \ldots, f_n$ cannot affect T_1, the optimal plan is to make $f_2 = f_3 = \ldots = f_n = 1$ regardless of row sampling or block sampling methods taken in the inner relation. When only row sampling is available to

the outer relation (such as an index range scan followed by a table access by index), we have to divide T_1 into two parts: TC_1 is the constant portion of T_1 (i.e. the time to perform an index range scan), and TF_1 is the flexible portion of T_1 that can be affected by f_1 (i.e. the time to perform a table access by index). So some of the above equations are rewritten as

$$T_1 = TC_1 + TF_1.$$
$$T = TC_1 + f_1*TF_1 + f_1*f_2*T_2 + f_1*f_2*f_3*T_3 + \ldots + (\prod f_i)*T_n.$$

Corollary 3:

When $f_i = 1$, where $2 \le i \le n$,
$\max(\prod f_i) = f_1 = (T - TC_1)/(\sum T_i - TC_1)$.

Therefore, when row sampling is applied to the outer relation, we should deduct the constant portion of T_1 from the time constraint, and then compute the sample size f_1.

5.2.3.2 Hash Join

Considering in-memory hash join as the join method for $S_1 \bowtie S_2$, and S_1 and S_2 are block-sampled directly from R_1 and R_2, we can simplify the total time to be:

$$T = t_1(f_1) + t_2(f_2) = f_1*T_1 + f_2*T_2.$$

To make f_1*f_2 maximum, we can prove the following theorem.

Theorem 4:

1. If $f_1 < 1$ and $f_2 < 1$, then
   $t_1(f_1)=t_2(f_2)=f_1*T_1=f_2*T_2= \frac{1}{2}*T$,
   $\max(f_1*f_2) = (T*T)/(4*T_1*T_2)$;
2. If $f_2 = 1$, then $t_1(f_1) \ge t_2(f_2)$ or $f_1*T_1 \ge T_2$,
   $\max(f_1*f_2) = f_1 = (T - T_2)/T_1$;
3. If $f_1 = 1$, then $t_2(f_2) \ge t_1(f_1)$ or $f_2*T_2 \ge T_1$,
   $\max(f_1*f_2) = f_2 = (T - T_1)/T_2$;

Proof:

(1). We start the case of $f_1 < 1$ and $f_2 < 1$. Assume when $f_1 = g_1$ and $f_2 = g_2$, f_1*f_2 is maximal. So other values meeting the time constraint such as h_1*h_2 should be not larger than g_1*g_2. We can take

$$h_1=(g_1*T_1+g_2*T_2)/(2*T_1). \quad (1)$$
$$h_2=(g_1*T_1+g_2*T_2)/(2*T_2). \quad (2)$$

Assume $T_1 \ge T_2$, then

$$h_1=(g_1*T_1+g_2*T_2)/(2*T_1) = g_1/2+(g_2*T_2)/(2*T_1)<1/2+1/2=1. \quad (3)$$

Assume $h_2 < 1$ for now and we will prove it later. Therefore,

$$h_1*h_2=((g_1*T_1+g_2*T_2)*(g_1*T_1+g_2*T_2))/(4*T_1*T_2) \le g_1*g_2. \quad (4)$$
$$\Rightarrow (g_1*T_1)^2+(g_2*T_2)^2+2*g_1*T_1*g_2*T_2 \le 4*g_1*T_1*g_2*T_2. \quad (5)$$
$$\Rightarrow (g_1*T_1-g_2*T_2)^2 \le 0. \quad (6)$$
$$\Rightarrow g_1*T_1-g_2*T_2 = 0, \text{ or } g_1*T_1=g_2*T_2=\tfrac{1}{2}*T. \quad (7)$$

To prove $h_2 < 1$, we take

$$m_2=1. \quad (8)$$
$$m_1=(g_1*T_1+g_2*T_2-m_2*T_2)/T_1=(g_1*T_1+g_2*T_2-T_2)/T_1= g_1+(g_2-1)*T_2/T_1. \quad (9)$$

Because

$$m_1*m_2 \le g_1*g_2. \quad (10)$$

Therefore,

$$g_1+(g_2-1)*T_2/T_1 \le g_1*g_2. \quad (11)$$
$$\Rightarrow (g_2-1)*T_2/T_1 \le (g_2-1)*g_1. \quad (12)$$
$$\Rightarrow \text{ because } (g_2-1) < 0, \quad T_2/T_1 \ge g_1, \text{ or } (g_1*T_1)/T_2 \le 1. \quad (13)$$

Go back to (2),

$$h_2=(g_1*T_1+g_2*T_2)/(2*T_2) = (g_1*T_1)/(2*T_2)+g_2/2. \quad (14)$$

Because (13) and $g_2/2 < 1/2$,

$$h_2 < 1. \quad (15)$$

Therefore, if $f_1 < 1$ and $f_2 < 1$, we should make $f_1*T_1 = f_2*T_2 = \frac{1}{2}*T$ to have $\max(f_1*f_2) = (T*T)/(4*T_1*T_2)$.

(2). The second case ($f_2 = 1$) and the third case ($f_1 = 1$) can be proved by contradiction, and are ommited due to space limitations.

Furthermore, the above theorem can be extended to in-memory multi-table hash joins. Assume that the total time to perform in-memory *n*-table hash joins can be written as follows.

$$T = f_1*T_1 + f_2*T_2 + f_3*T_3 + \ldots + f_n*T_n.$$

To maximize $f_1*f_2*\ldots*f_n$, we have the following theorem.

Theorem 5:

If $f_i < 1$ and $f_j < 1$, then
$t_i(f_i) = t_j(f_j) = f_i*T_i = f_j*T_j$.

The detailed proof is similar to the proof to Theorem 4 and is ommited due to space limitations. Both Theorem 4 and Theorem 5 tell us 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.

When row sampling is applied for some relations, we will have the constant and flexible portions of the time to process these relations. Assume S_i and S_j are row-sampled from R_1 and R_2 while S_k is block-sampled from R_k.

$$T_i = TC_i + TF_i.$$
$$T_j = TC_j + TF_j.$$
$$T = f_1*T_1 + \ldots + TC_i + f_i*TF_i + \ldots + TC_j + f_j*TF_j + \ldots + f_k*T_k + \ldots + f_n*T_n.$$

Corollary 6:

If $f_i < 1$, $f_j < 1$ and $f_k < 1$,
then $f_i*TF_i = f_j*TF_j = f_k*T_k$ or
$t_i(f_i)-TC_i = t_j(f_j)-TC_j = t_k(f_k)$.

Therefore, when row sampling is applied to the some relations, we should use only the flexible portions of T_i and T_j to compute the sample size f_i and f_j.

When computational resources (such as memory) are not sufficient for a hash join operation, in-memory data will be written to disk and reread multiple times. This hash join method, referred as multi-pass hash join, may degenerate into a case similar to the nested loop join (i.e. the number of the probe passes is linear to the size of the build relation). For sort-merge join, which has the time complexity of O(nlogn), there is no easy analytical solution for this case. So our approach is to heuristically use the equations obtained in Theorem 4, Theorem 5, and Corollary 6 to choose which tables need to be sampled and compute their sample sizes, i.e. to make $t_i(f_i)$ as close as $t_j(f_j)$ when S_i and S_j are block-sampled or to make $t_i(f_i)-TC_i$ as close as $t_j(f_j)-TC_j$ when S_i and S_j are row-sampled.

5.2.3.3 Algorithm for Multi-Table Query without Foreign Key Join

When the three join methods are used together in a query execution plan, we can view the pipelined nested loop joins as a single relation and then use the equations in Theorem 4, Theorem 5, and Corollary 6. The algorithm is shown in Figure 3. For the other join variants (such as outer-join and semi-join operations), the sampling operation is pushed in the way similar to the normal join operation. Note that outer-join query with sampling can return some rows with column values incorrectly reported as NULL. For the anti-join operations, like the *Difference* operation in 5.2.4, only the outer relation can be sampled.

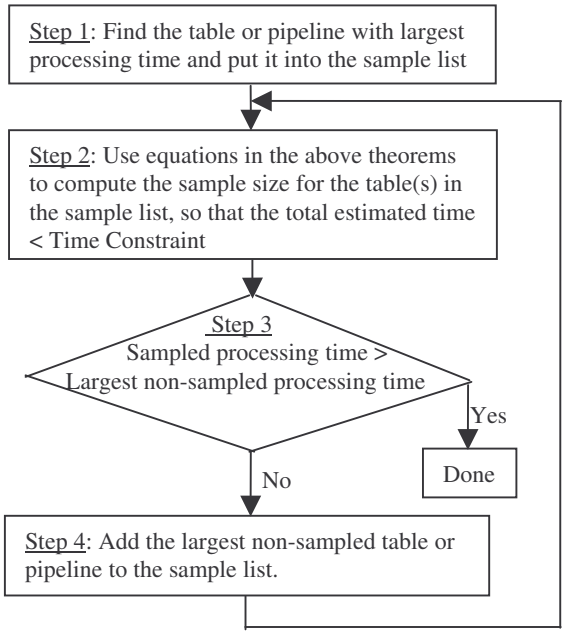

Figure 3: An algorithm to choose which tables to be sampled and their sample sizes

Note that in Step 2 of Figure 3, we may need a root finding algorithm to decide the sample sizes via several EXPLAIN PLAN trials for cases when the total cost function is not a linear function. In addition, steps 2 and 3 can be combined to avoid unnecessary iterations in the root-finding algorithm. However, for clarity of explaining this algorithm, these are shown as two separate steps. Because our algorithm uses EXPLAIN PLAN iteratively, it affects the compilation time. To reduce the compilation time of our algorithm, we retain the join orders and join methods of the original query execution plan since a major part of the compilation time is spent in determining the join orders and join methods. However in some cases, it may not generate the best execution plan for the transformed query. For example, the best plan for the original query can be a hash-join operation while the best plan for transformed query can be a nested-loop-join operation. Ideally we should set an adaptive limit to the compilation time, and allow our algorithm to try different join orders and join methods. However, this area is a candidate for future study.

5.2.4 Other Operations on Relations

As we describe above, the sampling methods are pushed to the table scan or index scan as early as possible. But some selection and projection operations can be performed before the row sample method. For example, an index scan first performs the selective access predicate, and then applies the row sample method. In other cases, the relational operations, including sort and group-by operations, are normally performed after sampling.

For the set operations, we briefly describe our schemes. (1) *Union:* For $\texttt{SAMPLE}(R_1 \cup R_2, f)$, we simplify the operation to be $\texttt{SAMPLE}(R_1, f_1) \cup \texttt{SAMPLE}(R_2, f_2)$ with $f_1 = f_2$. Therefore, $t_1(f_1)/t_2(f_2) = (f_1*T_1)/(f_2*T_2) = T_1/T_2$ when the cost function is linear. (2) *Difference:* $\texttt{SAMPLE}(R_1-R_2, f)$ is simplified to be $\texttt{SAMPLE}(R_1, f_1) - R_2$. Therefore, only R_1 is sampled. (3) *Intersect:* $\texttt{SAMPLE}(R_1 \cap R_2, f)$ is simplified to be $\texttt{SAMPLE}(R_1, f_1) \cap \texttt{SAMPLE}(R_2, f_2)$ with maximizing f_1*f_2. This is similar to the join operation.

5.2.5 Sub-queries

When a sub-query with an aggregate function appears in a predicate condition, we try not to push the sample operation into the sub-query until all other options are used up. That's because the predicate condition can change due to the approximate aggregate value and some incorrect rows can be returned as the result. For example, in the following query, the sample operation is performed only on the outer `employees` table if the sub-query takes less than 10 seconds:

```
Q4: SELECT *
    FROM employees outer
    WHERE outer.salary >
           (SELECT AVG(inner.salary)
            FROM tax_return inner)
    SOFT TIME CONSTRAINT (10);
```

If the sampling is also applied to the sub-query, we will insert a comment, which can be seen in EXPLAIN_TQ_RESULT view.

```
    RESULT_TYPE                    COMMENT
    -----------        ----------------------
    APPROXIMATE        APPROXIMATE PREDICATE
```

The time allocated to each stage is determined through linear interpolation: assuming the original query takes 30 seconds (18 seconds are for the sub-query block and 12 seconds are for the top query block), we will spend 6 (18*10/30) seconds in the sub-query block and 4 (12*10/30) seconds in the top query block. When the time constraint is increased to 20 seconds, the sub-query will be executed completely in the 18 seconds and only 2 seconds are spent in the top query. But the predicate used in the top query will be exact thereby producing accurate results.

The above *top-down* approach to allocate the time in each query block, i.e. the time allotment in each stage is determined first, and then the sample size is computed to meet the time allotment. It differs from the *bottom-up* approach in Figure 3, where the sample size(s) are computed first to meet certain conditions (such as equations in the previous theorems and corollaries) and then we try to satisfy the total time constraint.

The preference for exact aggregates in sub-queries used in predicate condition also applies to the correlated sub-queries. For example, if possible we should use the exact aggregate values computed from the sub-query in the following query:

```
Q5: SELECT *
    FROM employees outer
    WHERE outer.salary >
          (SELECT AVG(inner.salary)
           FROM tax_return inner
           WHERE inner.location = outer.location)
    SOFT TIME CONSTRAINT (10);
```

This is similar to the nested loop join operation, i.e. to push the sample operation into the outer relation. However the above query can be un-nested as shown below with an inline view [17]:

```
Q5-inline-view: SELECT *
    FROM employees outer,
         (SELECT AVG(salary) avg_salary, location
          FROM tax_return inner
          GROUP BY location) view
    WHERE outer.salary > view.avg_salary
      AND outer.location = view.location
     SOFT TIME CONSTRAINT (10);
```

This is similar to the hash join operation, i.e. to push the sample operations to both the relations to maximize the product of the sample sizes. This contradicts the preference that the exact aggregate values should be computed from the sub-query. To solve this problem, we can use a sample or hash method that takes a column value from each input row and then determines if the row is returned or not. For example, in oracle we can rewrite Q5-inline-view into the query as follows:

```
Q5-inline-view-T: SELECT *
    FROM employees outer SAMPLE BLOCK (20),
         (SELECT AVG(salary) avg_salary, location
          FROM tax_return inner
          WHERE ORA_HASH(location, 99) < 20
          GROUP BY location) view
    WHERE outer.salary > view.avg_salary
      AND outer.location = view.location;
```

The function `ORA_HASH(location, 99)` is applied to each row of the table `tax_return`, like a row sampling method. It computes a hash value (in [0,99]) for "`location`" column, and returns this row only when the hash value is less than 20. Since `ORA_HASH` is a deterministic function, the rows with the same location will be put into the same hash bucket. Therefore, the inline view returns the exact aggregate values of the sampled groups. In other words, when the row sample or hash method applies to the same group-by column, the exact aggregate values can be returned from some sampled groups.

6. SUPPORTING HARD TIME CONSTRAINTS

A hard time-constrained query must complete in the specified time. This is achieved by transforming the query assuming a soft time constraint, and then adding a timer to every blocking operation as well as adding a top-level timer. Note that a blocking operation does not produce any rows until it has consumed at least one of its inputs completely. For example, `Sort` and the `Build-relation` in a hash join are blocking operations. Once a timer reaches its limit, it terminates the underlying subordinate operation. However, if the operation completes before the timer expires then the timer is simply nullified.

To determine the time limit for each timer, we rely on the time estimated for each step by the EXPLAIN PLAN statement. Consider the EXPLAIN PLAN shown in the Figure 1. There are two blocking operations: 1) the `Build-relation` in `HASH JOIN` node (from `LINEITEM` table); 2) the `SORT AGGREGATE` node to compute the aggregate function. Since the estimated elapsed time to access the table `LINEITEM` is 262 seconds, the first timer is set up in the `Build-relation` of the `HASH JOIN` node (Id=2), which stops its subordinate operation (i.e. full table scan of `LINEITEM`) in 262 seconds. The second timer is set up in the `SORT AGGREGATE` node (Id =1), which stops its subordinate operation in 279 seconds. The third timer is the top-level timer (Id=0), which stops in 279 seconds. In this case where the child node of the top node is a blocking operator, the top-level timer may not be as important as the timer associated with the child node. However, for the case where the child node of the top node is not a blocking operator, the top-level timer is needed to guarantee that the query is completed within the time constraint.

Our approach of adding timers for each blocking operation leads to a more balanced distribution of time, namely, it avoids overspending of time in any one blocking operation. However, it can lead to approximate results. Thus, when a user specifies a hard time constraint with `'PARTIAL RESULT'`, only the top-level timer is added, which ensures that any result returned is accurate. But since the "`ROWNUM <= RNO`" condition can be pushed into some blocking operations, the variable `RNO` will be adjusted in each blocking operation to meet its time budget during the execution. For a hard time constraint with `'APPROXIMATE RESULT'`, the timers for blocking operations are added.

Because the compilation time can become an important factor for the query with hard time constraints, we will utilize the top-down approach (described in Section 5.2.5) to compute the sample size. For example, in a sort-merge join operation, assuming T_a: time to process the relation R_a and T_b: time to process the relation R_b, the time limits (t_a, t_b) for both relations can be either (T_a, $T - T_a$) for sampling only on relation R_b, (½*T, ½*T) for sampling on both relations, or ($T - T_b$, T_b) for sampling only on relation R_a. Their sample sizes are simplified as $f_a = \min(C_{safe}*(t_a/T_a), 1)$ and $f_b = \min(C_{safe}*(t_b/T_b), 1)$, where C_{safe} (> 1) is used as the safety margin to return enough sample rows before the stop timer. Thus, when the compilation time is a major factor for the query with hard time constraints, we can use the top-down approach to compute not only the time limit in each stage, but also its sample size, without iterative trials.

7. EXPERIMENTS

This section describes the experiments conducted using the prototype built on Oracle. Here we report the experiments for soft time constraints with approximate results. The experiments corresponding to execution time reduction by reducing the cardinality of the result set and employing top-k optimization are skipped as they have been well covered in several papers [3, 4, 5].

7.1 Experimental Setup

The experiments were done on a Dell PE650 machine (Intel P4 3.0Ghz with Hyper-Threading), with 2GB main memory, and 80GB hard disk. The operating system is Redhat Enterprise Linux 3 and the database system is Oracle Database 10g Release 2 Enterprise Edition. The relevant database parameters are: `db_block_size=8192`, `db_cache_size=160M`, and `shared_pool_size=160M`.

7.2 Dataset

The dataset is generated from the DBGEN program, recommended by the TPC-H Benchmark [26]. Scale factor is set to 10, which translates to a database size of about 10GB, consisting of 8 tables (PART, SUPPLIER, PARTSUPP, CUSTOMER, ORDERS, LINEITEM, NATION, and REGION). The LINEITEM table is the biggest and has ~60 million rows, whereas the ORDERS table is the second largest with 15 million rows. After loading the dataset, the primary key and foreign key constraints are created.

Before running the experiments, both system statistics and object-level statistics are collected. Each experiment is conducted six times in warm cache, and the average query execution time is computed. The estimated time is obtained from the EXPLAIN PLAN statement. Note that the query compilation time (including the optimization time) is not significant (< 0.5 second) in the four experiments. Thus, we report the total execution time only.

7.3 Experiment I: Single Table Query with Aggregates

This experiment is conducted using the Q6 (Forecasting Revenue Change) query in TPC-H. This query considers all the lineitems shipped in a given year with discounts between a ± 0.01 of DISCOUNT=0.06. The original query on LINEITEM table, estimated to complete in 269 seconds, completed in an average of 244 seconds. 10%, 20%, 30%, 40%, and 50% of 269 seconds are chosen as the time constraints. The query transformation involves determining the sample size using the method discussed in section 5.2. Since the table scan of LINEITEM is the major factor in the total execution time, the estimated sample size for each time limit is linear to the time limit, i.e. the sample sizes are close to 10%, 20%, 30%, 40%, and 50% respectively. Figure 4 shows the estimated and execution time for different time constraints.

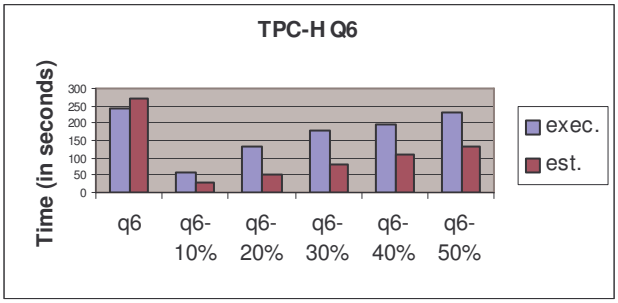

Figure 4: Estimated and execution time to run TPC-H Q6 in settings: no time constraint, 10%, 20%, 30%, 40%, and 50% of Q6's estimated time

Table 1: Sum, est. sum, and 95% conf. interval value for Q6

%	SUM	estimatedSum	sumConfidence
10%	113894821	1228484983	49916216
20%	243045023	1230244879	34194884
30%	370097887	1228624043	27692357
40%	489547623	1228986671	24081572
50%	617119157	1229137335	21449857
100%	1230113636	N/A	N/A

The transformed queries (augmented with sampling clause) take longer than the estimated time to complete. That is because the extra cost for "sample block" clause may not be properly accounted for by the cost-based optimizer. Specifically, the original query can fully utilize multi-block I/O, but the "sample block" query has to randomly skip some blocks. But the optimizer uses multi-block I/O time to calculate the estimated time for the "sample block" query. However, the overall results are still encouraging as the execution time does decrease as the user specifies smaller time-constraints.

We also obtained the `estimatedSum` and `sumConfidence` (95%) values for each time constraint, using Hoeffding-based bounds [6] (Table 1). The estimated value is close to the actual value of SUM, and the error reduces as the sample size increases.

7.4 Experiment II: Single Table Query with GROUP BY and ORDER BY clause

This experiment is conducted using the Q1 (pricing summary support) query of TPC-H. The original query on LINEITEM table, estimated to complete in 340 seconds, completed in an average of 367 seconds. 10%, 20%, 30%, 40%, and 50% of 367 seconds are chosen as the time constraints. Again, since the query primarily involves a single table, the estimated sample size percentages are linear with respect to the time constraints. Figure 5 shows the execution times for various time constraints.

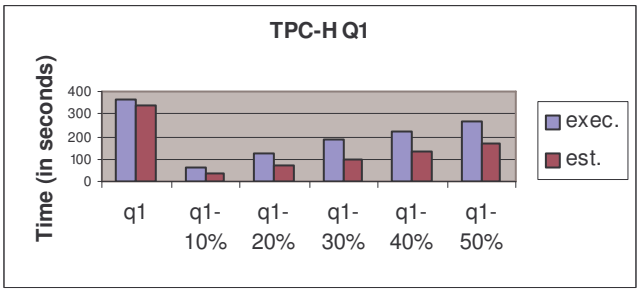

Figure 5: Estimated and execution time to run TPC-H Q1 in settings: no time constraint, 10%, 20%, 30%, 40%, and 50% of Q1's estimated time

The results (Figure 5) show that the query execution time reduces as smaller time constraints are chosen. Like Experiment I, the actual execution time is larger than the time estimated by Oracle's cost-based optimizer. However, the percentage error in estimation (measured as `abs(execution-time - estimated-time)*100/estimated-time`) is less when compared to the error observed in Experiment I observations (see Table 2).

Table 2: Percentage error in estimation (Experiment I vs. II)

	10%	20%	30%	40%	50%
Exp I	104	146	120	81	69
Exp II	85	82	83	66	55

The reduction in the error can be attributed to the fact that the under-estimation caused by the changes to multi-block I/O in the transformed sampling block query is mitigated by the additional processing costs of grouping and sorting operations.

7.5 Experiment III: Two Table Join Query with Aggregates

This experiment is conducted using the Q14 (promotion effect) query in TPC-H, which involves join of LINEITEM and PART tables. For the queries with the time constraint, the table LINEITEM is sampled because it has a foreign key reference to P_PARTKEY. Since LINEITM is much larger than PART table, the join query exhibits characteristics of a single table query. Thus, the sampling sizes estimated are linear with respect to time-constraints (as in Experiment I and II). Figure 6 shows the execution and estimated times for Q14. Like Experiment I and II, the query processing times are under-estimated as the optimizer

considers multi-block I/O count, which is not fully exercised for transformed sampling block queries.

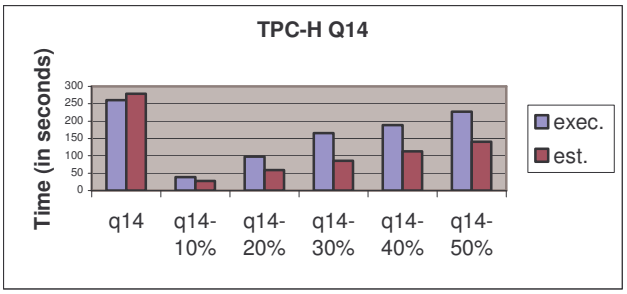

Figure 6: Estimated time and execution time to run TPC-H Q14 in settings: no time constraint, 10%, 20%, 30%, 40%, and 50% of Q14's estimated time

7.6 Experiment IV: Four Table Join Query with GROUP BY and ORDER BY clause

This experiment is conducted using the Q10 (returned item reporting) query in TPC-H. The original query returns the first 20 selected rows. However, in our experiment, we removed the ROWNUM predicate to return all the selected rows. This query needs to join CUSTOMER, ORDERS, LINEITEM and NATION tables. For the queries with the time constraint, the table LINEITEM is sampled because it has a foreign key reference to O_ORDERKEY. The estimated sampling sizes for various time constraints is shown in Figure 7, which displays the non-linear nature of relationship between sampling size and the time limit specified as part of the time constraint.

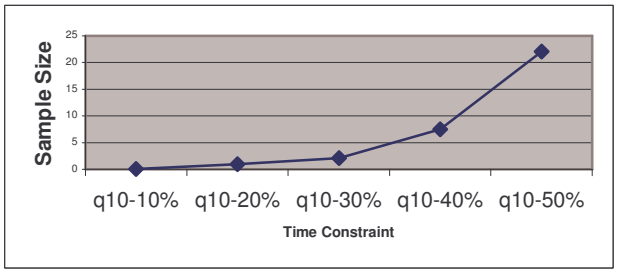

Figure 7: Estimated sample size in settings: 10%, 20%, 30%, 40%, and 50% of Q10's estimated time

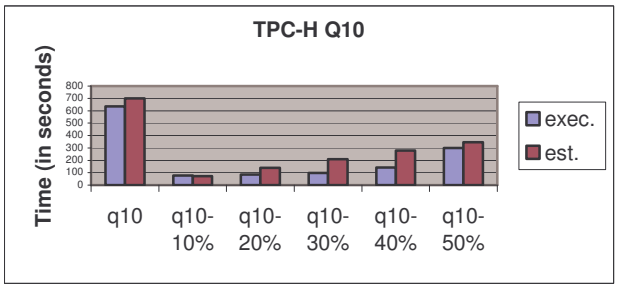

Figure 8: Estimated time and execution time to run TPC-H Q10 in settings: no time constraint, 10%, 20%, 30%, 40%, and 50% of Q10's estimated time

Figure 8 shows the execution and estimated times for the transformed queries. The time constraint is able to significantly reduce the overall query processing time. An interesting aspect is that the estimated times are larger than the execution times. This reversal in error (as compared to Experiments I, II, and III) can be attributed to the interaction of the augmented sampling clause with the WHERE clause predicates, which leads to reduction in the data that needs to be processed by the GROUP BY and ORDER BY clauses.

8. DISCUSSION

Here we discuss the various aspects of our approach.

Experiments: The experiments clearly demonstrate the effectiveness of time-constrained queries. That is, use of smaller time limits does reduce the overall query execution time.

Relying on Oracle's Cost-Based Optimizer: Our approach to rely on Oracle's cost-based optimizer (namely via EXPLAIN PLAN) to get time estimates allows us to leverage the rich functionality of Oracle's cost-based optimizer. For example, this allows us to handle queries involving partitioned tables and parallel execution in a seamless manner. However, we are relying on Oracle's cost-based optimizer capability to not only generate the optimal execution plans but also to accurately estimate the time needed to process the query. The latter is significantly more challenging and needs further exploration so that the error in estimated times with respect to actual query execution times can be minimized.

Soft vs. Hard Time Constraints: Although the soft time constraint query doesn't guarantee that the query can be completed in the specified time constraint, it does generate an execution plan for which the estimated time is less than the time constraint. This might be useful, for example, to enforce the policy where users are not allowed to submit queries that are estimated by the optimizer to take longer than the specified maximum execution time [18]. For such cases, the user can issue the same query with the soft time constraint. Moreover, the support for soft time-constrained queries forms the building block for hard time-constrained queries.

Loose Integration vs. Tight Integration: The prototype supports soft time-constrained query using the EXPLAIN PLAN feature and the extensibility features of Oracle (table functions). We use rewritable table functions to append additional predicates to the user query. Since all this functionality resides outside the Oracle server, it can be viewed as a loose integration. For our approach to work efficiently in all cases, we need to have access to the final query generated after view transformation, query rewrite using materialized views, and various other transformations [17]. Thus, the technique to augment a query with additional predicates will be more efficient when implemented in the database kernel. Also, for hard time constraints, the timers need to be associated with each blocking operation. Thus, a tight integration with the database would be highly desirable.

Nature of Results: In general, a time-constrained query result can have many aspects including i) Is result set *complete* or *partial*? ii) Are resulting row values *accurate* or *approximate*? iii) Are the results returned in *ordered* or *unordered* manner? (relevant for queries with ORDER BY clause) Consider the following query that needs to be executed within a time constraint:

```
SELECT AVG(salary) FROM employees
   GROUP BY dept ORDER BY dept;
```

These aspects in turn can lead to many different possible combinations, such as (complete, accurate, and ordered), the no

time-constraint case, (complete, approximate, and ordered), i.e., returning average salary for each and every dept even if the average values for each group is approximate and ordered by `dept`, etc. Ideally, the time-constraint specification should allow the user to specify preference for any one of these combinations. Our current work, does not allow specifying preference for result at this fine level of granularity. This we plan to address in a future work.

9. CONCLUSIONS AND FUTURE WORK

The paper makes a case for supporting time-constrained SQL queries in database systems. The advances in the top-k query optimization, approximate query processing, and error estimation have set the stage for this work. This, in conjunction with the capabilities of cost-based optimizer, namely, the optimal plan generation, and accurate estimation of the query execution time, makes it possible to support time-constrained SQL queries. This paper explored supporting both soft and hard time-constrained SQL queries by leveraging features of Oracle's cost-based optimizer. As a proof of concept a prototype implementation was done on top of Oracle Database. The experimental study conducted with the TPC-H dataset and queries demonstrates the effectiveness of time-constrained SQL queries.

In future, we plan to explore doing a tighter integration of the proposed techniques. Also, we plan to explore the feasibility and effectiveness of supporting hard time constraints.

Acknowledgments: We thank Jay Banerjee and Sue Mavris for their encouragement and support.

10. REFERENCES

[1] Wen-Chi Hou, Gultekin Özsoyoglu, Baldeo K. Taneja: Processing Aggregate Relational Queries with Hard Time Constraints. SIGMOD Conference 1989: 68-77

[2] Gultekin Özsoyoglu, Sujatha Guruswamy, Kaizheng Du, Wen-Chi Hou: Time-Constrained Query Processing in CASE-DB. IEEE Trans. Knowl. Data Eng. 7(6): 865-884 (1995)

[3] Michael J. Carey, Donald Kossmann: On Saying "Enough Already!" in SQL. SIGMOD Conference 1997: 219-230

[4] Michael J. Carey, Donald Kossmann: Reducing the Braking Distance of an SQL Query Engine. VLDB 1998: 158-169

[5] Donko Donjerkovic, Raghu Ramakrishnan: Probabilistic Optimization of Top N Queries. VLDB 1999: 411-422

[6] Joseph M. Hellerstein, Peter J. Haas, Helen J. Wang: Online Aggregation. SIGMOD Conference 1997: 171-182

[7] Joseph M. Hellerstein, Ron Avnur, Vijayshankar Raman: Informix under CONTROL: Online Query Processing. Data Min. Knowl. Discov. 4(4): 281-314 (2000)

[8] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, Sridhar Ramaswamy: Join Synopses for Approximate Query Answering. SIGMOD Conference 1999: 275-286

[9] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala: Congressional Samples for Approximate Answering of Group-By Queries. SIGMOD Conference 2000: 487-498

[10] Brian Babcock, Surajit Chaudhuri, Gautam Das: Dynamic Sample Selection for Approximate Query Processing. SIGMOD Conference 2003: 539-550

[11] Gang Luo, Jeffrey F. Naughton, Curt Ellmann, Michael Watzke: Toward a Progress Indicator for Database Queries. SIGMOD Conference 2004: 791-802

[12] Surajit Chaudhuri, Vivek R. Narasayya, Ravishankar Ramamurthy: Estimating Progress of Long Running SQL Queries. SIGMOD Conference 2004: 803-814

[13] Gang Luo, Jeffrey F. Naughton, Curt Ellmann, Michael Watzke: Increasing the Accuracy and Coverage of SQL Progress Indicators. ICDE 2005: 853-864

[14] Surajit Chaudhuri, Raghav Kaushik, Ravishankar Ramamurthy: When Can We Trust Progress Estimators for SQL Queries? SIGMOD Conference 2005: 575-586

[15] Microsoft SQL Server 2005 Books Online: Limiting Result Sets by Using TABLESAMPLE. http://msdn2.microsoft.com/en-us/library/ms189108.aspx

[16] IBM DB2 Version 9 for Linux, UNIX, and Windows: Data sampling in SQL and XQuery queries. http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=/com.ibm.db2.udb.admin.doc/doc/c0010970.htm

[17] Rafi Ahmed, Allison Lee, Andrew Witkowski, Dinesh Das, Hong Su, Mohamed Zait, Thierry Cruanes: Cost-Based Query Transformation in Oracle. VLDB 2006: 1026-1036

[18] George Lumpkin, Hakan Jakobsson, Maria Colgan: Query Optimization in Oracle Database 10g Release 2. http://www.oracle.com/technology/products/bi/db/10g/pdf/twp_general_query_optimization_10gr2_0605.pdf. June 2005

[19] Oracle® Database SQL Reference 10g Release 2 (10.2) Part Number B14200-02. December 2005

[20] Wikipedia: Root Finding Algorithm: http://en.wikipedia.org/wiki/Root-finding_algorithm October 2006

[21] Wikipedia: False position method: http://en.wikipedia.org/wiki/False_position_method September 2006

[22] Surajit Chaudhuri, Rajeev Motwani, Vivek R. Narasayya: On Random Sampling over Joins. SIGMOD Conference 1999: 263-274

[23] Surajit Chaudhuri, Luis Gravano: Evaluating Top-k Selection Queries. VLDB 1999: 397-410

[24] Ihab F. Ilyas, Walid G. Aref, Ahmed K. Elmagarmid: Supporting top-k join queries in relational databases. VLDB J. 13(3): 207-221 (2004)

[25] Ihab F. Ilyas, Rahul Shah, Walid G. Aref, Jeffrey Scott Vitter, Ahmed K. Elmagarmid: Rank-aware Query Optimization. SIGMOD Conference 2004: 203-214

[26] TPC-H: http://tpc.org/tpch/

[27] Eugene Inseok Chong, Souripriya Das, George Eadon, Jagannathan Srinivasan: An Efficient SQL-based RDF Querying Scheme. VLDB 2005: 1216-1227

[28] Jayant R. Haritsa, Krithi Ramamritham: Real-Time Database Systems in the New Millenium. Real-Time Systems 19(3): 205-208 (2000)

[29] Krithi Ramamritham, Sang Hyuk Son, Lisa Cingiser DiPippo: Real-Time Databases and Data Services. Real-Time Systems 28(2-3): 179-215 (2004)

[30] Peter J. Haas: Large-Sample and Deterministic Confidence Intervals for Online Aggregation. SSDBM 1997: 51-63

[31] Peter J. Haas: Hoeffding inequalities for join-selectivity estimation and online aggregation. IBM Research Report RJ 10040, IBM Almaden Research Center, San Jose, CA, 1999