Optimizing Recursive Queries in SQL

Carlos Ordonez
Teradata, NCR
San Diego, CA 92127, USA
carlos.ordonez@teradata-ncr.com

ABSTRACT

Recursion represents an important addition to the SQL language. This work focuses on the optimization of linear recursive queries in SQL. To provide an abstract framework for discussion, we focus on computing the transitive closure of a graph. Three optimizations are studied: (1) Early evaluation of row selection conditions. (2) Eliminating duplicate rows in intermediate tables. (3) Defining an enhanced index to accelerate join computation. Optimizations are evaluated on two types of graphs: binary trees and sparse graphs. Binary trees represent an ideal graph with no cycles and a linear number of edges. Sparse graphs represent an average case with some cycles and a linear number of edges. In general, the proposed optimizations produce a significant reduction in the evaluation time of recursive queries.

1. INTRODUCTION

Recursion is a fundamental concept in computer science. Most data structures, like trees or lists, are recursive. Many search algorithms have a natural recursive definition. Despite its prominent importance, recursion was not available in SQL for a long time. But the ANSI '99 SQL standard [7] introduced recursion into SQL with syntactic constructs to define recursive views and recursive derived tables. More recently, SQL recursive mechanisms have become available in relational Database Management Systems. This work focuses on optimizing linear recursive queries [11] in SQL, which constitute a broad class of queries used in practice.

The article is organized as follows. Section 2 introduces definitions. Section 3 presents our main contributions. Section 4 presents experiments focusing on query optimization. Related work is discussed in Section 5. Section 6 concludes the article and discusses some directions for future work.

2. DEFINITIONS

We define a base table T as $T(i, j, v)$ with primary key (i, j) and v representing a numeric value. Table T will be used as the input for recursive queries using columns i and j to join T with itself. Let R be the result table returned by a recursive query, with a similar structure to T. Table R is defined as $R(d, i, j, v)$ with primary key (d, i, j), where d represents recursion depth, i and j identify result rows at each recursion depth and v represents an arithmetic expression (typically recursively computed). For practical reasons, we assume there exists a recursion depth threshold k. Let $R^{[k]}$ represent a partial result table obtained from $k-1$ self-joins having T as operand k times. The queries we are interested are of the form $R^{[k]} = T \bowtie T \bowtie \cdots \bowtie T$.

To provide an abstract framework for discussion we use graphs. Let $G = (V, E)$ be a directed graph with n vertices and m edges. An edge in E links two vertices in V and has a direction. An edge can represent a parent/child relationship or a road between two locations. Notice our definition allows the existence of cycles in graphs. A tree is a particular case of a graph, where there is a hierarchical structure linking vertices and there are no cycles. There are two common representations for graphs; one is called the adjacency list and the other one is called the adjacency matrix. In this article we use the adjacency list. The adjacency list representation of a graph is a set L of edges joining vertices in V. If there is no edge between two vertices then there is no corresponding element in the list. Each edge has an associated weight (e.g. distance, capacity or cost). A path is defined as a subset of E linking two vertices in V. Therefore, a row from table T represents a weighted edge between vertices i and j in list L. Table T has m rows (edges), $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, n\}$. We focus on computing the transitive closure of G. The transitive closure G^* computes all vertices reachable from each vertex in G and is defined as: $G^* = (V, E')$, where $E' = \{(i, j)$ s.t. there exists a path between i and $j\}$.

3. OPTIMIZING RECURSIVE QUERIES

In this section we present several optimizations for recursive queries. We start by explaining the SQL syntax provided by Teradata to define recursive views. We explain the basic evaluation algorithm for recursive queries. We then propose query optimization strategies. Our discussion focuses on computing the transitive closure of G. Most of our optimizations involve rewriting queries, changing order of evaluation of relational operations and defining index structures to get equivalent non-recursive SQL queries that can be evaluated in less time.

3.1 Recursive Views

The basic mechanism to define recursive queries in the

Teradata RDBMS is a recursive view. We omit syntax for an equivalent statement for derived tables (WITH RECURSIVE). A recursive view has one or more base select statements without recursive references and one or more recursive select statements. Recursion is given in the join in a recursive select statement, where the view name appears in the "FROM" clause. A join condition can be any comparison expression, but we focus on equality (i.e. natural join). To avoid long runs with large tables, infinite recursion with graphs having cycles or infinite recursion with an incorrectly written query, it is advisable to add a "WHERE" clause to set a threshold on recursion depth, that we call k (a constant). We call the statement without the recursive join the base step and we call the statement with the recursive join the recursive step. They may appear in any order, but for clarity purposes the base step appears first.

The following view computes the transitive closure of a graph G stored as an adjacency list in T with a maximum recursion depth $k = 8$. Columns i, j, v are qualified with the corresponding table/view name to avoid ambiguity. The view computes the length/cost v of each path, but it will be irrelevant for the transitive closure.

```
CREATE RECURSIVE VIEW R(d, i, j, v) AS (
  SELECT 1,i, j, v FROM T            /* base step */
  UNION ALL
  SELECT d + 1, R.i, T.j, R.v + T.v
  FROM R JOIN T ON R.j = T.i     /* recursive step */
  WHERE d < 8 );

CREATE VIEW TransitiveClosureG AS (
  SELECT DISTINCT i, j FROM R );
```

In general, the user can write queries or define additional views on R treating it as any other table/view. Recursive views in Teradata have several constraints. There must be no "group by", "distinct", "having", "not in" clauses inside the view definition. However, such clauses can appear outside in a query calling the view, leaving the optimization task open for the query optimizer. Recursion must be linear; non-linear recursion is not allowed (view name appearing twice or more times in the internal "from" clause). Recursive views cannot be nested. We will study the optimization of queries using the recursive view introduced above, that represents a linear recursive query.

3.2 Algorithm to evaluate a recursive query

The algorithm to evaluate a recursive query is straightforward. Let $R^{[s]}$ be the result table after step s, where $s = 1 \dots k$. The base step produces $R^{[1]} = T$. The recursive steps produce $R^{[2]} = T \bowtie T = R^{[1]} \bowtie T$, ..., and so on. In general $R^{[s]} = R^{[s-1]} \bowtie T$. Finally, $R = R^{[1]} \cup R^{[2]} \cup \cdots \cup R^{[k]}$. Since step s depends on step $s-1$ the query evaluation algorithm is sequential and works in a bottom-up fashion. If $R^{[s]}$ is ever empty, because no rows satisfy the join condition, then query evaluation stops sooner. The query evaluation plan is a deep tree with $k-1$ levels.

3.3 Query Optimization

We study three optimizations: (1) Early evaluation of row selection conditions, (2) deleting duplicate rows, (3) indexing base and result table for efficient join computation. Due to lack of space we include only a brief discussion of another optimization, used for aggregation queries.

Early evaluation of row selection conditions

Early evaluation of row selection conditions may be used when there is a "WHERE" clause specifying a filter condition on columns from R. When G has cycles the recursion may become infinite; this becomes a practical problem for many applications. Therefore, we emphasize the use of a "WHERE" clause because it is the only way to guarantee a recursive query will stop in general. The queries we study are of the form

```
SELECT i, j, v FROM R
WHERE  <condition>;
```

One of the general guidelines in traditional query optimization is to evaluate selection (σ) of rows and projection (π) as early as possible. The rationale behind such optimization is that a join ($\bowtie$) operation can operate on smaller tables reducing work. This optimization involves automatically transforming the given query into an equivalent query that is evaluated faster. This guideline also applies to recursive queries, but we distinguish two cases. The first case is given by a condition on the columns from the primary key of R other than d (i.e. i, j). The second case is given by a condition on non-key columns v or depth d, that change at each recursive step.

We explain the first case. If there is a "WHERE" condition on a column belonging to the primary key (i or j), and the column does not participate in the join condition then the "WHERE" condition can be evaluated earlier. In this manner each intermediate table is smaller. Let us recall the transitive closure view introduced in Section 3.1. If we only want vertices reachable from vertex 1 the following query gives the answer.

```
SELECT i, j
FROM transitiveClosureG
WHERE i = 1;
```

The clause "WHERE $i = 1$" can be evaluated earlier during the recursion. It can be evaluated at the base step and at each recursive step, with caution, as explained below. Then the earliest it can be evaluated is at the base step to produce a subset of T, stored in $R^{[1]}$. This optimization propagates a reduction in the size of all intermediate tables $R^{[s]}$. Then the base step of the recursive view SQL code, presented in Section 3.1, is rewritten as follows.

```
SELECT 1,i, j, v FROM T /* base step */
WHERE i = 1
UNION ALL ...
```

Evaluating "WHERE $i = 1$" in the recursive step is tricky. First of all, i must be qualified. Using "WHERE $T.i = 1$" would produce incorrect results because it would only include vertex 1. Observe the recursive step uses $T.i$ in the "WHERE" clause, but not on the projected columns. Conversely, it uses $R.i$ in the projected columns and not on the "WHERE" clause. Evaluating "WHERE $R.i = 1$" produces correct results because $R.i$ is not part of the join condition, but in this case it is redundant because the partial result table $R^{[s-1]}$, only contains rows satisfying $R.i = 1$, propagated from the base step. Therefore, in this case it is sufficient to evaluate selection on key i on the base step.

We discuss the second case. Row selection with general "WHERE" conditions on v is hard to optimize and conditions on d are easier to optimize. The corresponding "WHERE" clause may be pushed into both the base and recursive part depending on how v is computed. In general v changes after each recursive step. We distinguish two possibilities: v is recursively computed (with addition or product) or v is not recursively computed when it is an attribute of vertex i or vertex j.

If the filter condition is of type "WHERE $v \leq v_U$" and v is recursively incremented then the query can stop at some step. If all T rows satisfy $v > 0$ and v is incremented at each step then the query will stop. But if there exist rows such that $v = 0$ or $v < 0$ then the query may not stop. Only in the case that $v > 0$ for all rows and v increases monotonically can we evaluate "WHERE $v \leq v_U$" at each recursive step. By a similar reasoning, if the condition is $v \geq v_L$ and $v > 0$ in every row then the query may continue indefinitely; then "WHERE $v \geq v_L$" cannot be evaluated at each recursive step. The transitive closure will eventually stop when the longest path between two vertices is found if there are no cycles, but it may produce an infinite recursion if there are cycles. If v is not recursively computed then v may increase or decrease after each recursive step; then it is not possible to push the "WHERE" clause because discarded rows may be needed to compute joins.

We can think of d as a particular case of v. Depth d monotonically increases at each recursive step since it is always incremented by 1. If the filter expression is "WHERE $d \leq k$" then this sets a limit on recursion depth and then query evaluation constitutes an iteration of at most k steps; this is the case we use by default because the recursion is guaranteed to stop. If the condition is of type $d \geq k$ then recursive steps may continue beyond k, perhaps indefinitely; we assume no recursive view is defined with such condition. Also, the clause "WHERE $d \geq k$" cannot be evaluated at early recursive steps because it would discard rows needed for later steps.

Deleting duplicate rows

Consider the problem of computing the transitive closure of G, but we are not interested in v, the weight/distance of each path; we just want to know all vertices that are reachable from each vertex. Refer to the recursive view given in Section 3.1.

SELECT DISTINCT i, j
FROM R;

Query evaluation is affected by how connected G is. If G is complete then there are $O(n)$ paths for each pair of vertices. If G is dense then there are probably two or more paths between vertices. This will produce duplicate rows that in turn will increase the size of partial tables after each recursive step. On the other hand, if G is sparse then there are fewer paths with less impact on join performance. In particular, if G is a tree there is only one path between pairs of vertices resulting in good join performance without using this optimization.

We optimize recursive queries by deleting duplicate rows at each step. If there are duplicate rows in T for any reason pre-filtering them reduces the size of the table from the base step. If there are no duplicate rows this optimization has no effect on table sizes. Applying this optimization the equivalent query used to evaluate the base step is:

SELECT DISTINCT 1,i, j FROM T /* base step */

The following equivalent query eliminates duplicates within one recursive step.

SELECT DISTINCT $d + 1, R.i, T.j$
FROM R JOIN T ON $R.j = T.i$ /* recursive step */
WHERE $d < k$

Assume G is a complete graph. If no optimization is done each recursive step s produces n^{s+1} rows. Therefore, time complexity is $O(n^{k+1})$ without optimization and $O(kn^3)$ with optimization at maximum depth k. This produces a significant speedup.

It is important to observe that another "SELECT" with DISTINCT on R at the end is required to get all distinct rows regardless of depth.

Aggregations queries on the recursive view can be optimized in a similar manner to queries with "SELECT DISTINCT". The grouping clause and the aggregate function can be evaluated at the base step and at each recursive step producing a reduction in size on all intermediate tables. In the case of the transitive closure this optimization is applicable only when rows are grouped by i and j.

Indexing base and result table for efficient join

We assume a recursion depth threshold k. If the partial result table $R^{[s]}$ is empty the recursion stops sooner at step $s < k$. The indexing schemes explained below are defined based on two facts. (1) The base table T will be used as a join operand $k-1$ times. (2) The result table R will be used as join operand $k - 1$ times, but selecting only result rows from the previous recursive step. Recall $R = \cup_s R^{[s]}$.

We propose two schemes to index the base table T and the result table R computed from the recursive view. Scheme 1 involves defining one index for T and one index for R based on the recursive join condition. In our two problems the join expression is $R.j = T.i$. Therefore, T has an index on (i) and R has an index on (j) allowing non-unique keys in both cases. Scheme 2 defines one index for T and one index for R based on their respective primary keys. That is, T is indexed on (i, j) and R is indexed on (d, i, j). The explanation behind scheme 1 is that T and R are optimally indexed to perform a hash join based on $R.j = T.i$. But having many rows satisfying the condition for each value of i may affect join performance because of hashing collisions. On the other hand, having a few rows (in particular one or zero) satisfying the join condition can improve hash join performance. In scheme 2 each recursive join cannot take advantage of the index because the join condition differs from the indexed columns, but each row can be uniquely identified efficiently. The optimizer uses a merge join making a full scan on both table R and table T at each step. However, only rows from $R^{[s]}$ are selected before the join.

3.4 Practical Issues

In general, a Cartesian product returns a large table, whose size is the product of the sizes of the tables involved. A Cartesian product appearing in a recursive view will be even worse since result size will grow fast as recursion depth grows. Most times this is caused by a user error because there is a missing join condition or the condition is not correctly written. This supports the idea of always setting a recursion depth threshold (k).

G type	m edges	cycles	complexity case
binary tree	$n-1$	N	best
sparse	$4n$	Y	average

Table 1: Characteristics for each type of graph G.

n	k	G: binary tree		G: sparse	
		opt=N	opt=Y	opt=N	opt=Y
32	2	2	2	3	2
32	4	3	3	4	2
32	8	4	3	5	3
32	16	5	5	8	5
64	2	2	2	2	2
64	4	3	2	4	2
64	8	4	3	6	3
64	16	5	5	8	5
128	2	3	2	2	2
128	4	3	2	4	2
128	8	5	3	6	4
128	16	6	5	8	5

Table 2: Early row selection. Times in seconds.

4. EXPERIMENTAL EVALUATION

In this section we present experiments on an NCR computer running the Teradata RDBMS software V2R6. The system had four nodes with one CPU each, running at 800 MHz, 40 AMPs (parallel virtual processors), 256MB of main memory and 10 TB of disk space. We used the data sets described below. Each experiment was repeated five times and the average time measurement is reported.

We study two broad query optimization aspects with two types of graphs: binary trees and sparse graphs The first set of experiments evaluates the impact of each optimization leaving the other optimizations fixed. The second set of experiments shows scalability varying the two most important parameters: n, the number of vertices in G and k, the maximum recursion depth. Due to the intensive nature of recursive queries all optimizations are turned on by default. Otherwise, several recursive queries, even on small data sets, could not be completed in reasonable time. We shall see query evaluation time depends on the type of graph.

We evaluate optimization strategies for recursive queries with synthetic data sets. We generated graphs G of varying number of vertices (n) and varying number of edges (m) to get different types of graphs. Each edge becomes a row in table T. Therefore, $m = |T|$. Two types of graphs were used. To evaluate the best case we used balanced binary trees; where G has $n-1$ edges (i, j) $(j = 1 \ldots n, i = j/2$) and no cycles; the number of rows grows linearly as n increases, $m = n - 1 = O(n)$. To evaluate an average case we used sparse graphs with 4 random edges per vertex; the number of rows grows linearly as n increases, $m = 4n = O(n)$. Data sets characteristics are summarized in Table 1.

4.1 Impact of Each Optimization

Early evaluation of row selection

We show experiments studying the performance gained by performing selection of rows as early as possible. The queries are based on the following query at different recursion depths, $k \in \{2, 4, 8, 16\}$. For this particular query row selection can be evaluated in the base step, as explained before.

SELECT d, i, j FROM transitiveClosure$G$
WHERE $i = 1$ AND $d \leq k$;

Table 2 shows the effect of early row selection turning the early row selection optimization on (Y) and off (N). In general, the gain in performance for small recursion depths (2 or 4) is marginal or zero. Differences come up with deeper recursion levels. For binary trees the gain in performance is small; there is an average difference of one second; times scale linearly in both cases. It is interesting that for the largest tree the difference in times becomes smaller. For sparse graphs the gain in performance is higher; time differences are around 2 seconds. Nevertheless, times scale linearly with and without this optimization. For larger sparse graphs queries run in half the time when the optimization is applied. We conclude this optimization is valuable in all cases, but becomes more important for highly connected graphs when recursion depth is high.

The impact of this optimization will depend on the selectivity of the condition being pushed, like in non-recursive queries, but combined with recursion depth. A highly selective filter condition that can be pushed into the base step will significantly improve evaluation time. Selecting rows in a binary tree that correspond to leaves will evidently produce a great evaluation time decrease since recursion will stop immediately, but selecting rows corresponding to upper nodes with many children will produce smaller tables, but recursion will go deep anyway. On the other hand, if G is highly connected then cycles will force the query to be evaluated at many recursion depth levels, but the sizes of intermediate results will decrease, producing again an important improvement.

n	k	G: binary tree		G: sparse	
		opt=N	opt=Y	opt=N	opt=Y
4	2	2	2	5	4
4	4	3	3	6	6
4	8	3	3	9	7
4	16	3	3	1951	10
8	2	2	2	5	5
8	4	3	3	7	7
8	8	3	3	12	10
8	16	3	3	*	11
16	2	2	2	5	5
16	4	3	2	6	6
16	8	3	4	15	7
16	16	3	4	*	10

Table 3: Deleting duplicates. Times in seconds.

Deleting duplicates

The next set of experiments studies the effect of deleting duplicate rows after each step for the transitive closure. The queries are based on the transitive closure view introduced in Section 3.1. The graphs used in these experiments are small, but queries become very demanding as recursion depth grows, as we explain below. The optimizer performs a sort operation to eliminate duplicates whenever the "DIS-

n	G	Indexing scheme	
	type	Indexing for Join	Index on PK
32	binary tree	4	6
64	binary tree	4	6
128	binary tree	4	5
256	binary tree	4	5
512	binary tree	4	5
1024	binary tree	4	5
16	sparse	9	8
32	sparse	7	8
64	sparse	7	9
128	sparse	8	9
256	sparse	12	16
512	sparse	32	25

Table 4: Indexing schemes. Times in seconds.

TINCT" keyword appears. Binary trees are shown for completeness since this optimization has no impact on them.

Table 3 summarizes results. The "opt" header indicates if the optimization is turned on (Y) or off (N). The entries marked with * mean the query could not be completed within one hour and then it had to be interrupted. The first general observation is that the transitive closure problem becomes intractable at modest recursion levels even with the small graphs studied. For binary trees times are similar with and without this optimization; there was only a single case where this optimization produced better times for binary trees. This is explained by the fact that there is at most one path between vertices and the overhead from the sorting process. In general, for sparse graphs and $n \geq 8$ times are better. Time growth is minimal for binary trees when k or n grow. Time measurements grow fast when this optimization is not used for sparse graphs. On the other hand, when duplicates are deleted times grow slowly as recursion goes deeper or data set size increases for sparse graphs. We conclude that duplicate rows should be deleted from intermediate tables whenever that does not affect the correctness of results or the semantics of the query.

Indexing

The following experiments compare the two indexing schemes introduced in Section 3.3, based on the join condition and primary keys, respectively. Default recursion depth is $k = 8$. Table 4 summarizes results. In general, the index optimized for hash-joining T and R provides best performance when G is a tree or a sparse graph; this confirms recursion is efficient with sparse graphs. However, as G becomes more connected collisions affect hash join performance. For a sparse graph with $n = 512$ indices on the table primary keys provide best performance. The trend indicates the difference in performance is not significant. We conclude that in most cases, enhanced indexing for hash-join provides best performance. However, there will be some performance loss when G is highly connected.

4.2 Scalability

The following experiments show scalability varying n and k with large data sets, using all optimizations except early row selection. The first goal is to evaluate time growth as m, n or k increase. The second goal is to know what problem characteristics are more critical for performance. The number of vertices n was chosen based on m and how connected G was. Therefore, graphs for binary trees have many vertices and sparse graphs have fewer vertices.

Graph size

The two left graphs on Figure 1 show time growth as n varies. Binary trees have a default recursion depth $k = 16$, whereas sparse graphs use $k = 8$. Notice n is much larger for binary trees; numbers shown must be multiplied by 1024 for correct interpretation. For binary trees time grows in a linear fashion. In sparse graphs time grows more rapidly in a super-linear fashion for both problems; the time difference becomes more significant as n grows. These experiments indicate graph size can become a serious performance problem for highly connected graphs.

Recursion depth

The right two graphs on Figure 1 show time growth as recursion depth k varies with a default data set size of $n = 131,072 = 128K$ for balanced binary trees and $n = 2,048$ for sparse graphs. For binary trees time growth is linear and towards the end time stops growing, confirming the fact that G is a balanced binary tree because the partial result table becomes empty. For sparse graphs time grows rapidly on the lower end, but gradually becomes linear as recursion depth approaches $k = 8$. This can be explained by the fact that each recursion level produces an increasingly more connected graph until it stabilizes. These experiments let us conclude that recursion depth will produce a linear increase in time for sparse graphs. Recursion depth in dense graphs may not be a significant performance problem, provided it is bounded and all optimizations are turned on.

5. RELATED WORK

SQL is the standard language used in relational databases. SQL has evolved significantly since its introduction [1], but it took a long time to have a standard syntax and semantics for recursion [7]. Research on extending SQL to compute recursive queries includes [8, 9]. In [8] an operator is proposed to allow recursion in a similar manner to Datalog. The evaluation algorithm works incrementally in a bottom-up fashion, in a similar manner to ours. The approach in [9] relies on creating tables that can contain a set of values in one column, thereby violating first normal form.

Efficient evaluation of recursive queries is a classic and broad topic in the database literature. Most work has been theoretical or in the context of deductive databases. Some approaches assume there are no cycles in the underlying graph, which sometimes is not a practical assumption (e.g. a geographical database); we have shown our optimization strategies also work on graphs with cycles. Important challenges exhibited by recursive queries are summarized in [2, 3, 4, 11]. Optimization by reducing redundant intermediate results is studied in [6]; our optimization on duplicate deletion is similar. Maintenance of recursive views through incremental insertions or deletions for the transitive closure is studied in [4]. Indexing for evaluation of recursive queries based on semi-naive or logarithmic evaluation algorithms, is studied in [10]. Our enhanced indexing scheme is somewhat similar to that proposed in [10], but we have shown that in some cases the alternative indexing scheme, based on primary keys, is better. Also, our enhanced indexing scheme is

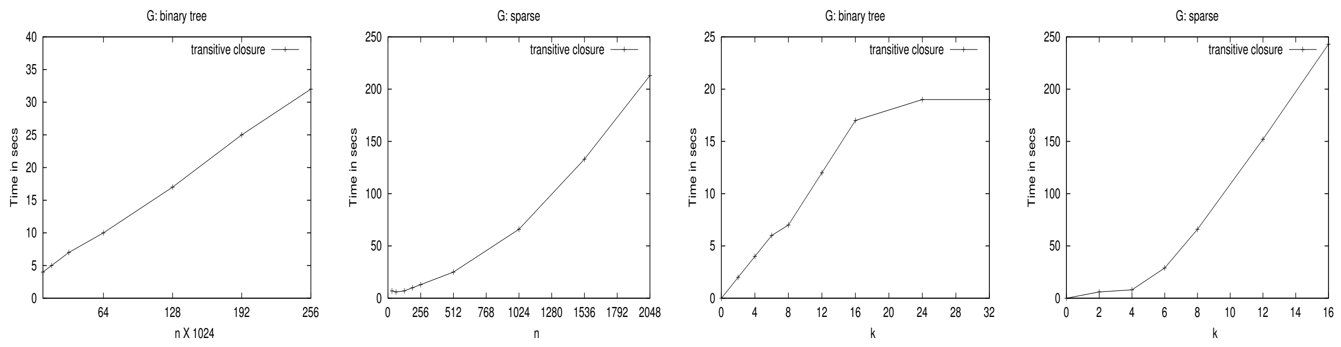

Figure 1: Scalability: Time growth varying n (graph size) and k (recursion depth).

tailored to perform hash-based joins storing required tuples on the same logical address. The authors of [12] study recursive queries when the underlying graph has cycles. Computing the transitive closure of a graph has received significant attention. An early work on the relationship between the transitive closure and matrix powering is [5].

6. CONCLUSIONS

We studied the optimization of linear recursive queries in SQL, which represent a broad class of useful queries based on recursion. Graphs were used to provide an abstract framework and were represented by a table having one row per edge. We focused on computing the transitive closure of a graph. Three query optimizations were analyzed, including early evaluation of row selection conditions, deleting duplicate rows and enhanced indexing for join computation. Experiments studied the individual impact of each optimization and scalability of recursive queries. Graph connectedness, recursion depth and data set size, were the main performance factors analyzed by the experiments. Two types of graphs were used to study the impact of each query optimization: balanced binary trees and sparse graphs. Early evaluation of row selection had an impact on performance for both binary trees and sparse graphs. A highly selective filter condition that can be pushed into the base step, significantly improves evaluation time. Deleting duplicate rows turned out to be an essential optimization to get results in reasonable time for sparse graphs due to cycles. Recursion depth significantly impacted evaluation time of queries on sparse graphs when duplicate rows were not deleted. Having a non-unique index based on the recursive view join expression for each table, was the best indexing scheme for binary trees and sparse graphs. Data set size and recursion depth had a strong impact on performance if the graph was sparse. We evaluated scalability using all optimizations except early row selection. In general, times for queries on binary trees scale linearly as graph size increases. Time on sparse graphs increases quadratically as graph size increases. Time scales linearly as recursion depth grows.

There are several aspects for future work. We want to study queries on unbalanced trees, non binary trees and lists. Composite keys may require special join optimizations. Some queries on highly connected graphs may produce too many rows; for instance, showing all potential paths between pairs of vertices. Evaluation of row selection needs to be further optimized for queries where the filter condition cannot be evaluated in the base step or only at the end of the recursion. Some queries may be evaluated by an algorithm with less recursive steps than those required by the sequential algorithm used in this work. Indexing can be improved based on the fact that the result table is incrementally appended and only one partial table is needed at a given time.

7. REFERENCES

[1] D.D. Chamberlin and R.F. Boyce. SEQUEL: A structured English query language. In *ACM SIGMOD Workshop*, pages 249–264, 1974.

[2] S. Cosmadakis. Inherent complexity of recursive queries. In *ACM PODS Conference*, pages 148–154, 1999.

[3] S. Dar, R. Agrawal, and H.V. Jagadish. Optimization of generalized transitive closure queries. In *ICDE Conference*, pages 345–354, 1991.

[4] G. Dong and J. Su. Incremental maintenance of recursive views using relational calculus/SQL. *ACM SIGMOD Record*, 29(1):44–51, 2000.

[5] M.J. Fischer and A.R. Meyer. Boolean matrix multiplication and transitive closure. In *IEEE FOCS Conference*, pages 129–131, 1971.

[6] J. Han and L.J. Henschen. Handling redundancy in the processing of recursive database queries. In *ACM SIGMOD Conference*, pages 73–81, 1987.

[7] ISO-ANSI. *Database Language SQL-Part2: SQL/Foundation.* ANSI, ISO 9075-2 edition, 1999.

[8] K. Koymen and Q. Cai. SQL*: a recursive SQL. *Inf. Syst.*, 18(2):121–128, 1993.

[9] V. Linnemann. Non first normal form relations and recursive queries: An SQL-based approach. In *IEEE ICDE Conference*, pages 591–598, 1987.

[10] P. Valduriez and H. Boral. Evaluation of recursive queries using join indices. In *Expert Database Systems*, pages 271–293, 1986.

[11] M.Y. Vardi. Decidability and undecidability results for boundedness of linear recursive queries. In *ACM PODS Conference*, pages 341–351, 1988.

[12] C. Wu and L.J. Henschen. Answering linear recursive queries in cyclic databases. In *FGCS*, pages 727–734, 1988.