Optimization of Linear Recursive Queries in SQL

Carlos Ordonez
University of Houston
Houston, TX 77204, USA

***Abstract*—Recursion is a fundamental computation mechanism which has been incorporated into SQL. This work focuses on the optimization of linear recursive queries in SQL. Query optimization is studied with two important graph problems: computing the transitive closure of a graph and getting the power matrix of its adjacency matrix. We present SQL implementations for two fundamental algorithms: Seminaive and Direct. Five query optimizations are studied: (1) Storage and indexing; (2) early selection; (3) early evaluation of non-recursive joins; (4) pushing duplicate elimination; (5) pushing aggregation. Experiments compare both evaluation algorithms and systematically evaluate the impact of optimizations with large input tables. Optimizations are evaluated on four types of graphs: binary trees, lists, cyclic graphs and complete graphs, going from best to worst case. In general, Seminaive is faster than Direct, except for complete graphs. Storing and indexing rows by vertex and pushing aggregation work well on trees, lists and cyclic graphs. Pushing duplicate elimination is essential for complete graphs. Early selection with equality predicates significantly accelerates computation for all types of graphs.**

Index terms: Recursive query, SQL, query optimization, transitive closure

I. Introduction

Recursion is fundamental in computer science. Most data structures, like trees or lists, are recursive. Most importantly, 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 introduced recursion into SQL with syntactic constructs to define recursive views and recursive derived tables. This article studies the optimization of linear recursive queries [24], [26] in SQL, which constitute a broad class of queries used in practice [2], [3], [17], [19]. Typical problems solved by linear recursive queries include parent/child relationships, path computations in a graph and bill of materials. Linear recursive queries have many applications in relational databases. Consider an ancestor/descendant example, with a table containing employee/manager information with the employee id the employee id of the manager. Examples are "who are all the employees that are managed directly or indirectly by person X?" or "is person X under person Y in the organization?". Suppose we have a table relating pairs of parts in a manufacturing environment where one column identifies one part and the second column corresponds to a subpart in a hierarchical fashion; this is the so-called bill of materials example. Examples are "list all subparts of part X" and "how many subparts does part X have two levels below?". Assume there is a geographical table with locations where each row indicates there exists a road (with distance as an attribute) between two locations. Examples are "which is the shortest path between X and Y?", "how many different routes are there between X and Y?", "what is the distance between X and Y?" or "what locations cannot be reached from X?.

Although recursive query optimization has been extensively studied in the past, mostly in deductive databases [3], [9], [19], [20], [24] and to a lesser extent in relational database systems [2], [5], [6], [11], [13] there is no recent work that studies the optimization of linear recursive queries in SQL. Most research has proposed complex algorithms using sophisticated data structures. Instead, our work studies how to optimize recursive queries with existing storage organization and indexing mechanisms and relational algebra transformations. Thus this paper revisits the classical problem of optimizing linear recursive queries, but focusing on SQL.

This is a summary of contributions. We present implementations in SQL of two classical algorithms to evaluate linear recursive queries: Seminaïve [3] and Direct [2]. We study the optimization of SPJA (selection-projection-join-aggregation) queries, following traditional query optimization principles [7]. SPJA queries represent the most common and general queries in a relational DBMS. Specifically, five query optimizations are studied: (1) Storage and indexing of input, intermediate and result tables; (2) early selection of rows by pushing predicates; (3) early or late evaluation of non-recursive (external) joins; (4 pushing duplicate and cycle elimination into intermediate recursive steps; (5) pushing aggregation through recursion. We also study how to improve performance when there are deep recursion levels, many duplicate rows and cycles. We perform a systematic experimental evaluation with large tables storing graphs with different structure and levels of connectivity.

The article is organized as follows. Section II introduces definitions and examples. Section III presents SQL implementations for two well-known recursive query evaluation algorithms. Section IV studies query optimization. Section V presents experiments focusing on query optimization and time complexity. Section VI discusses related work. Section VII presents the conclusions.

II. Definitions

A. Basic Definitions

To provide a mathematical framework for discussion we use graphs. Let $G = (V, E)$ be a directed graph with n

http://doi.ieeecomputersociety.org/10.1109/TKDE.2009.83

vertices and m edges. An edge in E links two vertices in V and has a direction. Notice our definition allows the existence of cycles in graphs. For instance, an edge can represent a parent/child relationship or a road between two locations. There are two common representations for graphs; one is called the adjacency list and the other one is called the adjacency matrix. 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. The adjacency matrix is an $n \times n$ binary matrix A, where A_{ij} represents an edge from vertex i to vertex j and is 1/0 indicating presence/absence of an edge. If G is an undirected graph then A is a symmetrical matrix.

Graph G is stored on table T (as an adjacency list) and the result of the recursive query is stored on table R (to be defined below in relational algebra and in SQL). Let the base table T be defined as $T(i, j, p, v)$ with primary key (i, j), p representing a path count and v representing a numeric value. Table T is 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, defined as $R(d, i, j, p, v)$ with primary key (d, i, j), where d represents recursion depth, i and j identify an edge at some recursion depth, p is a count (number of paths) and v represents a numeric value (typically recursively computed). Columns p and v are used to count paths between vertices and compute path lengths, respectively; we include them in both T and R to have consistent definitions and queries. A row from table T represents a weighted edge in G between vertices i and j in list L, some value attribute of either i or j or an entry A_{ij} of the adjacency matrix A. Table T has m rows (edges), $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, n\}$. In table R, p counts the number of paths and v the value (e.g. distance, capacity) of a path between two vertices. For practical reasons, we assume there is a recursion depth threshold k.

B. Problems

In this article we study queries of the form:

$$R = R \cup (R \bowtie T). \tag{1}$$

In Equation 1 the result of $R \bowtie T$ gets added to R itself. Since R is joined once with T recursion is linear. Table R is joined with T based on some comparison between $R.j$ and $T.i$. The most common predicate is based on equi-join $R.j = T.i$ (finding connected vertices). Within linear recursive queries the most well-known problem is computing the transitive closure of G, which accounts for most practical problems [2]. We focus on computing the transitive closure of G and the power matrix A^k in SQL. Both problems are similar, but their solution with relational queries is slightly different.

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. exists a path between i and $j\}$. That is, G^+ is a new graph with the same vertices, but new edges representing connectivity between two vertices.

The power matrix A^k (A multiplied by itself k times) contains the number of paths of length k between each pair of vertices, defined as: $A^k = \Pi_{i=1}^{k} A$. The power matrix can answer questions like: given a part, how many subparts does it have and what is their total cost?, how many paths are there between two cities below a certain distance and number of intermediate cities visited?, how many flights with no more that two stops are there between city X and city Y, and which one is the cheapest?

C. Recursive Views

In this article we focus on optimizing the SQL recursive view introduced below. Our discussion is based on tables T and R. The standard mechanisms to define recursive queries in the DBMS is a recursive view. We omit syntax for an equivalent SQL construct for derived tables. A recursive view has one or more base (seed) SELECT statements without recursive references and one or more recursive SELECT statements. Linear recursion is specified by a join in a recursive select statement, where the declared view name appears once in the "FROM" clause. In general, the recursive join condition can be any comparison expression, but we focus on equality (i.e. equi-join). To avoid long runs with large tables, infinite recursion with cyclic graphs or infinite recursion with an incorrectly written query, it is advisable to add a "WHERE" clause to set a threshold on recursion depth (k, a constant). The statement without the recursive join is called the base step (also called seed step [3], [15]) and the statement with the recursive join is termed the recursive step. Both steps can appear in any order, but for clarity the base step appears first.

We define recursive views for the two problems introduced in Section II. The following view R computes the transitive closure of a graph G stored as an adjacency list in T with a maximum recursion depth k. Columns i, j, p, v are qualified to avoid ambiguity. The view computes the length/cost v of each path. R is the fundamental linearly recursive view.

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

Based on R, the transitive closure (TC) G^+ is computed as follows. We consider DISTINCT as an optional clause to eliminate duplicates.

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

The power matrix (PM) view, based on R, returns $A, A^2, \ldots, A^k$ and the second statement returns A^k only:

```
CREATE VIEW PM AS (
  SELECT d,i,j,sum(p) AS p,max(v) AS v FROM R
  GROUP BY d,i,j );
SELECT * FROM PM WHERE d=k;      /* A^k */
```

This SQL code computes the total number of paths (sum(p)) at a certain depth (d) and the maximum length among all paths, with respect to v (max(v)). PM can be applied as follows. For

T

i	j	p	v
1	2	1	2
1	3	1	1
1	4	1	3
2	3	1	3
3	5	1	1
4	5	1	2
5	2	1	4

TC

i	j
3	2
3	5
4	2
4	3
4	5
5	2
5	3

PM

i	j	p	v
1	3	1	5
1	5	2	5
2	5	1	4
3	2	1	5
4	2	1	6
5	3	1	7

Fig. 1. Example: Input table T for G with $n = 5, m = 7$, TC ($i \geq 3, k = 5$) and PM for A^2 ($d = 2$).

instance, we can compute: the number of paths at a certain recursion depth d or across all depths (sum(p)), the total cost/distance v for all paths (sum(v)), or the longest/shortest length for all paths between two vertices with respect to their value v (max(v)). Thus, the power matrix provides additional information about R (i.e. about G^+).

In general, the user can write queries or define additional views on R treating it as any other table/view. Recursion must be linear; non-linear recursion is not allowed (i.e. view name R appearing twice or more times in the "FROM" clause). Recursive views have several constraints. There must be no "group by", "distinct", "having", "not in", "outer join", "order by" clauses inside the view definition. However, such syntactic constructs can appear outside in any query calling the view, leaving the optimization task open for the query optimizer. Recursive views cannot be nested to avoid indirect infinite recursion by mutual reference.

Figure 1 illustrates definitions. Table T stores all the edges in G. Notice G has a cycle (2,3,5). TC shows all vertices reachable from i excluding cycles; the longest path in this case has three edges. PM shows A^2; PM states there are two paths ($p = 2$) with two edges ($d = 2$) between vertices 1 and 5 whose maximum $v = 5$ (e.g. distance).

III. Algorithms to Evaluate Recursive Queries

We introduce SQL implementations of two classic algorithms to evaluate recursive queries: Seminaïve and Direct. Our proposed SQL implementations are based only on relational SQL queries and do not depend on any specific data structures or database system architecture.

A. Seminaïve Algorithm to Evaluate a Recursive Query

The standard algorithm to evaluate a recursive query comes from deductive databases and it is called Seminaïve [2], [3]. Seminaïve solves a broad class of recursive problems called fixpoint equations [2], [1]. Let R_d be the result table after step d, where $d = 1 \ldots k$. The base step produces $R_1 = T$. The recursive steps produce $R_2 = T \bowtie T = R_1 \bowtie_{R_1.j=T.i} T$, $R_3 = T \bowtie T \bowtie T = R_2 \bowtie_{R_2.j=T.i} T$, ..., and so on. In general $R_{d+1} = R_d \bowtie_{R_d.j=T.i} T$. For each recursive step the join condition is $R_d.j = T.i$. In each recursive step projection (π) is needed to make partial result tables union-compatible. Projection π computes $d = d + 1$, $i = R_d.i$, $j = T.j$, $p = R.p * T.p$ and $v = R_d.v + T.v$ at each iteration:

$$R_{d+1} = \pi_{d,i,j,p,v}(R_d \bowtie_{R_d.j=T.i} T). \quad (2)$$

To simplify notation from Equation 2 sometimes we do not explicitly write neither π nor the join condition between R and T: $R_{d+1} = R_d \bowtie T$. Finally, $R = R_1 \cup R_2 \cup \ldots \cup R_k$. If R_d becomes empty, because no rows satisfy the join condition, then query evaluation stops (i.e. R reaches a fixpoint [1], [22]). The query evaluation plan is a deep tree with $k-1$ levels. The tree has k leaves with operand table T and $k-1$ nodes with a $\bowtie$ between R_d and T. In practical terms the plan consists of a while loop of $k-1$ joins assuming bounded recursion by k.

The following SQL code implements Seminaïve. Cycles are filtered out to avoid double counting paths.

```
INSERT INTO R_1            /* base step */
 SELECT 1, i, j, v, 1 FROM T;
WHILE |R_d| > 0 DO
   INSERT INTO R_{d+1} /* recursive step */
   SELECT  d+1, R_d.i, T.j,R_d.p * T.p, R_d.v + T.v
   FROM R_d JOIN T ON R_d.j = T.i
   WHERE (R_d.i ≠ R_d.j) and d ≤ k ;
   INSERT INTO R SELECT * FROM R_{d+1};
   d = d + 1;
END;
```

B. Direct Algorithm to Evaluate a Recursive Query

Direct algorithms were adapted to evaluate a transitive closure query in a database system [2]. Such algorithms are called direct because their termination does not depend on path length; each vertex is processed once. We adapt the Warshall algorithm [25] (introduced as the fundamental direct algorithm [2]), to get the transitive closure of G in SQL based on a modified, but fast, binary matrix multiplication. There is another direct version with better I/O characteristics [2] when implemented in a high-level language like C, but it requires two passes with careful row blocking, making it less efficient when programmed in SQL. Assume we are manipulating G as the binary adjacency matrix A defined in Section II. The goal is to leave G^+ stored "in-place" in A. We first write the Direct algorithm with relational operators in order to program it with SQL queries. The expression $\pi(R_K)$ below makes the partial result union-compatible and performs the following computations: $d = R_i.d + R_j.d$, $i = R_j.i$, $j = R_i, j$, $p = R_i.p * R_j.p$ and $v = R_i.v + R_j.v$.

```
R = T
for K=1 to n do
    R_i = σ_{i=K}(R); R_j = σ_{j=K}(R)
    R_K = R_i ⋈_{R_i.i=R_j.j} R_j
    S = π_{d,i,j,p,v}(R_K)
    R = R ∪ S
end
```

There is no conditional logic to stop in the for loop. The algorithm traverses G, one vertex at a time. Old edges from R are replaced with new edges (i.e. they are left "in place" on R). The join between R_i (Kth row of A) and R_j (Kth column of A), for $K = 1 \ldots n$, links the predecessors of vertex K with

its successors; this is the crucial step. $\cup$ eliminates duplicates in case the edge (i, j) already exists. The number of paths computed as $p = R_i.p * R_j.p$ can be explained by the number of paths between $R_j.i, R_j.j$ and the number of paths $R_i.i, R_i.j$ when $R_j.i$ and $R_i.j$ are linked by a new edge. Compared to Seminaïve, Direct does not need k, but it needs n iterations. However, a path length threshold k may be included to avoid a large number of paths in dense graphs. If some vertex J does not have predecessors then there is no edge added by the join operator. Similarly, no edge is added if J has no successors. In Direct it is not possible to filter out cycles, because that would produce incorrect results.

The SQL code below implements Direct. n is computed as the maximum of i and j. Duplicates and embedded cycles are not eliminated by default. Nevertheless, the path count p is set to zero for cycles to make the count a tighter bound for paths with embedded cycles. To avoid getting a large number of edges in dense/cyclic graphs and to get the same output as Seminaïve the WHERE clause includes k (i.e. a maximum path length). Notice K and k have a different meaning.

```
SELECT
  max(CASE WHEN i ≥ j THEN i ELSE j END) AS n
FROM T;
INSERT INTO R  SELECT 1, i, j, v, 1 FROM T;
FOR K = 1 TO n DO
   INSERT INTO R  SELECT
    R_i.d + R_j.d, R_j.i, R_i.j
    ,CASE WHEN R_j.i ≠ R_i.j THEN R_i.p * R_j.p
     ELSE 0 END
    ,R_i.v + R_j.v
   FROM R R_i JOIN R R_j ON R_i.i = R_j.j
   WHERE R_i.i = K and R_j.j = K and R_i.d + R_j.d ≤ k ;
END;
```

IV. Query Optimization

We focus on optimizing queries based on the recursive view R, introduced in Section II-C. Such queries can include any valid SQL clause treating the recursive view R as an input table. In particular, some queries refer to the transitive closure computation (TC) and the rest refer to the power matrix (PM). We study five optimizations: (1) Storage and indexing for efficient join computation; (2) pushing aggregation; (3) pushing duplicate row elimination; (4) early evaluation of non-recursive joins; (5) early selection. Our proposed optimizations cover SPJA queries (select-project-join-aggregation), which are the most common queries in relational database systems. Storage and indexing is an essential aspect in a database system to efficiently process queries in SQL. All optimizations are presented and compared for the Seminaïve algorithm and the Direct algorithm. Optimizations involve defining adequate storage and indexing of T an R, rewriting queries through sound transformations and changing the order of evaluation of relational operations.

A. Storage and Indexing

We study the fundamental aspect about query processing: storage and indexing of table T and partial result tables R_i for efficient join computation. We consider graphs with and without cycles and of varying connectivity. We first discuss storage and indexing for Seminaïve and then for Direct.

For Seminaïve we study two schemes. Let k be the recursion depth threshold. Recall that if the partial result table R_d becomes empty then recursion stops sooner at step $d < k$. The storage and indexing schemes explained below are defined based on two facts. (1) Table T is used as a join operand $k - 1$ times. (2) Table R is used as join operand $k - 1$ times, retrieving rows from R_{d-1}. The final result table R is computed as $R = \cup_d R_d$ (see Section II). We now discuss the two schemes. Scheme 1 allows efficient retrieval by vertex. This scheme pays particular attention to the efficient retrieval of rows to evaluate the recursive join. For the two problems the join expression is $R.j = T.i$, where i and j are columns in both tables. Therefore, in T all edges with the same value for i are stored together (clustered) on the same logical address (block) and T has a clustered index on i. On the other hand, R has all rows corresponding to vertex j on the same logical address (block) and R has an index on j. That is, rows from T are clustered by i and rows from R are clustered by j. Scheme 1 allows non-unique join values corresponding to multiple edges to be retrieved in less I/Os in both cases. This storage and indexing scheme is efficient for acyclic graphs (e.g. trees and lists), but it can be inefficient for graphs whose transitive closure computation generates many duplicates. Scheme 2 enables efficient retrieval by edge. Edges having the same origin or destination vertex are not clustered. T has rows stored and indexed by both columns i, j, whereas R has rows stored and indexed by d, i, j. Scheme 2 can manage multiple edges more efficiently when G is highly connected, but it ignores the join condition for vertex connectivity. If there are multiple paths between i and j or there are cycles this scheme is more efficient to store and retrieve repeated edges. The justification 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. In such case the query optimizer uses a merge join making a full scan on both tables R and T at each step. However, only rows from R_d are selected before the join.

In a similar manner to Seminaïve, we now present two storage and indexing schemes for the Direct algorithm. For Direct R is joined with itself n times and T does not participate in those $n - 1$ joins. That is, T is only used for initializing R. Therefore, T impact on performance is marginal. Scheme 1 allows efficient retrieval by vertex. Therefore, R rows are clustered on the same logical address (block) by vertex and R has an index on i. In other words, all edges for the same vertex i are clustered together on secondary storage. Thus Scheme 1 allows efficient retrieval of all edges departing from one vertex i. During the n iterations we need to efficiently select the ith row and the $j$column from the binary matrix represented

by R. Therefore, Scheme 1 allows efficient retrieval by one subscript (i), but not when using the other one (j). In an analogous manner to Seminaïve, Scheme 2 is designed for efficient edge retrieval. During evaluation all repeated edges are stored on the same address. Each row in R has an index on vertices (i, j), allowing duplicates. Scheme 2 allows the efficient elimination of duplicate edges during evaluation since they are stored together by the previous iteration. In summary, Seminaïve considers storage and indexing for T, whereas T is not important for Direct. On the other hand, in Seminaïve R is joined with T multiple times, whereas in Direct R is joined with itself. Therefore, in Seminaïve R and T have index definitions optimized based on the joining condition $R.j = T.i$, whereas Direct has index definitions only to join R with itself based on a different join condition: $R_i.i = R_j.j$.

B. Early Selection

Pushing selection through recursion is one of the most well studied aspects in recursive query optimization [2], [18]. Here we revisit the problem. Early selection may be used when there is a "WHERE" clause specifying a filter condition on columns from R. Evaluating selection earlier is possible for Seminaïve, but not for Direct. We first explain how to evaluate selection predicates earlier for Seminaïve and then we explain why this is not possible for Direct.

When G has cycles the recursion can be infinite; this is a practical problem for many database applications. Therefore, we emphasize the use of a "WHERE" clause with $d \leq k$ 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, p, v FROM R WHERE <condition>;

By default the WHERE clause is evaluated at the end of recursion, producing correct results. 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, thus reducing work. This optimization involves transforming (rewriting) 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 d, p, v, 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 II-C. Suppose we only want vertices reachable from $i = 1$:

SELECT i, j FROM R
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. Therefore, 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_d. Then the base step of the recursive view SQL code, presented in Section II-C, is rewritten as follows:

SELECT 1,i, j, v /* base step */
FROM T WHERE $i = 1$

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_d, 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. This optimization cannot be applied to the next query:

SELECT d, i, j,sum(v) FROM R
GROUP BY d, i, j
WHERE $j = 1$;

The reason that hinders pushing the WHERE clause is because $R.j$ is part of the join condition $R.j = T.i$. Even further, "WHERE $T.j = 1$" cannot be evaluated neither on the base step nor on the recursive step.

A similar reasoning applies to more complex WHERE expressions. For instance, selecting a row/column from the power matrix A^k. Consider the query

SELECT d, i, j,sum(v) FROM R
GROUP BY d, i, j
WHERE $d = 10$ and $i = 1$ and $j = 1$;

This query can be evaluated more efficiently by filtering with "WHERE $T.i = 1$" in the base step of R and "WHERE $R.i = 1$" at each recursive step. However, "WHERE $R.i = 1$" cannot be pushed into the base step because it uses R; "WHERE $T.j = 1$" cannot be pushed either.

We discuss the second case, involving predicates on p, v. Row selection with general WHERE conditions on v is difficult to optimize, whereas conditions on d are easier to optimize. The corresponding WHERE clause may be pushed into both the base and recursive step depending on how v is computed. We distinguish two possibilities: v is recursively computed (with addition or product) or v is not recursively computed when it is a property of vertex i or vertex j (e.g. employee age).

If there is no recursion depth k and 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 we can 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. For instance, the power matrix may produce an infinite recursion for cyclic graphs selecting rows with "WHERE $v > 0$" and d not having a threshold k. 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" predicate because discarded rows may be needed to compute future joins.

We can think of d and p as particular cases of v. Depth d monotonically increases at each recursive step since it is always incremented by 1. The number of paths will be one without pushing aggregation and $p \geq 1$ when aggregation is pushed. The filter expression "WHERE $d \leq k$" sets a limit on recursion depth and then query evaluation iterates at most k steps; this is the case we use by default because recursion is guaranteed to stop. With a WHERE predicate $d \geq k$ recursive steps may continue beyond k, perhaps indefinitely; we assume no recursive view is defined with such condition. Also, "WHERE $d \geq k$" cannot be evaluated earlier because it would discard rows needed for future steps.

As mentioned earlier, it is not possible to evaluate selection predicates earlier for the Direct algorithm. Direct depends on the entire graph computed so far at each iteration. If a selection predicate is applied earlier it may filter out edges that may be needed by a future vertex (in the loop). To illustrate this limitation, assume G is a list with vertices connected in descending order with edges $\{(n, n-1),(n-1, n-2),..,(2, 1)\}$. Then G is analyzed in inverse order by Direct, as presented before. If we applied the selection "WHERE $i = n$" at every iteration we would be left with T at the end of the n iterations: no new edge would be added: $G^+ = T$. This is because the successor list of every vertex would be empty, given the filter condition. Therefore, the transitive closure would be incorrectly computed. On the other hand, if we applied the filter in some of the first iterations we would eliminate edges that belong to the path departing from $i = n$.

A cartesian product appearing in a recursive view can produce huge tables since R size will grow fast as recursion depth k grows. In general, this can be caused by a user error because there is a missing join condition or such condition is not correctly written. Also, dense and complete graphs may lead to infinite recursion since each partial result table is non-empty. These potential issues support always including a recursion depth threshold (k) in the WHERE clause.

C. Evaluation of Non-recursive Joins

This optimization is applicable when a query has a non-recursive join between the recursive view R and another table (describing properties of vertices or edges in G) or when there are non-recursive joins inside the view definition of R. Non-recursive joins can also be understood as external joins because they do not need to form part of the recursive definition and therefore they can be evaluated outside of the recursion. For graphs such queries are useful to get vertex properties. Suppose we want to join R with another table N ($N \neq T$) to get vertex names (e.g. given a city or product id we need their full name). Assume vertex names and other vertex properties are stored on table N, defined as $N(\underline{i}$,name).

There exist two strategies to get vertex names: (1) performing two joins between the final result table R and N to get names for i and j, without changing the recursive view definition; (2) creating a denormalized table T_N joining T with N and substituting T for T_N inside R. Both strategies sit at two extremes: perform join evaluation late (as given in the query) and earliest (rewritten by the query optimizer). Therefore, Strategy 1 is called late non-recursive join, and Strategy 2 is called early non-recursive join. We omit discussion on a third strategy that evaluates external joins at each iteration because it is straightforward and inefficient.

We now discuss the query plan for each strategy in more detail. Strategy 1 is the simplest: R is evaluated first and the external join is evaluated at the end between R and N. Strategy 1 follows traditional optimization guidelines: a join operation should be evaluated last. Strategy 2 represents the other extreme. In this case the join operation is pushed all the way through recursion and it is evaluated once, in the base (seed) step. In this case, the query plan is similar: we just substitute T for T_N. Strategy 2 defies common query optimization principles: a join operation is evaluated first. If referential integrity is violated or there is a selection predicate on N Strategy 2 is clearly the best choice since it filters out T rows before recursion.

Our discussion focuses on the Seminaïve algorithm, but this optimization applies in a similar manner to the Direct algorithm. We start by analyzing Strategy 1, which does not change R. Consider the following query to compute the transitive closure.

SELECT d, i,N_i.name,j,N_j.name
FROM R JOIN N AS N_i ON $R.i = N_i.i$
JOIN N AS N_j ON $R.j = N_j.i$;

After R is computed we just perform two joins to get each vertex name; N must be aliased to avoid ambiguity. The I/O cost for this query mainly depends on the size of R because N is comparatively smaller. N can be optimally indexed on i, but there are several indexing choices for R based on combinations of $\{d, i, j\}$, as seen before.

Strategy 1 gets names in a lazy manner, after R is computed. Since vertices names remain static (constant through recursion) we can optimize the query by performing an early non-recursive join *before* the base step to create a denormalized table T_N having vertex names for i and j. Then non-recursive joins pass through R and are avoided during recursion. Hence we call Strategy 2 early non-recursive join.

Strategy 2 may not always be more efficient than Strategy 1 (late non-recursive join). Strategy may be better when T_N has much bigger rows than T or there is also duplicate elimination. A wide denormalized table T_N with many property columns for i and j will impact I/O. Notice vertex names could be potentially retrieved from R_1 since they are available for every edge after the base step, but that would require joining T with R twice producing a nonlinear (quadratic) recursion. That is

not feasible for SQL queries restricted to linear recursion.

D. Pushing Duplicate Row Elimination

We consider cycle detection and eliminating duplicate cycles as a special case of this optimization.

We first discuss Seminaïve. 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 reachable from each vertex. Refer to the recursive view given in Section II-C. This query provides the answer.

Query efficiency is affected by how connected G is. If G is dense or complete then there are many paths for each pair of vertices. If G is cyclic 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 acyclic then there are fewer paths with less impact on join performance. In particular, if G is a tree there is only one or no path between pairs of vertices, resulting in good join performance without applying this optimization.

Pushing duplicate elimination works as follows: Duplicate rows are eliminated at each iteration, instead of doing it at the end of the recursion. Using DISTINCT or a count() aggregation grouping by i, j are equivalent. If there are duplicate rows in any intermediate step this optimization reduces the size of temporary tables. If there are no duplicate rows this optimization has no effect on table sizes. The impact of this optimization will depend on the type of graph. A last "SELECT" statement on R (with DISTINCT or GROUP-BY) is required at the end of recursion to get all distinct rows regardless of path length.

We now turn our attention to Direct. In the Direct algorithm duplicates can also be eliminated at each iteration. In SQL this can be done using GROUP-BY, DISTINCT or set containment. Duplicates can be efficiently eliminated exploiting an index on either (i, j) for the transitive closure of (d, i, j) for the power matrix. The impact of this optimization will depend on the type of graph, like Seminaïve.

We now explain how to manage cyclic graphs, which have multiple paths at deeper recursion levels. In the Seminaïve algorithm duplicate cycles (i.e. an edge (i, i)) are eliminated to avoid double counting paths. In such case existing cycles are not extended with new edges. However, shortest cycles are detected and are stored on R. On the other hand, for the Direct algorithm it is not straightforward to filter out cycles during iterations without producing incorrect results. This is due to the fact that a new edge may create an embedded cycle and such edge cannot be formed without the corresponding vertex otherwise. Therefore, SQL queries do not exclude the embedded cycle $R_i.i, R_i.j$ or $R_j.i, R_j.j$ when $i = j$, like the Seminaïve algorithm. However, duplicate cycles are eliminated to achieve a partial reduction in size.

E. Pushing aggregation

Recall the power matrix A^k is computed from R as

SELECT d, i, j, sum(p), max(v) FROM R
GROUP BY d, i, j WHERE $d = k$;

We are concerned with efficient evaluation of aggregation queries on the recursive view R. Our discussion is based on the power matrix problem, defined in Section II. Notice pushing aggregation can be used to eliminate duplicates (e.g. by simply computing count(*) for each group). Therefore, pushing aggregation is a generalization of pushing duplicate elimination. We propose evaluating the "group by" clause and the aggregate function at every iteration, when possible, instead of doing it at the end of the recursion. This optimization is applicable to all standard SQL aggregations(count(), sum(), min(), max()) and when the given "group by" clause includes both i and j. For Seminaïve the GROUP-BY clause is evaluated at each step, both base and recursive. On the other hand, this optimization can also be applied to Direct by pushing the GROUP-BY at each iteration, but it requires careful interpretation when applied in cyclic graphs. The equivalence between both the non-optimized and optimized queries results from the distributive laws [8] of arithmetic operations + and *.

For Seminaïve the equivalent query for the base step is:

SELECT 1 AS d, i, j, sum(p), max(v)
FROM T /* base step */
GROUP BY d, i, j;

In general the base step query produces no performance improvement if there are no duplicate keys (i, j) in T. The equivalent query evaluated at each recursive step is:

SELECT $d + 1$ AS d,$R.i, T.j$,
sum($R.p * T.p$) AS p,max($R.v + T.v$) AS v
FROM R JOIN T ON $R.j = T.i$ /* recursive step */
WHERE $R.d < k$
GROUP BY $d, R.i, T.j$;

Going back to the transitive closure, this optimization is applicable to the following query since it involves the primary key of R. For instance, consider the query computes the longest distance (based on v) between two locations at each depth. This is useful when there are two or more paths between locations.

SELECT d, i, j, max(v)
FROM R GROUP BY d, i, j;

This optimization can also be applied if the grouping is done on i, j but not d. In practical terms, this is the same case as having "group by" on all the primary key columns of R. The important fact is that each step uses the primary key of the partial aggregation table required, but a "group by" at then end is required anyway. Also, each recursive step must still store partial results at depth $d = 1 \ldots k$. In the following query the v maximum can be computed for every pair of vertices at each depth pushing "group by i, j". The final aggregation gets the maximum across all depths.

SELECT i, j, max(v)
FROM R GROUP BY i, j;

This optimization is not directly applicable when the grouping columns do not include (i, j), but it can be partially

applied using all grouping columns eliminating redundant rows (if any). For our two problems that means partially grouping by either i or j. Examples are computing the total sum of salaries of all employees under each manager or finding the most expensive/cheapest subpart of each part. Such computations require "carrying" the aggregated salary of each sub-employee or the aggregated subpart cost at each step for the future aggregation. Consider the query based on the modified transitive closure view R, using $v = T.v$ instead of $v = R.v + T.v$:

SELECT i, max(v) FROM R GROUP BY i;

Performing an early "GROUP BY i" would incorrectly eliminate rows with different paths from i to j. This would in turn hinder recursive joins on the condition $R.j = T.i$ and would return a number different of terms from those in the view. Therefore, early aggregation with "GROUP BY i" is not possible because intermediate vertices at each recursion depth are needed to perform the next recursive step. However, "GROUP BY i, j" can be evaluated at each step saving work by eliminating redundant rows; that is the case if there are two or more paths between i and j. Therefore, if the the query has "GROUP BY i" or "GROUP BY j" then "GROUP BY i, j" is pushed. This optimization is applicable to any distributive aggregation [8].

We now consider transitive closure, where we only want to know all vertices reachable from each vertex. Pushing aggregation has the desirable effect of eliminating duplicate rows at each step because each group represents a partition of the partial table R_d. Hence pushing aggregation can be used to eliminate duplicates and accelerate computation where there are many. In general a query optimizer evaluates "SELECT DISTINCT" with a different plan from "SELECT/GROUP BY". Selecting distinct rows are generally obtained performing a sort, whereas computing aggregation requires creating a temporary table indexed by grouping columns.

We now turn our attention to the Direct algorithm. For Direct we can evaluate the GROUP-BY at each iteration using a temporary table R_T to store the aggregation results and then replacing R with R_T, leaving the compressed table "in-place". This optimization reduces the size of R when there are multiple paths between vertices. We omit this SQL code. Evaluating aggregations with the Direct algorithm requires careful interpretation of results. We distinguish two main cases: acyclic and cyclic graphs. For acyclic graphs the Direct algorithm computes the exact number of paths p for any pair of vertices in the power matrix problem. If G is acyclic, but not a tree, it can be decomposed into trees. Therefore, pushing aggregation at each iteration may produce a reduction in the size of R. However, such reduction is not as significant as it can be for cyclic graphs. On the other hand, if G is cyclic then the Direct algorithm computes an upper bound on the number of paths between a pair of vertices if there is an embedded cycle. If there are no cycles along the path then p is accurate. This is an effect of analyzing V in some fixed order (e.g. $1 \ldots n$), which causes new edges to be added to R in a random order, producing embedded cycles. Then when G is cyclic and dense (resembles a complete graph) the number of paths in R will grow as K increases producing a lower number p for vertices analyzed earlier and a higher p for vertices analyzed later. Unfortunately, it is not straightforward to eliminate embedded cycles without keeping track of all paths (storing all intermediate vertices), which would render a different algorithm.

TABLE I
SUMMARY OF APPLICABILITY OF OPTIMIZATIONS.

Optimization	Seminaive	Direct
Storage and indexing	Y	Y
Early selection	Y	N
Non-recursive join	Y	Y
Push duplicate elimination	Y	Y
Push aggregation	Y	Y

Table I summarizes our five optimizations and their applicability to each algorithm. Notice duplicate elimination also includes cycle elimination.

V. EXPERIMENTAL EVALUATION

A. DBMS Software and Hardware

This section presents experiments on a computer running the Teradata DBMS V2R6. The system had one CPU running at 3.2 GHz, 4 AMPs (parallel virtual processors), 4 GB of main memory and 256 GB on disk.

B. Overview of Experiments

We perform an experimental evaluation with four types of graphs: trees, lists, cyclic and complete graphs, summarized in Table II. Each type of graph is described in detail below in Section V-C. The first set of experiments compares Seminaïve and Direct. The second set of experiments evaluates the impact of each optimization individually on simple queries. The third set of experiments evaluates all optimizations interacting together on complex queries. The fourth set of experiments analyzes scalability varying problem sizes: n, the number of vertices in G and k, the maximum recursion depth. Each experiment was repeated five times and the average time measurement in seconds is reported.

Due to the demanding nature of linear recursive queries we had to carefully set defaults for optimizations. Storage and indexing by vertex was used for trees, lists and cyclic graphs. Duplicate elimination and aggregation were not used by default for acyclic graphs; the rest of optimizations were query-dependent. Each experiment re-created the input table T to use the default storage scheme and avoid caching; all tables were read "fresh" from disk for each query, avoiding any caching by the DBMS. In summary tables the "opt" header indicates if the optimization is turned on (Y) or off (N). Table entries marked with * mean query evaluation could not end within one hour and then it had to be interrupted.

An SQL code generator was implemented in the Java language. The recursive view was unfolded by creating a script of SQL statements to evaluate it. The program had parameters to specify the input tables and columns, pick an algorithm and turn optimizations on/off. Query evaluation was performed

TABLE II
TYPE OF GRAPH G.

G	m edges	cyclic	time complexity
tree	$n-1$	N	best
list	$n-1$	N	good
cyclic	$2n$	Y	bad
complete	n^2-n	Y	worst

using temporary tables for each step populating each table with SELECT statements. Time measurements were obtained with SQL using timestamps for maximum accuracy.

C. Input Tables with Graphs

We study query optimization with synthetic graphs. Graphs G were generated by 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, $m = |T|$. Four 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, $m = n-1 = O(n)$. Lists represented a good case (slightly worse than trees) since G has no cycles, but G is an unbalanced tree. To evaluate a bad case we used cyclic graphs with 2 random edges per vertex; the number of rows grows linearly as n increases, $m = O(n)$, but the number of paths grows much faster as k increases. To evaluate the worst case we created complete graphs having two directed edges for every pair of vertices (one from i to j and another from j to i). In complete graphs the number of rows grows quadratically as n increases, $m = n(n-1) = O(n^2)$. For cyclic and complete graphs we avoided generating self-cycles (a loop from i to i), since that provided no insight. Data sets characteristics are summarized in Table II. The time complexity column describes the expected time growth when evaluating the query: the existence of cycles and a higher number of edges make R grow faster. In our tables we include an approximate value of m to make comparisons an interpretation easier (i.e. 100K instead of 99,999). Notice it is m, and not n, the actual size of T. In most experiments we duplicate m (2X) and multiply m by 10 (10X), starting at $m = 100K$, to understand scalability and complexity. In general, the largest graphs have around 1M edges, which represent a computationally intensive problem.

For binary trees and cyclic graphs A_{ij} entries equal to zero are not included in T producing an automatic performance improvement to compute the power matrix A^k. This is based on the fact that aggregations on the full matrix A with zero entries are equivalent to aggregations on a "lean" matrix version excluding zeroes. The absence of row $[i,j,v]$ from A means $A_{ij} = 0$. Otherwise, all computations for A^k on trees and cyclic graphs would require the same time as complete graphs. Likewise, for the transitive closure an absence of a row from T means the corresponding edge is not present, as explained in Section II.

TABLE III
SEMINAIVE VERSUS DIRECT FOR TRANSITIVE CLOSURE (TIMES IN SECONDS).

G	n	m	k	Seminaïve	Direct
tree	10K	10K	5	2	175
	100K	100K	5	23	2948
	500K	500K	5	75	*
	1M	1M	5	170	*
	2M	2M	5	341	*
	10K	10K	-	3	204
	100K	100K	-	96	*
	500K	500K	-	492	*
	1M	1M	-	1287	*
	2M	2M	-	*	*
list	100K	100K	5	23	*
	1M	1M	5	208	*
	2M	2M	5	297	*
	10K	10K	10	6	283
	100K	100K	10	94	*
	1M	1M	10	377	*
	2M	2M	10	719	*
	1K	1K	-	331	261
	10K	10K	-	*	*
cyclic	1K	2K	5	3	20
	10K	20K	5	37	1104
	100K	200K	5	2818	*
	1K	2K	10	28	324
	10K	20K	10	1834	*
	100K	200K	10	*	*
complete	100	10K	5	8	8
	200	40K	5	102	46
	300	90K	5	1033	357
	1000	1M	2	*	*

D. Seminaïve versus Direct

Table III compares both algorithms with all types of graphs at different recursion depths. The table includes the number of rows in T (m) and recursion depth (k) when used. For trees, lists and cyclic graphs we increment n 10-fold to identify important trends (recall $n = |V|$). For complete graphs it was not possible to increment n in the same manner because the number of edges grows $O(n^2)$. Therefore, for complete graphs we increment V linearly. For large cyclic graphs it was not possible to finish execution within one hour when $k > 10$. For complete graphs the situation was even worse since we could not go beyond $k = 5$, despite G being relatively small. Evidently, the issue was the exponential growth of edges (paths) as k grows.

The first general trend is that Seminaïve is faster than Direct, except for complete graphs and lists using unbounded k. Both algorithms are impacted by recursion depth; time growth becomes significant for cyclic and complete graphs, highlighting the challenge they represent. Since binary trees are balanced $k = O(\log_2(n))$ and then it s not necessary to bound k. Lists represent a deep recursion problem, where Seminaïve ends up being slightly slower than Seminaïve. There is a jump in time when k goes from 5 to 10 for cyclic graphs in Seminaïve and evaluation cannot end within one hour for Direct. Seminaïve is slower than Direct for complete graphs, but it is not an order of magnitude slower. However, the time trend indicates Seminaïve will always be slower with a gap widening as n grows.

TABLE IV
SEMINAIVE: STORAGE/INDEXING SCHEMES FOR TRANSITIVE CLOSURE (TIMES IN SECONDS).

G	n	m	$k=2$ Storage/index Vertex	$k=2$ Storage/index Edge	$k=4$ Storage/index Vertex	$k=4$ Storage/index Edge
tree	100K	100K	6	10	17	40
	200K	200K	9	31	32	97
	1M	1M	79	242	128	502
list	100K	100K	5	12	19	36
	200K	200K	9	27	27	79
	1M	1M	82	245	126	380
cyclic	10K	20K	2	2	19	16
	20K	40K	5	6	20	34
	100K	200K	16	34	96	199
complete	100	10K	25	10	86	16
	142	20K	83	27	202	75
	316	100K	1166	919	3539	2261
	1000	1M	*	*	*	*

TABLE V
DIRECT: STORAGE/INDEXING SCHEMES FOR TRANSITIVE CLOSURE (TIMES IN SECONDS).

G	n	k	Storage/indexing Vertex	Storage/indexing Edge
tree	4k	-	74	85
	8k	-	154	208
list	4k	10	82	92
	8k	10	182	214
cyclic	4k	5	107	141
	8k	5	284	476

TABLE VI
SEMINAIVE: EARLY SELECTION FOR TRANSITIVE CLOSURE (TIMES IN SECONDS).

G	n	m	$k=2$ N	$k=2$ Y	$k=4$ N	$k=4$ Y
tree	100K	100K	6	1	17	1
	200K	200K	9	2	32	2
	1M	1M	79	18	128	22
list	100K	100K	5	1	19	1
	200K	200K	9	2	27	2
	1M	1M	82	19	126	19
cyclic	10K	20K	2	1	19	1
	20K	40K	5	1	20	1
	100K	200K	16	3	96	3
complete	100	10K	10	1	16	1
	142	20K	27	1	75	1
	316	100K	1166	2	2261	2

TABLE VII
SEMINAIVE: NON-RECURSIVE JOIN EVALUATION FOR TRANSITIVE CLOSURE (TIMES IN SECONDS).

G	n	m	$k=2$ late	$k=2$ early	$k=4$ late	$k=4$ early
tree	100K	100K	16	12	52	28
	200K	200K	30	23	102	52
	1M	1M	154	114	509	273
list	100K	100K	16	9	42	28
	200K	200K	35	21	87	59
	1M	1M	147	90	352	245
cyclic	10K	20K	7	5	45	35
	20K	40K	15	7	75	39
	100K	200K	53	36	749	171
complete	100	10K	8	16	15	44
	142	20K	28	56	75	153
	316	100K	1101	*	*	*

E. Impact of each Optimization

Storage and Indexing

Table IV summarizes results for transitive closure. In general, storage by vertex and an index based on the join condition provides best performance for trees, lists and cyclic graphs; the gap widens as n increases. Storage and indexing by edge is better for small cyclic graphs and for complete graphs; the difference in performance is significant, but the gap narrows as n increases. Therefore, storage and indexing by vertex should be the default alternative (Scheme 1).

Table V compares storage/indexing for Direct. Recursion depth k is unbounded for trees, $k = 10$ for lists and $k = 5$ for cyclic graphs. Interestingly enough, the trend is the same as Seminaïve: row storage by vertex and an index on the join condition provides best performance.

Early Selection

We study the performance gained by performing selection (filtering) rows as early as possible. The queries are based on the transitive closure view with a selection predicate "WHERE $i = n/2$", where $n = |V|$. Such condition provides high selectivity for all types of graphs.

Table VI analyzes early selection. In all cases early row selection is faster. The explanation is that pushing selection works on a smaller table and it reduces the sizes of all intermediate tables (like traditional query optimization). For trees and lists the gain in performance is small at low k, but becomes significant at deep recursion depth k. For cyclic graphs this optimization makes Seminaïve almost two orders of magnitude faster and three orders of magnitude faster for complete graphs. Therefore, we conclude that this optimization is valuable in all cases. The impact of this optimization will depend on the selectivity of the condition being pushed, like in traditional SPJ queries, but combined with recursion depth. A highly selective filter condition that can be pushed into the base step will significantly improve evaluation time.

Early Non-recursive Joins

The following experiments compare the strategies proposed in Section IV to evaluate non-recursive joins with the transitive closure recursive view. We concentrate on comparing non-recursive join evaluation with late (Strategy 1) and early (Strategy 2) evaluation.

Table VII compares efficiency varying n and k. Table N had small records with i (integer) and a vertex description (char(10)). Such table simulates retrieving one property for each vertex (e.g. a city name, part description). For trees, lists and cyclic graphs early join evaluation is more efficient than late evaluation. On the other hand, late join evaluation becomes a winner for complete graphs. This may sound counterintuitive because these results state that it is better to evaluate a join operation first, rather than at the end like traditional SPJ queries. However, we must point that if table N has large records and several columns are needed this would hurt performance for early join because each recursive step

TABLE VIII
SEMINAIVE: PUSHING DUPLICATE AND CYCLE ELIMINATION FOR TRANSITIVE CLOSURE (TIMES IN SECONDS).

G	n	m	elimCycles	$k=2$ pushDup		$k=4$ pushDup	
				N	Y	N	Y
cyclic	10k	20K	N	4	3	29	21
	10k	20K	Y	3	2	23	22
	20k	40K	N	8	5	46	43
	20k	40K	Y	7	4	44	40
	100k	200K	N	40	37	318	274
	100k	200K	Y	42	35	316	241
complete	100	10K	N	41	6	*	16
	100	10K	Y	37	5	*	15
	142	20K	N	84	32	*	82
	142	20K	Y	102	25	*	80
	316	100K	N	1623	945	*	2297
	316	100K	Y	1584	924	*	2263

TABLE IX
DIRECT: PUSHING DUPLICATE ELIMINATION FOR TRANSITIVE CLOSURE (TIMES IN SECONDS).

G	n	$k=2$		$k=4$		$k=6$	
		N	Y	N	Y	N	Y
tree	10K	164	426	172	466	186	537
cyclic	1000	21	50	24	55	139	186
complete	100	5	8	21	8	165	8

would generate larger intermediate tables. The explanation is that the non-recursive join is evaluated once at the beginning of the recursion with small tables; the price is that each row in intermediate tables becomes bigger. On the other hand, if this join is evaluated at the end of the recursion it works with "leaner", but much larger, tables.

Pushing Duplicate Elimination

The next set of experiments studies the impact of pushing duplicate row elimination. That is, doing at each iteration instead of at the end of recursion.

Table VIII summarizes results for the Seminaïve algorithm computing the transitive closure. We consider cycle elimination as an extra feature of duplicate elimination. Eliminating cycles turns out to always work well: evaluation is faster or at most equal, but never slower. Therefore, it is a good idea to eliminate cycles when they are detected. In complete graphs this optimization is not only good, but essential. Without duplicate elimination evaluation becomes slow and with large graphs it becomes extremely slow.

Table IX analyzes duplicate elimination for Direct. For cyclic graphs we used smaller graphs ($n = 1000$), because eliminating duplicates with larger graphs took several hours. Pushing duplicate elimination is expensive in large acyclic graphs: in trees the time to evaluate queries pushing duplicate elimination is almost triple. The trends are similar to Seminaïve: eliminating duplicates is required for complete graphs.

Pushing aggregations

The following experiments show the impact made by pushing aggregations. Table X analyzes the power matrix. For acyclic graphs (trees and lists) this optimization works well: in all cases it is faster to push the aggregation at each recursive step instead of computing the aggregation at the end. For large cyclic graphs, pushing aggregations increases time, highlighting the cost to group rows. Finally, at $k = 2$ it is bad for complete graphs, but it is clearly required at $k = 4$ where duplicate rows are abundant. The explanation behind the ineffectiveness of this optimization for complete graphs at deep recursion depth is that the number of duplicates grows exponentially. Therefore, the cost to eliminate or summarize duplicates at each recursive step is higher.

TABLE X
SEMINAIVE: PUSHING AGGREGATION FOR POWER MATRIX (TIMES IN SECONDS).

G	n	m	$k=2$		$k=4$	
			N	Y	N	Y
tree	100K	100K	49	22	80	60
	200K	200K	71	59	162	142
	1M	1M	562	483	1425	1154
list	100K	100K	39	23	94	53
	200K	200K	67	50	231	160
	1M	1M	437	428	1042	923
cyclic	10K	20K	9	2	79	67
	20K	40K	19	10	117	145
	100K	200K	139	178	708	1369
complete	100	10K	36	27	*	104
	142	20K	79	138	*	434
	316	100K	1812	3508	*	*

All Optimizations Working Together

We now consider the interaction of optimizations with each other. We consider two sets of complex queries: queries with a high selectivity predicate and queries without selection. The first set of queries return a small table R, whereas the second one produces a large table R. For each type of graph we used the optimal setting for optimizations whose defaults were as follows. Storage and indexing was based on vertex for trees, lists and cyclic graphs, whereas we used edge for complete graphs. Duplicate and cycle elimination were pushed at each iteration for all graphs. Non-recursive joins (i.e. with N) were evaluated early. Selection (when there was a WHERE selection predicate) was evaluated early for all graphs. Recursion depth was $k = 4$, which is challenging as seen in previous experiments. In order to understand the importance of each each optimization we tested optimizations with the largest graphs analyzed in previous experiments.

We first analyze a "complex" query for transitive closure which eliminates duplicates, has a non-recursive join to get vertex names and uses a high selectivity condition in the WHERE clause ($i = n/2$). In this case table N was small: N was precomputed selecting all vertices reachable from one vertex; such small table acted as a filter when join with T. Table XI compares the impact of turning each optimization off, maintaining the rest turned on. Since results for smaller graphs were very similar they are omitted. All optimizations working together achieve the minimum evaluation time. Evaluating selection at the end has the most significant impact for all graphs, followed by the non-recursive join evaluated at the end of recursion. Duplicate elimination at the end of recursion had a significant impact only for complete graphs, confirming our findings from experiments discussed before.

TABLE XI
SEMINAIVE: OPTIMIZATIONS WORKING TOGETHER (TIMES IN SECONDS).

G	n	all on	Each optimization off: storage index	non-rec join	selection	duplic elimin
tree	1M	0.7	0.7	3.7	605.7	0.8
list	1M	0.7	0.7	3.8	570.9	0.8
cyclic	100K	0.7	0.7	1.5	631.8	0.8
complete	316	0.7	0.7	4.6	*	*

TABLE XII
SEMINAIVE: OPTIMIZATIONS WORKING TOGETHER WITHOUT SELECTION (TIMES IN SECONDS).

G	n	all on	Each optimization off: storage index	non-recur join	duplic elimin
tree	1M	547	*	741	916
list	1M	551	*	711	800
cyclic	100K	476	*	671	720
complete	316	4242	>3 hrs	4053	>3 hrs

Finally, storage/indexing had minimal impact, which can be explained by the impact of early selection producing a small table from the start of recursion.

We now analyze the second case, where the "complex" query does not have a selection predicate and table N has all vertices. That is, the query has duplicate elimination and a non-recursive (external) join. All rows from R are returned. Table XII analyzes the impact of turning each optimization off, omitting times over 1 hour for trees, lists and cyclic graphs. All optimizations turned on achieve the lowest time, but compared to the previous experiment, time differences are not as significant (except for storage). Storage/indexing is the most important optimization. Using storage/indexing by edge for trees, lists, cyclic increases time over one hour. Similarly, time goes over one hour when a complete graph is stored and indexed by vertex. Pushing duplicate elimination is essential for complete graphs and has a moderate impact on the other graphs (i.e. not an order of magnitude slower). Non-recursive joins show the smallest gap; early evaluation of non-recursive joins is still a slightly better alternative.

F. Time Complexity

The next experiments study scalability varying n and k with large tables, having two goals: (1) understanding time complexity; (2) quantifying the impact of the type of graph. We used the optimal storage and indexing scheme based on vertex. Queries did not have duplicate elimination.

Figure 2 shows time growth as n and k vary. We use these default values for k: unbounded k for trees, for lists $k = 10$ and for cyclic graphs $k = 5$. Scalability is linear for all types of graphs. Time growth is better than linear for lists; quasi-linear for trees; and linear for cyclic graphs. There is a big gap in performance between cyclic graphs and acyclic graphs, despite the fact that $k = 5$. These results highlight the difficulty of computing transitive closure on cyclic graphs, given the rapidly growing number of paths as k grows. Evidently, time growth is much faster for complete graphs; those times are not plotted. We now discuss time complexity for k. In this case

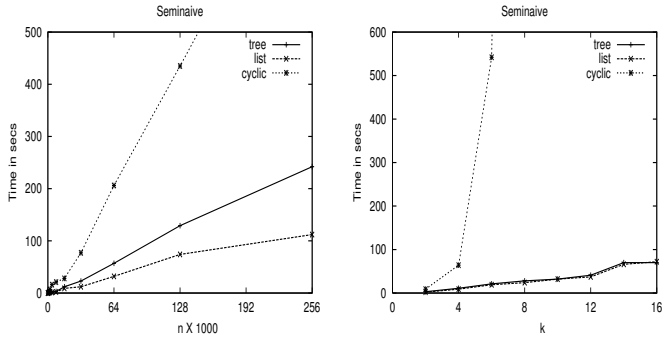

Fig. 2. Seminaive: Query evaluation time varying n and k.

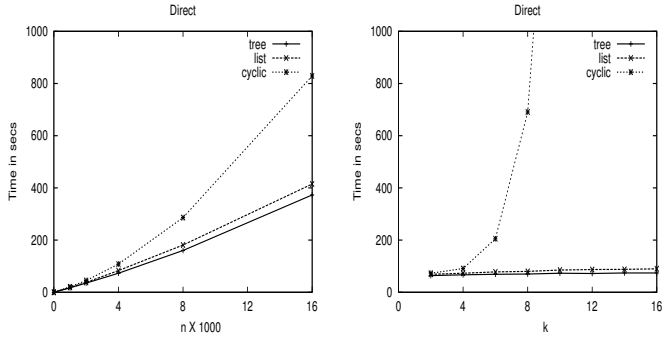

Fig. 3. Direct: Query evaluation time varying n and k.

the default is n =64k for Seminaïve. Time grows slowly and linearly for trees and lists with Seminaïve, but exponentially for cyclic graphs. Then it becomes constant at $k = 15$ which is close to $log(n)$ for n=64k.

Figure 3 shows trends for Direct. Scalability is worse than linear for trees and lists. Time grows even faster for cyclic graphs. Compared to Seminaïve, there is also a big gap between acyclic and cyclic graphs. Time is almost the same for trees and lists, indicating the dominating factor is n rather than the type of graph.

G. General Recommendations

Storage and indexing should be based on vertex, unless the graph is highly connected (e.g. similar to a complete graph). Selection should be evaluated as early as possible on any type of graph, preserving correctness. Non-recursive joins should be performed early if the external table columns do not significantly increase result row size or if the external table is small. Otherwise, a late join at the end of the recursion can be faster. Pushing duplicate elimination will generally speedup processing and it is required to process highly connected graphs (e.g. complete). Similarly, pushing aggregation is generally effective because it compresses intermediate tables during recursion. Finally, queries should always have a small recursion depth threshold k, which can be gradually increased as the user explores the properties of the underlying graph.

VI. RELATED WORK

We start by discussing relational query optimization surveys [4], [7]. Reference [4] explains the main ideas to push selection

predicates in SPJ queries and performing a "group-by" operation before a join operation (early or eager). This optimization is applicable when the number of resulting groups is smaller than the size of one of the tables to be joined and it is related to pushing aggregation. On the other hand, [7] discusses query optimization exploiting physical database operators and SPJ query transformation techniques. The optimization of recursive queries in SQL is not considered in [4], [7].

Research on recursive queries and transitive closure computation is extensive. Most work has been in the context of deductive databases [1], [3], [10], [22], [18], [19], [21], [26], or adapting deductive database techniques to relational databases [5], [14], [16], [15], [23]. There exists a somewhat orthogonal line of research that has adapted graph-based algorithms to solve the transitive closure problem in a database system (not necessarily relational) [2], [10]. Finally, there is significant theoretical work on recursive query computation in the Datalog language [12], [20], [24]. There exist several algorithms to evaluate recursive queries including Seminaïve [3], Logarithmic [23], Direct [2], and BTC [10]. It has been shown that Seminaïve solves the most general class of recursive queries based on so-called fixpoint equations [2], [3]. Both Seminaïve [3] and Logarithmic [23] are based on iterative joining until no more rows are added to the result. Logarithmic [23], also called Smart, is an optimization of Seminaïve in which paths whose length is a power of 2 are added to the transitive closure on a first pass, and then remaining edges are added on a second pass; our optimizations can be applied to Logarithmic. Direct algorithms [2] are a class of methods that terminate regardless of the underlying graph structure; they are not recursive and they process each vertex a constant number of times (e.g. once). Direct algorithms are in theory more robust with graphs of different characteristics. However, we have shown the Direct algorithm performance is indeed impacted by graph structure or recursion depth, when implemented in SQL. Another drawback of direct algorithms is that they are less general than Seminaïve [2]. That is, their computation power is inferior. In [2] the authors present direct algorithms that evaluate a transitive closure recursive query in less time than Seminaïve. They improve I/O time by rearranging tuples into blocks. Our implementation of Direct followed this work. There are several differences though. We programmed Direct with SQL queries instead of exploiting any special data structures (lists) or bit operations, which would require internally modifying the database system. They do not consider the power matrix problem, which is more difficult. Last, we studied query optimization with larger graphs.

We now discuss optimization techniques for recursive queries. Pushing selection predicates is the most well researched optimization in traditional SPJ query optimization [4], [7] and deductive databases [3], [19]. Optimization of row selection, mostly based on equality, has been extensively studied with the magic sets transformation [14], [16], [15], [21]. The magic set transformation was proposed with equality comparison (passing variable bindings) and was later generalized to inequality comparisons [16], [15]. The magic sets transformation is similar to the early selection optimization and pushing aggregation optimizations, introduced in our work. Both are query rewriting techniques and both attempt to reduce the size of intermediate results. However, there are important differences. The standard magic set query transformation creates additional tables (relations), introduces extra joins and queries are transformed (rewritten) with additional clause terms. Filtering happens in a different way: in magic sets filtering is performed when joins are evaluated and then only relevant tuples are kept at each evaluation step. On the other hand, early selection attempts to keep only relevant tuples by filtering rows as early as possible like traditional SPJ queries. Magic sets were later adapted to work on relational database systems [16], even on non-recursive queries, but the authors caution other specialized techniques for linear recursive queries (like Direct algorithms) can provide better performance than magic sets. Magic sets require join reordering using cost-based optimization [16]. Deductive and relational databases have different semantics [16], [19]: SQL requires tuples to always have values (non-ground facts not allowed) and it allows duplicates, nested queries, aggregations, existential and universal quantifiers. Therefore, special care must be taken when applying deductive database optimizations such as magic sets [14], [16], [15]. Pushing agregation through recursion is quite different from the magic set transformation; in magic sets the filtering predicate passes through recursion, but the group-by operation is evaluated in the same order, but on fewer rows. On the other hand, we have shown a group-by operation can be evaluated before a join, as explained in [4]. Both optimizations are similar in the sense that they try to reduce the size of intermediate results. and we show non-recursive (external) joins can be evaluated at the beginning or a the end of the recursion. Early selection is similar to transforming queries with magic predicates in the sense that both produce smaller intermediate results. Sideways information passing (sip) refers to passing variable bindings from rule head to rule body [21]; it is related to early selection and pushing aggregation. Reference [21] explains that the magic set transformation may be less efficient than Seminaïve when unnecessary joins are computed. Indexing for evaluation of recursive queries based on the Seminaïve and Logarithmic algorithms, is studied in [23]; the enhanced indexing scheme for Seminaïve is similar to that proposed in [23], but we have shown that in some cases the alternative indexing scheme based on edges is almost as good. Also, we have shown the same storage and indexing scheme works well for the Direct algorithm. Compared to previous work, we focused on expressing recursive queries in relational algebra and evaluating them with SQL queries, without using any memory or disk-based data structures.

This article is an expanded version of [17], studying two query evaluation algorithms instead of one, analyzing query optimization in more depth and evaluating optimizations and scalability on much larger graphs.

VII. Conclusions

This work studied the optimization of linear recursive queries in SQL. In order to study query optimization under an abstract framework we used directed graphs. A graph is represented by a relational table with one row per edge. We

focused on two complementary and deeply related problems: computing the transitive closure of a graph and getting the power matrix of its adjacency matrix. We explained how to program two classical algorithms in SQL: Seminaïve and Direct. These SQL implementations did not use any specific data structures, proprietary language extensions or internal database system features. Both implementations can detect cycles which significantly impact time and may produce infinite recursion. Five query optimizations were proposed and studied: enhanced storage and indexing, early selection. early evaluation of non-recursive joins, pushing duplicate elimination and pushing aggregations. Experiments on a relational database system compared both algorithms with large input tables, studying the impact of each optimization and time complexity. We studied optimizations on four types of graphs: binary trees, lists, cyclic and complete graphs, covering a wide spectrum of time complexity. Seminaïve was significantly faster than Direct, except for complete graphs. The impact of optimizations was as follows. In general, optimized storage and early selection have a strong impact on any type of graph. For acyclic graphs storage and indexing by vertex based on the join condition was the best scheme. For cyclic graphs storage and indexing by edge worked well at low recursion depth and was best for complete graphs. Early selection produced a significant acceleration for Seminaïve for equality predicates, but it cannot be applied to Direct. Early evaluation of non-recursive joins worked well in general, producing a modest performance improvement. Pushing duplicate elimination was generally better than eliminating duplicates at the end of recursion. In complete graphs pushing duplicate elimination proved to be required to make the problem tractable. Cycle elimination worked well in every case, but producing a marginal time improvement. Pushing aggregation through recursion worked well in every case. When all optimizations work together on a complex query with selection, non-recursive joins and duplicate elimination storage/indexing and early selection have the most significant impact. From a time complexity perspective, Seminaïve scales linearly with respect to table size, whereas Direct is superlinear. Seminaïve showed linear scalability for acyclic graphs for increasing recursion depth. Both algorithms show exponential time growth with increasing recursion depth for cyclic and complete graphs, highlighting a combinatorial explosion of paths.

Important issues for future research include the following. We plan to study recursive query optimization based on I/O cost models. Early row selection needs to be studied for range queries or when the filter condition cannot be evaluated in the base step. Pushing duplicate elimination needs to be studied in more depth since it an expensive computation and it can be avoided in acyclic graphs. Non-recursive join evaluation needs to be studied in more generality, with multiway joins, varying external table size, larger records and more columns. Efficient mechanisms are needed to detect queries that may produce an infinite recursion.

REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases : The Logical Level*. Pearson Education POD, facsimile edition, 1994.

[2] R. Agrawal, S. Dar, and H.V Jagadish. Direct and transitive closure algorithms: Design and performance evaluation. *ACM TODS*, 15(3):427–458, 1990.

[3] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. ACM SIGMOD Conference*, pages 16–52, 1986.

[4] S. Chaudhuri. An overview of query optimization in relational systems. In *Proc. ACM PODS Conference*, pages 84–93, 1998.

[5] S. Dar and R. Agrawal. Extending SQL with generalized transitive closure. *IEEE Trans. Knowl. Eng.*, 5(5):799–812, 1993.

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

[7] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[8] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab and sub-total. In *ICDE Conference*, pages 152–159, 1996.

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

[10] Y.E. Ioannidis, R. Ramakrishnan, and L. Winger. Transitive closure algorithms based on graph traversal. *ACM TODS*, 18(3):512–576, 1993.

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

[12] L. Libkin and L. Wong. Incremental recomputation of recursive queries with nested sets and aggregate functions. In *DBPL*, pages 222–238, 1997.

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

[14] I.S. Mumick, S.J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *ACM SIGMOD*, pages 247–258, 1990.

[15] I.S. Mumick, S.J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic conditions. *ACM TODS*, 21(1):107–155, 1996.

[16] I.S. Mumick and H. Pirahesh. Implementation of magic-sets in a relational database system. In *ACM SIGMOD*, pages 103–114, 1994.

[17] C. Ordonez. Optimizing recursive queries in SQL. In *ACM SIGMOD Conference*, pages 834–839, 2005.

[18] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. Implementation of the coral deductive database system. In *ACM SIGMOD*, pages 167–176, 1993.

[19] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. The CORAL deductive system. *VLDB J.*, 3(2):161–2120, 1994.

[20] S. Seshadri and J.F. Naughton. On the expected size of recursive datalog queries. In *ACM PODS Conference*, pages 268–279, 1991.

[21] S. Sippu and E.S. Soininen. An analysis of magic sets and related optimization strategies for logic queries. *J. ACM*, 43(6):1046–1088, 1996.

[22] J.D. Ullman. Implementation of logical query languages for databases. *ACM Trans. Database Syst.*, 10(3):289–321, 1985.

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

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

[25] H.S. Warren. A modification of Warhsall's algorithm for the transitive closure of binary relations. *CACM*, 18(4):218–220, 1975.

[26] C. Youn, H. Kim, L.J. Henschen, and J. Han. Classification and compilation of linear recursive queries in deductive databases. *IEEE TKDE*, 4(1):52–67, 1992.