Comparing Columnar, Row and Array DBMSs to Process Recursive Queries on Graphs

Carlos Ordonez, Wellington Cabrera, Achyuth Gurram
Department of Computer Science
University of Houston
Houston, TX 77204, USA *

Abstract

Analyzing graphs is a fundamental problem in big data analytics, for which DBMS technology does not seem competitive. On the other hand, SQL recursive queries are a fundamental mechanism to analyze graphs in a DBMS, whose processing and optimization is significantly harder than traditional SPJ queries. Columnar DBMSs are a new faster class of database system, with significantly different storage and query processing mechanisms compared to row DBMSs, still the dominating technology. With that motivation in mind, we study the optimization of recursive queries on a columnar DBMS focusing on two fundamental and complementary graph problems: transitive closure and adjacency matrix multiplication. From a query processing perspective we consider the three fundamental relational operators: selection, projection and join (SPJ), where projection subsumes SQL group-by aggregation. We present comprehensive experiments comparing recursive query processing on columnar, row and array DBMSs to analyze large graphs with different shape and density. We study the relative impact of query optimizations and we compare raw speed of DBMSs to evaluate recursive queries on graphs. Results confirm classical query optimizations keep working well in a columnar DBMS, but their relative impact is different. Most importantly, a columnar DBMS with tuned query optimization is uniformly faster than row and array systems to analyze large graphs, regardless of their shape, density and connectivity. On the other hand, there is no clear winner between the row and array DBMSs.

1 Introduction

Recursion is fundamental in computer science: many algorithms are naturally recursive, such as graph algorithms. Recursion has been incorporated into SQL via recursive queries [15, 12, 19]. Unfortunately, recursion is not available in all DBMSs and its implementation varies widely despite an ANSI SQL standard. In fact, most row DBMSs offer recursive queries (e.g. Postgres, Oracle, Teradata, IBM DB2, MS SQL Server), but they are not currently available in most columnar DBMSs (e.g. MonetDB, Vertica, C-Store, with the exception of SAP Hana [4]). This lack of querying capability is no coincidence as recursive queries represent one of the most challenging class of queries. A current trend in analytic database systems and data warehousing are so-called column stores [27] (also called column-oriented databases or columnar database systems), which have been shown to provide an order of magnitude performance improvement in evaluating analytical queries on large tables, mixing joins and aggregations. Since we are concerned about systems used in practice we focus on fully functional column-based database systems (e.g. supporting SQL, basic ACID properties, parallel evaluation, basic fault tolerance), which we simply call "columnar DBMSs" to contrast them with "old" row-oriented DBMSs. Within big data analytics graph problems are particularly difficult given the size of data sets, the complex structure of the graph (density, shape) and the mathematical

nature of computations (i.e. graph algorithms). With that motivation in mind, we study the optimization of recursive queries on a columnar DBMS to analyze large graphs.

1.1 Motivation

We focus on the evaluation of queries with linear recursion, which solve a broad class of difficult problems including reachability, shortest paths, network flows and hierarchical aggregation. Some motivating examples include the following. Assume there is a human resources database, with a table containing employee/manager information (a self relationship in ER modeling terms). That is, there are two columns corresponding to the employee and the manager, respectively. Typical queries include: "list employees managed directly or indirectly by manager X", "how many people are managed by Y?", "list managers with at least 10 managed employees", "sum the salaries of all employees managed by Z". Assume you have a database with flight information from multiple airlines. In this case the input table has a departing city and an arriving city, with associated cost and distance. Representative queries include: "give me all cities I can arrive to departing from this airport with no more than 2 connections", "find the cheapest flight between every pair of cities", "count the number of cities reachable with no more than 3 connections", "for every pair of cities count how many potential flights there are". As a more modern example related to the Internet, consider a social network like Facebook or Twitter, where typical queries include the following. "How many people are indirectly related to other persons with up to two common acquaintances?", "is there anyone in group A who knows someone in group B, and if so, how many connections are there between both groups?", "if one person spreads some news/gossip, how many people can be reached?". We should mention we do not tackle queries mixing negation and recursion which represent a harder class of queries. In summary, recursive queries open up the possibility to exploit a columnar DBMS to solve many fundamental graph problems.

Efficient processing of recursive queries is a fundamental problem in the theory of databases, where Datalog is the most prominent declarative language [1]. In contrast, research on recursive queries in SQL is rather scarce and existing research has only focused on row storage, the dominating storage for the past three decades. This is due to variations in recursive query implementation (despite an ANSI standard), the difficulty in understanding how recursion and query optimizations are combined and the common perception that a DBMS is hard to tune. However, graph problems are becoming more prevalent and more graph-structured data sets are now stored on SQL engines. To the best of our knowledge, optimization of recursive queries has not been revisited with columnar DBMSs. Having columnar DBMSs as the main motivation to perform graph analytics, these are some representative research issues: can relational DBMSs tackle large graphs or should they get out of the way and let other no-SQL systems do the job?, are columnar DBMSs indeed faster?, is it necessary to adapt classical query optimization techniques to columnar DBMSs?, are there considerations to change or improve existing storage or indexing techniques?, are there new considerations to accelerate recursive joins, the most demanding relational operator?, can aggregation help reducing the size of intermediate results and perhaps query evaluation time?, does the graph structure and connectivity impact query processing time, as it happens in Hadoop/noSQL systems?, is recursion depth a big hurdle in dense graphs, as past research has shown? We attempt to provide clear answers to these questions.

1.2 Contributions

This is an overview of our research contributions. We start by reviewing a unified Seminaïve algorithm that works on both column and row DBMSs, based on automatically generated SQL queries. As a major contribution of our paper, we establish a connection between two graph problems and two recursive queries: transitive closure evaluated with a recursive join and adjacency matrix multiplication evaluated with a recursive query combining join and aggregation. We revisit query optimization of SPJ queries showing even though recursive query optimization is a well studied problem, there are indeed new research issues on

columnar DBMSs. In order to study scalability and query optimizations with predictable results, we introduce a flexible graph generator that allows simulating graphs representing the Internet and social networks. Finally, we present a benchmark comparing a columnar DBMS, a row DBMS and an array DBMS, covering a wide spectrum of database technologies available today.

1.3 Article Outline

Section 2, a reference section, introduces graph and relational database definitions and gives an overview of storage mechanisms. Section 3 presents our main technical contributions: the standard Semi-naïve algorithm to evaluate recursive queries, SQL queries for each algorithmic step, query optimizations for relational operators, and their algebraic query transformations, highlighting differences in a columnar DBMS. We also include a time complexity analysis per relational operator. Section 4 compares query processing in a columnar DBMS with two prominent DBMSs: row-based and array. Experiments also evaluate the impact of each query optimization on graphs with different structure, density and connectivity. Section 5 discusses closely related work, focusing on SQL query optimization. Section 6 summarizes theoretical contributions, experimental findings, and directions for future research.

2 Definitions

This is a reference section which introduces standard graph definitions from a discrete mathematics perspective, the relational database model and basic SQL queries implementing the Semi-naïve algorithm. Each subsection can be skipped by a reader familiar with the material.

2.1 Graphs from a general mathematical perspective

To provide a mathematical framework and objective analytic goals we use graphs as input. Let $G = (V, E)$ be a directed graph with $n = |V|$ vertices and $m = |E|$ edges. We emphasize that m represents size to store G as an edge list (space complexity). We would like to clarify that we prefer to use m instead of m (or alternative letters) to emphasize space complexity (i.e. storage). An edge (i, j) in E links two vertices in V and has a direction. Undirected graphs, a particular case, are easily represented by including two edges, one for each direction. Notice our definition allows the existence of cycles and cliques in graphs, which make graph algorithms slower. A cycle is path starting and ending on the same vertex. A clique is a complete subgraph of G. The adjacency matrix of G is a square $n \times n$ matrix denoted by E. We use the term node to refer to a vertex graph that contains cliques.

2.2 Graphs in a Relational Database

We now introduce additional definitions in a database context, extending the graph definitions introduced above. Assuming G is a sparse graph, G is stored on table E as a list of edges and the result of the recursive query is stored on table R with similar schema. Let E be defined as a table with schema $E(i, j, v)$. Assuming there are no duplicate edges E has primary key (i, j) and v represents a numeric value (e.g. distance). Otherwise, it is necessary to either introduce an additional column to distinguish multiple edges or aggregate duplicate edges before populating E (e.g. storing only the edge with min(v)). Table E is the input for a recursive query using columns i and j to join E with itself, multiple times, as explained below. Let R be the result table returned by a recursive query, with schema $R(d, i, j, p, v)$ and primary key (d, i, j), where d represents recursion depth, i and j identify an edge at some recursion depth, p counts the number of paths and v represents some numeric value (typically recursively computed, e.g. minimum distance). A row from table E represents either a weighted edge in G between vertices i and j or a binary matrix entry, representing the existence of an edge. Table E has m rows (edges), where $1 \leq m \leq n^2$, $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, n\}$. To guarantee recursion termination and reasonable computation time, there is a query

recursion depth threshold k, provided by the user. In summary, from a mathematical point of view E is a sparse matrix and from a database angle E is a long and narrow table having one edge per row.

2.3 Fundamental Graph Problems

We study the optimization of recursive queries based on two complementary and deeply related graph problems: (1) transitive closure of G, which involves computing a new graph $G+$: $G^+ = (V, E')$ s.t. $(i, j) \in E'$ if there exists a path between i and j. (2) computing the power matrix E^k, via an iterative matrix multiplication of E: $E \cdot E \ldots \cdot E$. That is, multiplying E by itself $k-1$ times. Problem (1) is classical in graph algorithmic theory, whereas Problem (2) establishes a theoretical connection with linear algebra and discrete mathematics. Evidently, both problems have a strong connection. However, their solution based on relational queries is different.

Transitive Closure: The transitive closure of G computes all vertices reachable from each vertex in G, building a new graph G^+ and it is defined as: $G^+ = (V, E')$, where $E' = \{(i, j) \mid \exists$ a path between i and $j\}$. That is, G^+ is a new graph with the same vertices, but with additional edges representing connectivity between two vertices. As extreme cases, if $m = 0$ (empty graph, $E = \emptyset$) or $m = n^2$ (complete graph) then $G = G^+$, resulting in the fastest and slowest computation, respectively. The challenge is that, in practice, G is a sparse graph and m is somewhere in the middle.

Iterative Matrix Multiplication: Recall from Section 2 that E is a real or binary matrix. When E is binary the power matrix E^k (E multiplied by itself k times) provides the number of paths of length k between each pair of vertices, and it is defined as: $E^k = \Pi_{i=1}^{k} E$. Notice that in linear algebra terms we compute $E \cdot E$, and not $E \cdot E^T$. The iteration produces the sequence of matrices $E, E^2, \ldots, E^k$, where E^J counts the number of paths of length J per vertex pair. When $k = 2$ this computation is equivalent to a standard matrix multiplication and when $k > 2$ this computation is the power matrix E^k. It is important to emphasize that recursive queries subsume matrix multiplication as a particular case because many graph algorithms are based on matrix multiplication (e.g. neighborhood density, counting triangles, Jaccard coefficient).

2.4 Recursive Queries in SQL

We study queries of the form: $R_d = R_d \bowtie E$, where the most common join predicate is equality in equi-join $R_d.j = E.i$ (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]. As noted above, transitive closure is deeply related to matrix multiplication.

In this work, we focus on the evaluation of the SQL recursive view introduced below, based on input table E and output table R. The standard mechanisms to define recursive queries in ANSI SQL is a recursive view using "RECURSIVE VIEW". We do not discuss syntax for an equivalent SQL construct for derived tables (WITH RECURSIVE, or the CONNECT BY clause used in Oracle [17]). 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 operator 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 we show the base step first.

We define queries for the two problems introduced in Section 2. We start by defining the following recursive view R, which expresses the basic recursion to join E with itself multiple times. We emphasize R appears once in the FROM clause obeying a linear recursion.

CREATE RECURSIVE VIEW

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

Based on R, the transitive closure (TC) G^+ is computed as follows. As a first solution, we can query R quickly by not eliminating duplicates (this will be our default query form for TC):

CREATE VIEW Gplus AS (SELECT i, j FROM R);

To produce a more succinct output, we can eliminate duplicate edges from G^+ using the optional DISTINCT keyword. Notice this query is slower because it requires sorting rows.

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

In SQL the power matrix (P) view, based on R, returns $E, E^2, \ldots, E^k$ and the second statement only returns E^k:

CREATE VIEW P AS (
  SELECT d, i, j,sum(p) AS p,min(v) AS v
  FROM R
  GROUP BY d, i, j);
SELECT * FROM P WHERE $d = k$; /* E^k */

The SQL view above counts the total number of paths (sum(p)) at each recursion depth (d) and computes the minimum length among all paths (i.e., shortest path), with respect to v. In other words, P provides an interesting summarization of G^+.

In general, the user can write queries or define views using R like another input table. The most important constraint from a theoretical perspective is that recursion must be linear. This means R can only appear once. In other words, R cannot appear twice or more times in the "FROM" clause.

The SQL ANSI standard, introduces several constraints in recursive views and queries to enforce standard syntax and semantics. However, optimization is the responsibility of each query optimizer. The recursive view definition cannot contain "group by", "distinct", "having", "not in", "outer join", "order by". However, such SQL keywords can indeed appear outside in any query calling the view. On the other hand, multiple recursive views cannot be nested with each other to avoid indirect infinite recursion by mutual reference.

2.5 DBMS Storage

In this article we study query optimization and experimentally compare column, row and array DBMSs, which represent three major alternatives to analyze big data, with row DBMSs being legacy systems and column and array DBMSs being new competing technologies. It is impossible to explain storage and query processing on each kind of DBMS in technical detail. Instead we highlight and compare their salient features to process recursive queries.

Column, row and array DBMSs have fundamentally different storage mechanisms, which lead to different query processing strategies. Column Database Systems rely on compression, a consequence of keeping column values ordered. In columnar database systems like C-Store/Vertica base (original) tables are substituted with projections, where in each projection there is one file per column [27]; tuple materialization at the end is required. On the other hand, in systems like MonetDB [9], there is one file per column for the base

table, but no projections are needed; tuple materialization is avoided. Column-based storage is significantly different from the row blocks and B-trees used in a row DBMS [29]. Physical storage to decrease query processing time varies significantly. In a columnar DBMS maintaining column values sorted is essential [7, 13, 27], whereas in a row DBMS indexes (B-trees, hash tables or bitmaps) are the most common mechanism, complemented by ordering block rows. In an array DBMS storage is neither row, nor column-based, but as multidimensional subarrays (called chunks in SciDB [28], indexed by a grid data structure with chunk boundaries in main memory).

3 Recursive Query Processing

We start by reviewing SQL queries for the standard algorithm to evaluate recursive queries with SQL: Seminaïve. Such SQL queries do not depend on any specific storage mechanism or database system architecture. After understanding these basic aspects, we revisit optimization of recursive queries. Our research contribution lies in contrasting how recursive queries are optimized in a columnar DBMS compared to row and array DBMSs. We study the optimization of SPJ queries involving selection, projection and join operators where projection includes duplicate elimination and group-by aggregations as two particular cases. For each operator we first present its optimization from an algebraic perspective and then we discuss how the operator is evaluated considering each different DBMS architecture.

3.1 Seminaïve Algorithm

In order to make the paper self-contained we review Seminaïve, using as input the graph G defined in Section 2. The standard and most widely used algorithm to evaluate a recursive query comes from deductive databases and it is called Seminaïve [2, 3]. The Seminaïve algorithm solves a general class of mathematical logic problems called fixpoint equations [2, 1]. Let R_k represent a partial output table obtained from $k-1$ self-joins with E as operand k times, up to a given maximum recursion depth k:

$$R_k = E \bowtie E \bowtie \ldots \bowtie E,$$

where slightly abusing notation each join uses $E.j = E.i$ (i.e. in SQL each table has an alias $E_1, E_2, \ldots, E_k$). The base step produces $R_1 = E$ and the recursive steps produce $R_2 = E \bowtie E = R_1 \bowtie_{R_1.j=E.i} E$, $R_3 = E \bowtie E \bowtie E = R_2 \bowtie_{R_2.j=E.i} E$, and so on. Notice that the general form of the recursive join is $R_{d+1} = R_d \bowtie_{R_d.j=E.i} E$, where the join condition $R_d.j = E.i$ links a source vertex with a destination vertex if there are two edges connected by an intermediate vertex. Notice that at each recursive step a projection (π) is required to make the k partial tables union-compatible. Assuming graphs as input, π computes $d = d+1$, $i = R_d.i$, $j = E.j$, $p = R.p * E.p$ and $v = R_d.v + E.v$ at each iteration:

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

In general, to simplify notation from Equation 1 we do not show neither π nor the join condition between R and E: $R_{d+1} = R_d \bowtie E$. The final result table is the union of all partial results: $R = R_1 \cup R_2 \cup \ldots \cup R_k$. If R_d eventually becomes empty at some iteration, because no rows satisfy the join condition, then query evaluation stops. In other words, R reaches a fixpoint [1, 30]). The query evaluation plan is a deep tree with $k-1$ levels, k leaves with table E and $k-1$ internal nodes with a $\bowtie$ between R_d and E. Therefore, the query plan is a loop of $k-1$ joins assuming recursion bounded by k.

The following SQL code implements Seminaïve [19] and it works on any DBMS supporting SQL. Notice graph cycles are filtered out to avoid double counting paths and reducing redundancy. It is a good idea to set a threshold k on recursion depth instead of reaching a fixpoint computation [3], in order to bound evaluation time on large or dense graphs. Since a real database may contain multiple edges per vertex pair it may be necessary to pre-process the graph. In a similar manner, we may insert into temporary tables multiple edges (i.e. bag semantics), which can be later eliminated to compute the final set union in R.

/* pre-process E: delete duplicate edges per vertex pair
from some input table T with multiple edges per vertex pair */
SELECT $i, j, min(v), max(1)$
INTO E
FROM T
GROUP BY i, j ;

/* base step */
INSERT INTO R_1
SELECT $1, i, j, v, 1$
FROM E;
/* recursive step expansion */
FOR $d = 1 \ldots k-1$ DO
INSERT INTO R_{d+1}
SELECT $d+1$, $R_d.i$, $E.j$,$R_d.p * E.p$, $R_d.v + E.v$
FROM R_d JOIN E ON $R_d.j = E.i$
WHERE ($R_d.i \neq R_d.j$) /* eliminate loops */ ;
END

/* $R = R_1 \cup R_2 \ldots \cup R_k$ */
FOR $d = 2 \ldots k$ DO
INSERT INTO R
SELECT i, j, p, v FROM R_d;
END

3.2 Optimizing Recursive Join: Storage, Indexing and Algorithm

We first study how to efficiently evaluate the most demanding operator in recursive queries: the join operator. We focus on computing G^+, the transitive closure of G, without duplicate elimination since it involves an expensive sort. As explained above, G^+ requires an iteration of $k-1$ joins between R_d and E, where each join operation may be expensive to compute depending on m and G structure. Notice that computing E^k requires a GROUP BY aggregation, which has important performance implications and which has a close connection to duplicate elimination. Duplicate elimination is studied as a separate problem and time complexity is analyzed at the end of this section, after all optimizations have been discussed.

The first consideration is finding an optimal order to evaluate the $k-1$ joins. From the Seminaïve algorithm recall $R_d \bowtie E$ with join comparison $R_d.j = E.i$ needs to be evaluated $k-1$ times. But since evaluation is done by the Semi-naïve algorithm there exists a unique join ordering. $R_1 = E$ and $R_d = E \bowtie E \bowtie \ldots \bowtie E = ((E \bowtie E) \bowtie E) \bowtie \ldots E) \bowtie E$ for $d = 2 \ldots k$. Clearly, the order of evaluation is from left to right. In this work, we do not explore other (associative) orders of join evaluation such as $((E \bowtie E) \bowtie (E \bowtie E)) \bowtie ((E \bowtie E) \bowtie (E \bowtie E)) \ldots$ (logarithmic) because they require substantially different algorithms. Computing the final result $R = R_1 \cup R_2 \ldots \cup R_k$ does not present any optimization challenge when duplicates are not eliminated (Naïve algorithm). Therefore, we will focus on evaluating $R_d \bowtie E$ and then discuss its generalization to $k-1$ joins.

In a columnar DBMS [7, 9, 13, 27] each column is stored on a separate file, where column values are sorted by a specific, carefully selected, subset of columns. Since repeated values end up being contiguous it is natural to use compression. In this case the natural compression algorithm is Run-Length Encoding (RLE), where instead of storing each value the system stores each unique value and its frequency. When there are many repeated values such compressed storage dramatically reduces I/O cost and it can help answering aggregations exploiting the value frequency. It is noteworthy there are no indexes from the DBA perspective:

the columnar DBMS maintains internal sparse indexes to the first and last value of each compressed block. The join optimization involves sorting E ordering by i, j and creating a sorted temporary table R_d ordering by j, i (i.e., inverting the two ordering columns) which enables a hash join or a merge join, with a merge join being preferable because for the columnar DBMS merge joins work in time $O(m)$, skipping the sort phase of a sort-merge join. Otherwise, hash joins are a good alternative, with average time $O(m)$, but they are sensitive to skewed key distributions.

In a row DBMS the fastest algorithms are hash joins ($O(m)$ average), followed by merge-sort joins ($O(m \log(m))$ worst case). If both tables are sorted by the joining key the row DBMS can choose the same merge join algorithm, explained above, bypassing the sorting phase resulting also in time $O(m)$. On the other hand, if one table is sorted, but the other is not the row DBMS generally chooses a sort-merge join taking time $O(m \log(m))$. We should point out that because the join condition is $R_d.j = E.i$, in general there are multiple connecting edges per vertex, resulting in many duplicates. When using hash joins such high number of duplicate values produces many collisions which must be handled. In a row DBMS there are two major choices to accelerate joins: physically sorting rows in R_d or E (or both) or creating an index on i or j to speedup joins. In general, creating an index is more expensive than sorting in multiple iterations. Since index creation on a large temporary table is expensive and there are $k-1$ temporary tables we sort E edges by i and R_d by j, as the default join optimization. This tuned optimization is equivalent to the sorted projection used in a columnar DBMS.

In the array DBMS [28] there are two major features to improve join evaluation: (1) chunk size, which involves setting sizes of a 2-dimensional subarray (i.e. similar to a block of records). (2) sparse or dense storage, which requires knowledge on the fraction and distribution of zeroes across chunks. For a 2-dimensional array setting chunk size further requires deciding if the chunk shape should be squared or rectangular. Since E is squared we believe it is more natural to choose a squared chunk. Deciding sparse or dense storage is easy since it simply involves deleting zeroes. However, manipulation in main memory must be carefully considered: a dense chunk is transferred almost directly into a dense array in main memory, whereas a sparse chunk requires either converting to a dense representation or using a special subscript mechanism for sparse arrays. Needless to say, a dense array in RAM for a dense E is faster. Chunks are automatically indexed based on the chunk boundaries with a grid-like data structure, which may saturate RAM if chunk size is too small (i.e. a fine grid). The array DBMS physically sorts cells within each chunk in major column order (i.e. 1st chunk dimension, 2nd chunk dimension and so on). Therefore, changing cell order on secondary storage is not possible. We emphasize that in general it is necessary to tune chunk size depending on the graph density. But there is a catch-22 situation: tuning chunk size cannot be done without having some knowledge about G and knowing G structure requires loading G into the array DBMS with a chunk size already set. Therefore, this is a fundamental optimization difference with the column and the array DBMS, where the block size plays a less significant role.

3.3 Optimizing Projection: Pushing Duplicate Elimination and Aggregations

We consider π as a general operator that projects chosen columns, eliminates duplicates and compute GROUP BY aggregations. We start by discussing duplicate elimination, from which we generalize to group-by aggregations.

Optimizing Duplicate Elimination

This optimization corresponds to the classical transitive closure problem (reachability): edge existence in G^+ instead of getting multiple paths of different lengths (in terms of number of edges, distance or weight). In fact, getting all paths is a harder problem since each path represents a graph and therefore, storage requirements can grow exponentially for dense graphs.

Recall $\pi_{i,j}(R_d) = \pi_{i,j}(E \bowtie E \bowtie \ldots \bowtie E)$. Then the unoptimized query is $\pi_{i,j}(R) = \pi_{i,j}(R_1 \cup R_2 \cup \ldots \cup R_k)$. On the other hand, the equivalent optimized query is

$$\pi_{i,j}(R) = \pi_{i,j}(\pi_{i,j}(R_1) \cup \pi_{i,j}(R_2) \cup \ldots \cup \pi_{i,j}(R_k)).$$

Notice a $\pi_{i,j}(R)$ is required after the union. In general, pushing $\pi_{i,j}(R)$ alone requires more work at the end than pushing $\pi_{i,j,sum}(R)$ due to the additional pass, but it does not involve computing any aggregation.

When this optimization is turned off duplicates are eliminated only at the end of the recursion.

SELECT DISTINCT i, j FROM R;

On the other hand, when this optimization is turned on duplicates are incrementally eliminated at each recursion depth d. Notice an additional pass on R is still needed at the end.

FOR $d = 2 \ldots k$ DO
 INSERT INTO R
 SELECT DISTINCT d, i, j FROM R_d;
END

SELECT DISTINCT i, j FROM R;

Notice a GROUP BY i, j a with a sum() aggregation, does not make sense: the sum $E+E^2+..+E^k$ does not make sense since it would overlap path length information, but it makes sense for a min() aggregation. Therefore, this is an important difference between both classes of queries.

In the array DBMS duplicates must be eliminated at each iteration due to the array storage model. Otherwise, it would be necessary to add a new array dimension with the recursion depth, resulting in a 3D array. Therefore, this optimization cannot be turned off in the array DBMS.

Optimizing GROUP BY Aggregation

Assume we want to compute E^k as defined in Section 2. Therefore, we need to compute a GROUP BY aggregation on R, grouping rows by edge with the grouping key $\{i, j\}$. Computing aggregations by vertex, not involving edges, is more complicated as explained in [19], but fortunately, they represent less common queries in practice. A byproduct of a GROUP BY aggregation is that duplicates get eliminated in each intermediate table. Therefore, a SELECT DISTINCT query represents a simpler case of a GROUP BY query. Therefore, it is natural to extend the relational operator π with aggregation functions. For instance, an aggregation query grouping rows by recursion depth d and edge is $\pi_{d,i,j,\text{sum}(v)}(R)$.

Since storage, indexing and sorting are already considered in join optimization for column and row DBMSs the same principles apply to aggregations. Therefore, for this optimization we do not consider optimizations based on sorting values or indexing rows.

The fundamental question is to know if it is convenient to wait until the end of recursion to evaluate the GROUP BY or it is better to evaluate the GROUP BY *during* recursion. Considering that a GROUP BY aggregation is a generalization of the π operator extended with aggregation functions (i.e. $\pi_{i,j,\text{sum}(p)(R)}$) this optimization is equivalent to pushing π through the query tree, like a traditional SPJ query. On the other hand, pushing π resembles pushing σ, but there is a fundamental difference: we cannot do it once, we need to do it multiple times. In relational algebra terms, the unoptimized query is:

$$S = \pi_{d,i,j,sum(v)}(R)$$

and the optimized query is below. In contrast to duplicate elimination, a final $\pi_{d,i,j,sum(v)}(S)$ is unnecessary to get E^k:

$$S = \pi_{1,i,j,sum(v)}(R_1) \cup \pi_{2,i,j,sum(v)}(R_2) \ldots \cup \pi_{k,i,j,sum(v)}(R_k).$$

Assuming G is large and having a complex structure we cannot assume all the distinct grouping key values of $\{i, j\}$ fit in RAM. Therefore, in the most general case we must assume each GROUP BY evaluation requires having the input table to be sorted by edge. On the other hand, each sort may eliminate many duplicate $\{i, j\}$ keys resulting in decreased I/O in future iterations. We make the hypothesis that this optimization should work well for dense graphs where there are multiple paths per vertex pair.

3.4 Optimizing Row Selection: Pushing Filtering

Pushing the σ operator is the most well-proven optimization in relational query processing, based on the equivalence of different queries based on relational algebra. In general, a highly selective predicate used in a σ expression should be pushed all the way up through recursion when possible. The main difference compared with traditional SPJ queries is that σ must be pushed carefully. We consider a filter predicate on the vertex id (i.e. i, j). Pushing filters on lookup tables with vertex information is similar to pushing filters on i, j. Pushing filters on the edges requires considering monotonic aggregations on v, which is an aspect reserved for future work. In our following discussion consider a vertex selection predicate $i = 1$ for $i \in V$. The unoptimized query is

$$S = \sigma_{i=1}(R).$$

On the other hand, by pushing σ we obtain the optimized query at the bottom:

$$\begin{aligned} S &= \sigma_{i=1}(R) \\ &= \sigma_{i=1}(R_{k-2}) \bowtie E \\ &= \ldots \\ &= \sigma_{i=1}(R_2) \bowtie E \bowtie \ldots \bowtie E \\ &= \sigma_{i=1}(R_1) \bowtie E \bowtie \ldots \bowtie E \\ &= \sigma_{i=1}(E) \bowtie E \bowtie \ldots \bowtie E \end{aligned}$$

We emphasize it would be incorrect to push σ into every occurrence of E because the transitive closure would not be computed correctly. To be more specific,

$$\sigma_{i=1}(E) \bowtie \sigma_{i=1}(E) \neq \sigma_{i=1}(E) \bowtie E = \sigma_{i=1}(E \bowtie E).$$

In SQL terms pushing σ means evaluating the WHERE clause as early as possible, provided the result is the same. The unoptimized query is

SELECT * FROM R WHERE i=1;

On the other hand, the sequence of optimized SQL queries is:

SELECT * FROM R_1 WHERE $i = 1$;
SELECT * FROM R_2 WHERE $i = 1$; /* redundant */
..
SELECT * FROM R_k WHERE $i = 1$; /* redundant */

This sequence of queries can be reduced to the following query since applying the filter on the remaining queries is redundant.

SELECT * FROM R_1 WHERE $i = 1$;

As explained in [19] it would be incorrect to push a filter on the qualified columns used in the join predicate. For instance, the following query is incorrect ($E_2.j$=1 is also incorrect):

SELECT * FROM E E_1 JOIN E E_2 ON $E_1.j = E_2.i$
WHERE $E_2.i = 1$;

3.5 Time Complexity

We analyze time complexity per Semi-naïve iteration considering different graphs of different structure and connectivity. We also discuss how algorithms and time complexity change depending on DBMS storage. To provide a common theoretical framework we first discuss time complexity from a general perspective based on each relational algebra operator: $\bowtie$, π and σ. Our order of presentation is motivated by the importance of each operator in recursive query evaluation.

- Join $\bowtie$:
  To simplify analysis we assume a worst case where $|R_d| = O(m)$, which holds at low k values and it is a reasonable assumption on graphs with skewed vertex degrees (e.g. having cliques). Then time complexity for the join operator can vary from $O(m)$ to $O(m^2)$ per iteration, as explained below. Since R_d is a temporary table we assume it is not indexed. On the other hand, since E is an input table and it is continuously used, we assume it is either sorted by the join column or indexed. During evaluation, R_d is sorted in some specific order depending on the join algorithm. At a high level, these are the most important join algorithms, from slowest to fastest: (1) nested loop join, whose worst time complexity is $O(m^2)$, but which can generally be reduced to $O(m \cdot \log(m))$ if R_d or E are sorted by the join column. (2) sort-merge join, whose worst case time complexity is $O(m \cdot \log(m))$, assuming either table R_d or E is sorted. (3) hash join, whose worst case time complexity can be $O(m^2)$ with skewed data, but which on average is $O(m)$ assuming selective keys and uniform key value distribution, which heavily depends on G structure and density. That is, it is not useful in dense graphs because many edges are hashed to the same bucket. (4) finally, merge join is the most efficient algorithm, which basically skips the sorting phase and requires only scanning both tables. This is a remarkably fast algorithm, with time complexity $O(m)$, but which assumes both tables are sorted.

  In the columnar DBMS the algorithm of choice is hash join regardless of G structure, followed by merge join when either table needs to be sorted. Since column-based storage requires sorting column values, a nested loop join algorithm does not make sense. In the row DBMS the best algorithm is the sort-merge join, which works well on sparse or dense G. If G is very sparse (e.g. a tree), a rare case, a hash join can be faster. Finally, in the array DBMS hash joins are preferred because chunks are hashed and distributed over processing nodes based on the array dimensions.

- Push π:
  As explained in Section 3.3, from a theoretical perspective eliminating duplicates and computing group-by aggregations represent a generalized π operator. In query optimization terms, this means that pushing π (DISTINCT or GROUP BY) requires a sort at each iteration. Assuming large m, large n and $m \gg n$ time complexity is $O(m \cdot \log(m))$ per iteration. The key question is to know whether $k-1$ sorts may be more expensive than only one sort at the end. On one hand if k is deep and the graph is dense the cost of $O(k)$ sorts can be substantial: $O(km \cdot \log(m))$. On the other hand, if the sorting phase is done only at the end the size of R can be potentially very large if there are multiple paths (duplicate edges) between vertices. In the worst case $|R| = km$. Therefore, time complexity for one sort in the worst case can be $O(km \cdot \log(km))$. For a dense graph, where $m = O(n^2)$ (dense graph) the computation cost may be prohibitive: can be $O(kn^2 \cdot \log(kn^2))$. But for a hyper-sparse graph, where $m = O(n)$, time can be $O(kn \cdot \log(kn))$.

  Since a projection requires visiting every row the physical operators is a (table/array) scan across DBMSs, regardless of storage. The main difference is that a columnar DBMS needs to assemble and disassemble rows, whereas the row DBMS can do it directly. In the array DBMS performance will depend on projecting arrays on fewer dimensions, which requires changing chunk storage.

- Push σ:
  There are two main aspects: (1) the time to evaluate the filter predicate; (2) the reduction in size of

a table to be joined. In general, the time to evaluate any predicate on an unsorted or unindexed table E, regardless of predicate selectivity is $O(m)$. The impact of this optimization on the recursive query highly depends on the predicate selectivity. A highly selective predicate can significantly reduce time. Assume $|R_d| = O(1)$. Then time to compute $R_d \bowtie E$ can be $O(n)$ and the k iterations then $O(kn)$. On the other hand, if $|R_d| = O(n)$ time complexity does not decrease.

The time to evaluate the filter predicate in the DBMS depends on the predicate itself and being able to exploit some ordering property. If there is no ordering the input array or table must be scanned in time $O(m)$. A vertex equality predicate (e.g. i=10), the common type of filter, can be evaluated in time $O(\log(m))$ for a sorted table (projection in the columnar DBMS) or B-tree indexed table E (row DBMS). An edge equality predicate (e.g. $i = 10 \wedge j = 100$) can range from $O(\log(n))$ with ordered input down to $O(1)$ if there is a hash index (array DBMS).

4 Experimental Evaluation

Since our SQL-based algorithm and optimizations produce correct results (i.e. we do not alter the basic Semi-naive algorithm), our experiments focus on measuring query processing time. In order to evaluate query processing under challenging conditions we analyze a wide spectrum of graphs having different structure, shape and connectivity. Our experimental evaluation analyzes three major aspects:

1. Evaluating the impact of classical query optimizations.
2. Understanding the impact of G structure on query processing.
3. Comparing column, row and array DBMSs with each other.

We conducted a careful benchmark comparison tuning each DBMS, but results may vary with other DBMSs, especially if they provide hybrid storage (i.e. row+column) or specialized subsystems for graphs. Also, we aim to understand how effective query optimizations are on new generation DBMSs. We report the average time of three runs per recursive query. Table entries marked with "stop" mean query evaluation could not finish in reasonable time and thus queries were stopped; to evaluate query optimizations we stopped at 30 minutes (1800 seconds) and to analyze the most challenging graphs with optimizations turned on we stopped at 2 hours (7200 seconds). When a DBMS crashed for any reason (insufficient RAM, temporary file/array overflowing temporary storage, bugs) we report "fail". All measured times are given in seconds.

4.1 Experimental Setup

Here we provide an overview of how we conducted experiments in order to replicate them.

DBMS Software and Hardware

We compared the three database systems under demanding conditions to force continuous I/O between a single disk and small main memory. To make sure the input table was read from disk, the buffers of each DBMS were cleared before processing each recursive query. We conducted experiments on two identical servers, each with an Intel Quad Core 2.13 GHz CPU, 4 GB RAM and one 3TB disk, each running the Linux Ubuntu operating system. Following DBMS user's guide recommendations, each DBMS was tuned to exploit parallel processing with multi-threading in the multicore CPU. A benchmark on a parallel cluster is out of scope of this paper since DBMSs vary widely on hardware supported, exploiting distributed RAM and parallel capabilities. However, trends should be similar and gaps in performance wider.

We used a columnar DBMS and a row DBMS supporting ANSI SQL. The array DBMS was SciDB [28], which supports AFL, a functional language to define arrays and write queries on arrays and AQL,

Table 1: Type of graph G.

G	cycles	cliques	density	m edges	complexity
tree	N	N	very sparse	$O(n)$	best
cyclic	Y	N	sparse	$O(n)$	fair
clique-tree	Y	Y	medium	$O(MK^2)$	medium
clique-cyclic	Y	Y	medium	$O(MK^2)$	bad
clique-complete	Y	Y	dense	$O(M^2K^2)$	very bad
complete	Y	Y	very dense	$O(n^2)$	worst

an SQL-like language based on AFL calls. Our choice of SciDB was motivated by being parallel, matrix-compatible, fully functional and providing the AFL language, capable of expressing SPJ queries, including group-by aggregation. In order to preserve anonymity of the other DBMSs, we do not mention the DBMS name or whether the DBMS is open source or industrial. However, since our benchmark study is based on analyzing query processing without modifying DBMS internal source code, our major research findings should be valuable to users, developers and DBAs trying to decide which system to use.

SQL Code Generator

We developed a generic SQL code generator in the Java language connecting to each DBMS via JDBC (i.e. aiming to generate standard SQL queries). This Java program had parameters to specify the input table (and columns), choose DBMS SQL dialect and turn each optimization on/off. The recursive view was unfolded by creating the iteration of of k SQL statements, following the Seminaïve algorithm from Section 3. Query evaluation was performed using temporary tables for each step populating each table with SELECT statements. Time measurements were obtained with SQL with timestamps for maximum accuracy.

Experimental Parameters

The buffers of each DBMS were cleared before evaluating the recursive query (i.e., clearing the DBMS cache). That is, we made sure table E was initially read from disk. In order to get evaluation times within one hour and produce a uniform set of intermediate results, we did not run recursion to get the full G^+, which would require a practically unbounded recursion depth (i.e. $k = n$). We initially tested queries on several graphs to investigate a maximum recursion depth k, so that evaluation could finish in less than 1 hour. Based on our findings, we consider $k = 2$ a shallow recursion depth (equivalent to matrix multiplication), $k = 4$ medium and $k = 6$ deep. Only for trees it was feasible to go beyond $k = 6$. We shall convince the reader these seemingly "low" recursion depth levels stress the capabilities of each DBMS.

Graph Data Sets

We analyzed synthetic and real graph data sets. Synthetic data sets vary in size and structure, whereas real data sets are fixed.

Synthetic Graphs: We evaluated recursive queries with synthetic graphs, but we were careful to generate realistic graphs with complex structure and varying degree of connectivity, summarized in Table 1. Our experimental evaluation used two major classes of graphs: simple graphs where cliques are not part of data generation (tree, cyclic, complete) and graphs where cliques are initially generated and then connected (having prefix "clique-"). Within each class there are three graph types based on their density (connectivity): trees (binary, balanced), cyclic (long cycles) and complete (no edges missing), going from easiest to hardest. Notice a complete graph represents a worst, unrealistic, case, full of cliques from size 3 (triangles) to n (i.e.

Table 2: Optimization: recursive join to compute transitive closure G^+ (recursion depth $k = 6$; clique size $K = 4$; times in seconds).

			columnar projection		row order		array storage	
G	n	m	N	Y	N	Y	dense	sparse
tree	10M	10M	112	101	454	437	stop	stop
cyclic	1M	1M	11	12	48	47	stop	1314
clique-tree	312k	1M	1124	1055	stop	stop	fail	771
clique-cyclic	312k	1M	1082	1004	stop	stop	fail	405
clique-complete	1300	100k	stop	stop	stop	stop	41	41
complete	100	10k	stop	stop	stop	stop	25	25

G itself) to test recursive query processing. In order to understand how recursive queries behave with different graphs we applied a 2-phase data generation approach. In Phase 1 we decide if the graph will have cliques (also called "fat" nodes), which is a major factor impacting query processing time. Then during Phase 2 vertices (or cliques) are connected. Graphs and their parameters are summarized in Table 1. If G has cliques each "fat node" is a clique, whose size we control. We decided to call this parameter K, after the well-known Kuratowski graph K_n, (an important observation is that K, clique size, is denoted in uppercase, not be confused with recursion depth k, in lower case). If G has no cliques each node is "lean", representing a simpler time complexity case. In graphs with "lean" nodes we connect vertices directly with an edge, according to the graph structure, with the number of edges going from n to n^2. For graphs with fat nodes (i.e. prefixed with "clique-") we assume there are initially M "fat nodes", then connected by $M - 1$ edges (clique-tree), M edges (clique-cycle) and $M(M - 1)$ edges (clique-complete). In this case, we connect some vertex in clique i with some vertex in clique j, in a random manner, guaranteeing cliques are connected with each other. Our graph definitions are comprehensive and subsume disconnected graphs, where each disconnected component can be any of the graphs above. In short, our synthetic graph generator has the following input parameters: n nodes, m edges, M fat nodes and clique size K. Since m is the actual storage size in SQL we generated graphs with m growing in log-10 scale. For graphs with "lean" nodes m determines n, whereas for graphs with "fat" nodes M and K determine n and m. To simplify study we maintain K fixed (e.g. $K = 4$, which represents a family or close mutual friends in a social network, emphasizing solving with K=4 is much harder than with triangles). Needless to say as $K \rightarrow n$, G starts resembling a complete graph, making the problem of computing recursive queries intractable.

Real Graphs: For the real data set we picked two well-known data sets from the Stanford SNAP repository: (1) wiki-vote with n=8k and m=103K. (2) The web-Google with $n = 916k$ and $m = 5.1M$. Both data sets have a significant number of cliques (including many triangles) and medium diameter, resulting in long paths. Since real data sets are particularly challenging because we cannot totally understand their structure, we analyze them with the best query optimizations turned on in each DBMS.

4.2 Evaluating Query Optimizations

We proceed to test the effectiveness of each optimization on each relational operator: $\bowtie$, π, σ (in importance order). In the following experiments the computation is stopped at 30 minutes (1800 seconds).

Optimizing Recursive Join

We start by analyzing the most demanding operator: the recursive join. As explained in Section 3, it is necessary to evaluate an iteration of $k - 1$ joins. Since storage is different in each DBMS the physical join

operator is different and therefore the specific optimization is different as well: projections for the columnar DBMS, sorting rows (edges) in the row DBMS and choosing between dense/sparse storage in the array DBMS. Table 2 compares query processing time turning each optimization on and off.

We start by discussing the columnar DBMS, which did not require major tuning. Projections help the columnar DBMS when the graph is very sparse, especially with large trees. For denser graphs, including graphs with cycles, the time gain becomes smaller. Assuming that in general the structure of G is not known, projections (sorted tables by the join key) in the columnar DBMS are a good optimization. Therefore, projections are turned on by default in our remaining experiments.

We now discuss tuning the row DBMS. We experimentally tried two optimizations: (1) indexes on the join vertices $R_d.j$ and $E.i$ and (2) physically sorting rows in R_d and E by the join vertices, as explained in Section 3.2, to evaluate the iteration of $k-1$ joins. We found out that physically sorting rows in E with an ORDER BY clause in the CREATE TABLE (i.e. clustered storage for edges) was faster than creating a separate index on E (based on source vertex). Sorting R_d, after being created, was expensive when the DBMS used a hash join. Maintaining an index on R_d was expensive as well. Notice that under pessimistic conditions, for every recursive query evaluation we included the initial time to sort or index E in our total times. In practice, however, E is sorted or indexed once, but queried multiple times. Therefore, we identified ORDER BY $E.i$ as the default row optimization to accelerate joins. From Table 2 we can see sorting rows is moderately effective for sparse graphs, but it does not help anyway with denser graphs (we had to stop the query). Based on these results, we decided to initially sort E to accelerate join processing. Therefore, this optimization is turned on by default.

For the array DBMS the optimization choice are sparse and dense storage for arrays, which are the respective choice for sparse and dense graphs, respectively. As discussed in Section 3.2 it is necessary to tune chunk size depending on the graph density. Based on chunk tuning experiments we use two default chunk sizes for the remaining experiments: (1) 1000×1000 for dense graphs; (2) $100,000 \times 100,000$ for sparse graphs, which produced chunks of average size of 8 MBs, as recommended by the DBMS User's Guide. As can be seen from Table 2, sparse storage is preferable since times are always smaller and because array storage becomes dense when G is complete. Overall the array DBMS is the fastest with dense graphs (cliques, complete), but it is slower by two orders of magnitude than the columnar DBMS with sparse graphs (trees). The pattern is the same compared to the row DBMS, but with a smaller gap (i.e. row DBMS faster one order of magnitude). The main reason the array DBMS is so slow using dense storage for trees is that it evaluates joins on arrays with almost empty chunks, full of zeroes (i.e. doing unnecessary work). On the other hand, joins are still significantly slow using sparse storage for trees (i.e. zeroes are deleted) because the graph is too sparse and chunks remain sparsely populated (helped a bit by RLE compression). When embedding cliques into the graph array size explodes as depth k grows: only sparse storage works well. These results highlight that the array DBMS is inefficient to evaluate joins on sparse graphs or semi-dense graphs (with cliques) that produce a dense transitive closure graph as they are explored. In conclusion, in further experiments we store G in sparse matrix form by default, eliminating all zeroes.

Optimizing Projection: Pushing Duplicate Elimination and Group-by

The previous experiments do not give the column and row DBMS the opportunity to eliminate duplicates. Table 3 helps understanding the impact of duplicate elimination when computing transitive closure G^+. Recall from Section 3.3 that in the array DBMS duplicates must be eliminated due to the array storage model. All graphs, except trees, produce duplicates in intermediate results during recursion. Therefore, it is necessary to know whether to eliminate duplicates during recursion or it is better to wait until the end of recursion. Duplicate elimination is unnecessary for trees. Therefore, it is expected time are worse on every DBMS. However, the negative impact is not equally significant: it is significantly worse on the columnar DBMS. Our explanation is that rows must be assembled from separate files for each column and then sorted at each iteration in order to detect duplicates. For the row DBMS, the impact is small, whereas for the

Table 3: Optimizing projection: Pushing duplicate elimination (recursion depth $k = 6$; clique size $K = 4$; times in seconds).

			columnar optimization		row optimization		array
G	n	m	Y	N	Y	N	default=Y
tree	10M	10M	148	112	728	577	523
cyclic	1M	1M	16	11	67	57	109
clique-tree	312k	1M	49	1103	297	stop	226
clique-cyclic	312k	1M	44	963	229	stop	223
clique-complete	1300	100k	310	stop	stop	stop	616
Complete	100	10k	2	stop	20	stop	16

Table 4: Optimizing projection: Pushing GROUP BY aggregation to compute E^k (recursion depth $k = 6$; clique size $K = 4$; times in seconds).

			columnar optimization		row optimization		array
G	n	m	Y	N	Y	N	default=Y
tree	10M	10M	288	114	964	689	stop
cyclic	1M	1M	29	11	90	70	1314
clique-tree	312k	1M	60	1450	503	stop	771
clique-cyclic	312k	1M	42	1419	434	stop	405
clique-complete	1300	100k	601	stop	stop	stop	666
Complete	100	10k	3	stop	29	stop	25

array DBMS duplicates are automatically eliminated at each iteration. On the other hand, for dense graphs (with cliques, complete) this optimization becomes a requirement to make the problem tractable: without it times are more than an order of magnitude bigger. With this optimization column and row DBMSs become much more competitive with the array DBMS. In fact, the columnar DBMS becomes uniformly faster. Therefore, the effectiveness of this optimization depends on G structure and recursion depth k. In the absence of information about G structure and because G most likely contains cliques it is better to apply this optimization by default.

Computing the GROUP BY aggregation for E^k should produce a similar trend to computing transitive closure since the query plan is the same. The main difference is computing the aggregated value v, which requires an extra column. Table 4 compares pushing GROUP BY aggregation through recursion, as explained in Section 3.3. Recall pushing GROUP BY acts as a compression operator since it reduces the size of intermediate results. As we can be seen from Table 4 this optimization works very well for column and row DBMSs for dense graphs: without it they crawl. Overall, with this optimization the columnar DBMS becomes the fastest and the row and array DBMS exhibit similar performance with each other. Only with the largest complete graph, an unrealistic worst case, the row DBMS is the worst. In summary, the trends are the same as duplicate elimination. In big data analytics, G is likely to contain cycles and cliques. Therefore, this optimization should be turned on by default.

Table 5: Optimizing row selection: Pushing row filtering on power matrix E^k (recursion depth $k = 6$; clique size $K = 4$; times in seconds).

			columnar optimization		row optimization		array optimization	
G	n	m	Y	N	Y	N	Y	N
tree	10M	10M	18	100	27	361	13	stop
cyclic	1M	1M	2	9	3	41	9	1314
clique-tree	312k	1M	2	990	4	stop	12	771
clique-cyclic	312k	1M	2	976	3	stop	12	405
clique-complete	1300	100k	1	stop	2	stop	7	666
complete	100	10k	1	stop	1	stop	6	25

Optimizing Row Selection: Pushing Selection Filters

Evaluating the effectiveness of pushing σ requires deciding some comparison predicate. By far, the most common is equality. In the case of G the most common predicate is equality on a vertex attribute (e.g. id, name, description). Since G structure varies significantly in our synthetic graphs and in order to have repeatable results, we decided not to choose some random vertex. Instead, we chose vertex i=1, making clear the DBMS has no specific knowledge about such vertex. In this manner, our experiments are repeatable and explainable. That is, optimizing row filtering is done with a WHERE predicate $i = 1$, as shown on Table 5. Confirming decades of research, this optimization works well across all DBMSs, regardless of storage mechanism. However, the relative impact is different: in the columnar DBMS and the array DBMS the speed gain is two orders of magnitude with dense graphs, whereas in the row DBMS three or more orders of magnitude with dense graphs (the recursive query cannot finish in less than 30 minutes). Therefore, this optimization confirms that a highly selective predicate should be pushed all the way up through recursion when possible. In summary, with this optimization turned on all DBMSs come much closer to each other (assuming the user knows which vertex to explore), but the columnar DBMS still has the leading edge.

4.3 Comparing Column, Row and Array DBMSs

In this section we compare the three DBMSs with the best optimization settings based on previous experiments, analyzing challenging graphs at a recursion level as deep as possible. We analyze synthetic and real graphs. We emphasize real graphs are "harder" than trees, but "easier" than complete graphs. These experiments aim to understand strengths and weaknesses of each system when facing with the task on analyzing a large graph whose structure is not well understood. In this case we stop the computation at 2 hours (7200 seconds), giving each DBMS full opportunity to evaluate the recursive query.

We made a point G structure (shape and connectivity) plays a big role on query processing time. Experiments in Section 4.2 uncovered two important facts: (1) there is big time gain when tuning the query plan or graph storage to get faster joins; (2) pushing projection significantly reduces the size of intermediate tables and the additional time to do it in acyclic graphs is small (but not negligible). Therefore, we evaluate recursive query processing with: (1) the fastest join algorithm provided by each DBMS; (2) pushing projection (duplicate elimination, aggregation) at each iteration. In order to make the comparison in a more challenging manner and having a more realistic (informative) query we analyze the computation of the power matrix E^k, which requires computing a GROUP BY aggregation. Since selecting vertices assumes knowledge about the graph and makes query evaluation much easier (i.e. all DBMSs have similar performance) pushing selection is not applied (e.g. WHERE $i = 1$).

Table 6 provides a comparison under a "tuned" configuration, but still without assuming anything about

Table 6: Comparing DBMSs with best optimization to get power matrix E^k (fastest join, push group by, duplicates eliminated, no row filtering; stop at 2 hours; times in secs).

					DBMS		
G	cliques	n	m	k	columnar	row	array
tree	N	10M	10M	8	57	1158	3391
clique-cyclic	Y	1M	1M	6	258	322	405
clique-complete	Y	1300	100k	6	601	stop	666
complete	Y	100	10k	6	3	29	25
wiki vote	Y	8k	100k	4	85	4500	426
wiki vote	Y	8k	100k	6	187	stop	1461
web-Google	Y	916k	5M	3	1068	stop	stop
web-Google	Y	916k	5M	4	4232	stop	stop

G. Results are interesting: The columnar DBMS is the fastest overall, being faster than both the row DBMS and the array DBMS. For the second place there is no winner: the array DBMS is faster for dense graphs, but loses with sparse graphs. Neither the row DBMS nor the array DBMS can finish analyzing the Google graph. In summary, the columnar DBMS is the fastest, but certainly struggles with the Google graph.

5 Related Work

We start with an overview of past work on recursive query optimization coming from deductive databases. Then based on the most important approaches in deductive databases, we discuss related work on optimizing recursive queries in SQL. We conclude by explaining new contributions in our DOLAP paper comparing it with closely related work on a row DBMS [19].

Research on recursive queries and transitive closure computation is extensive, especially in the context of deductive databases [1, 3, 10, 30, 23, 26, 25, 33], or adapting deductive database techniques to relational databases [6, 16, 15, 31]. 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 [14, 24, 32]. There exist several algorithms to evaluate recursive queries including Seminaïve [3], Logarithmic [31], Direct [2], and BTC [10]. Past work has shown that Seminaïve solves the most general class of recursive queries, based on fixpoint equations [2, 3]. Both Seminaïve [3] and Logarithmic [31] work by iteratively joining the input table until no more rows are added to the result table (i.e., reaching a fixpoint computation). More recently, [26] proposed a hybrid approach to query graphs with a language similar to SPARQL, maintaining data structures with graph topology in main memory and storing graph edges as relational tables like we do. This work considers reachability queries (i.e. transitive closure), but not their optimization on modern architecture DBMSs.

The optimization of recursive queries in SQL has also received attention, but as a body of research it is comparatively smaller than Datalog. To review query optimization of SPJ queries, from a classical perspective, read survey papers [5, 8]. Most DBMSs offer recursive queries via Common Table Expressions (CTEs) [4, 19], which are temporary tables created and referenced during evaluation. Another well-known for of recursive queries is the CONNECT BY clause, introduced by the Oracle DBMS [17]. This clause works in the same manner as the RECURSIVE VIEW with a join condition linking two vertices via an intermediate vertex (the ”connecting” vertex). Pushing selection predicates is the most well researched optimization in traditional SPJ query optimization [5, 8] and deductive databases [3]. Optimization of row selection, mostly based on equality, has been extensively studied with the magic sets transformation [16, 15, 25], but not

so much in SQL. The magic set transformation was proposed with equality comparison (passing variable bindings) and was later generalized to inequality comparisons [16, 15]. Magic sets transformation resemble early selection optimization and pushing aggregation. Both techniques are based on query rewriting and both reduce the sizes of intermediate tables. Nevertheless, there exist important differences. Magic sets create additional tables (relations), introduce extra joins and queries are rewritten with additional clause terms. Filtering works in a different manner: filtering takes place when joins are computed, keeping only relevant tuples at each iteration. 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) are more efficient than magic sets. Datalog and SQL have different semantics [16]: SQL requires tuples to have values or values be marked as missing (non-ground facts not allowed) and it allows duplicates, nested queries, existential and universal quantifiers. Therefore, special care must be taken when applying deductive database optimizations such as magic sets [16, 15]. Pushing aggregation 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 explained how a group-by aggregation is evaluated before a join, around similar lines to [5]. Indexing tables for fast evaluation of recursive queries based on the Seminaïve and Logarithmic algorithms is studied in [31]; our ordered (clustered) storage schemes is similar to that proposed in [31]. Since computing a recursive aggregation on a graph is really an iteration of matrix multiplication there is potential to optimize this computation as a linear algebra problem; a closely related work on this angle is [11], which studies query optimization of matrix multiplication on a column store, but with the entire matrix stored on RAM. An outer matrix product for data set summarization [20, 22] is similar to the matrix multiplication of the graph adjacency matrix with itself, explored in this article; this topic is an important research issue.

Recent works on optimizing SQL recursive queries are [18, 19], which reopened the study of recursive queries in SQL. Motivated by the graph analytics trend and the new wave of column stores, [21] revisited the problem with column DBMSs. This is a summary of new research contributions in our paper with respect to [21]. We justified efficiency and correctness of optimized queries via algebraic query transformations (i.e. with faster equivalent queries). We studied two additional query optimizations: duplicate eliminate and pushing row selection. In this manner, we consider the full spectrum of SPJ queries, including GROUP BY aggregation as a special case. We added a time complexity analysis based on graph structure. From an experimental standpoint the major additions were studying the impact of optimization for the three most important relational operators as well as adding the array DBMS as a third competitor. In addition, we conducted benchmark experiments with the most challenging synthetic graphs and real graphs tuning each DBMS with the best query optimization settings.

6 Conclusions

We presented a first study to compare recursive query processing in columnar, row and array DBMSs to analyze large graphs. We introduced a simplified Seminaïve algorithm that works on several DBMSs based on queries (SQL or equivalent query language). We analyzed query processing with the three fundamental relational operators: join, projection, selection, where projection includes duplicate elimination and group-by aggregation as particular cases. We studied how to push relational operators, preserving result correctness, to decrease query processing time. Our time complexity analysis makes clear the graph structure plays a major role. We presented an extensive experimental evaluation analyzing the impact of query optimizations and a benchmark comparing columnar, row and array DBMSs. We introduced a benchmark graph generator, having size, shape and clique size as main parameters. Our results show that in general pushing relational operators produces significant time improvement, confirming query processing research, but whose impact heavily depends on DBMS storage, graph density and recursion depth, aspects that had not been studied before. We showed graph structure plays a major role, especially when the graph is very sparse or when the graph has cliques. We provided evidence it is necessary to tune each DBMS for faster join processing and it

is a requirement to push projection (eliminating duplicates and pushing GROUP BY aggregation) to make graph analysis tractable. Based on a final "expert" comparison with synthetic and real graphs, we showed that the columnar DBMS with tuned query optimization is orders of magnitude faster than its competitors on sparse graphs and much faster with dense graphs.

Since our work represents a first study of recursive query processing in a columnar DBMS and a preliminary exploration of array DBMSs to analyze graphs, there are many directions for future research. Within columnar storage we need to study alternative algorithms to evaluate joins and apply other compression mechanisms beyond run-length encoding. Join reordering to improve upon Seminaïve is also important. Analyzing join time complexity at deep recursion levels on dense graphs or graphs with skewed vertex degrees is another important problem. Pushing aggregations deserves further study on graphs with skewed connectivity, varying clique size and deeper recursion. We also need to develop cost models considering main memory and secondary storage that can give more accurate estimations of temporary tables cardinalities and processing time. Within array storage, determining optimal chunk size and chunk shape is a fundamental problem, with impact beyond graph analytics. Given the big data analytics trend, it is also necessary to make a direct performance comparison with so-called graph analytic systems not based on SQL (i.e., no-SQL, plain C++/Java), built on top of the Hadoop distributed file system (HDFS), MapReduce or Spark. Storing paths is a harder problem than computing transitive closure, but in general only a few paths are interesting after an initial exploratory analysis. We need to derive tighter bounds for time complexity considering sparse graphs and deep recursion levels.

Acknowledgments

The first author thanks Michael Stonebraker for his comments to understand query processing in columnar and array DBMSs. This work was partially conducted while the first author was visiting MIT.

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] C. Binnig, N. May, and T. Mindnich. SQLScript: Efficiently analyzing big enterprise data in SAP HANA. In *Proc. of BTW (Datenbanksysteme für Business, Technologie und Web)*, pages 363–382, 2013.

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

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

[7] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database: An architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

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

[9] S. Idreos, F. Groffen, N. Nes, S. Manegold, K.S. Mullender, and M.L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

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

[11] D. Kernert, F. Köhler, and W. Lehner. SLACID - sparse linear algebra in a column-oriented in-memory database system. In *Proc. of SSDBM*, page 11, 2014.

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

[13] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandier, L. Doshi, and C. Bear. The Vertica analytic database: C-store 7 years later. *PVLDB*, 5(12):1790–1801, 2012.

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

[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] Oracle. *SQL Reference*. Oracle Corp., 10g edition, 2003.

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

[19] C. Ordonez. Optimization of linear recursive queries in SQL. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(2):264–277, 2010.

[20] C. Ordonez. Statistical model computation with UDFs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(12):1752–1765, 2010.

[21] C. Ordonez, A. Gurram, and N. Rai. Recursive query evaluation in a column DBMS to analyze large graphs. In *Proc. ACM DOLAP*, pages 71–80, 2014.

[22] C. Ordonez, Y. Zhang, and W. Cabrera. The Gamma matrix to summarize dense and sparse data sets for big data analytics. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2016.

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

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

[25] 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.

[26] S.Sakr, S.Elnikety, and Y.He. Hybrid query execution engine for large attributed graphs. *Inf. Syst.*, 41:45–73, 2014.

[27] M. Stonebraker, D.J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E.J. O'Neil, P.E. O'Neil, A. Rasin, N. Tran, and S.B. Zdonik. C-Store: A column-oriented DBMS. In *Proc. VLDB Conference*, pages 553–564, 2005.

[28] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science and Engineering*, 15(3):54–62, 2013.

[29] M. Stonebraker, L. Rowe, and M. Hirohama. The implementation of Postgres. *IEEE TKDE*, 2(1):125–142, 1990.

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

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

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

[33] 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.