Time Complexity and Parallel Speedup of Relational Queries to Solve Graph Problems

Carlos Ordonez[1] and Predrag T. Tosic[2]

[1] University of Houston, USA
[2] Washington State University, USA

Abstract. Nowadays parallel DBMSs compete with graph Hadoop Big Data systems to analyze large graphs. In this paper, we study the processing and optimization of relational queries to solve fundamental graph problems, giving a theoretical foundation on time complexity and parallel processing. Specifically, we consider reachability, shortest paths from a single vertex, weakly connected components, PageRank, transitive closure and all pairs shortest paths. We explain how graphs can be stored on a relational database and then we show relational queries can be used to efficiently analyze such graphs. We identify two complementary families of algorithms: iteration of matrix-vector multiplication and iteration of matrix-matrix multiplication. We show all problems can be solved with a unified algorithm with an iteration of matrix multiplications. We present intuitive theory results on cardinality estimation and time complexity considering graph size, shape and density. Finally, we characterize parallel computational complexity and speedup per iteration, focusing on joins and aggregations.

1 Introduction

Graph analytics remains one of the most computationally intensive task in big data analytics. This is due to large graph size (going beyond memory limits), graph structure (complex interconnectivity) and the potential existence of an exponential number of patterns (e.g. paths, cycles, cliques). On the other hand, graph problems are becoming more prevalent on every domain, there is a need to query graphs and more graph-structured data sets are now stored on SQL engines. There has also been recent interest in the Hadoop world revisiting recursive queries with SPARQL [19]. In our opinion, even though query optimization is a classical, well studied, topic optimization of relational queries on graphs needs further research. We focus on the evaluation of relational queries which solve a broad class of graph problems including reachability, shortest paths, network flows, PageRank and connected components.

Previous research revisited relational query processing to analyze graphs on parallel DBMSs [4, 12], showing a DBMS is faster than Hadoop "Big Data" systems like GraphX and showing a columnar DBMS is significantly faster than a traditional row DBMS. These papers presented many experimental time performance comparisons and scalability analyses showing the impact of optimizations and efficiency of algorithms. In contrast, in this paper we focus on theoretical issues. To that end, we unify two broad classes of graph algorithms, we study relational query processing and we present important theory results characterizing time complexity and parallel speedup. Given the foundational focus of this paper and space limitations, we do not present experiments, but the reader can find extensive experimental evaluation in [4, 12].

2 Definitions

2.1 Graphs from a Mathematical Perspective

Without loss of generality we consider directed graphs because undirected graphs can be represented including two directed edges per vertex pair. Let $G = (V, E)$ be a directed graph with $n = |V|$ vertices and $m = |E|$ edges. G can contain cycles (simple paths starting and ending at the same vertex) and cliques (hidden complete subgraphs). We will show such graph patterns make graph algorithms slower. From an algorithmic perspective, G is processed as an $n \times n$ adjacency matrix E where E contains binary or real entries (i.e. each edge is present or not, or it has a distance). Notice we overload E to make notation intuitive. Then algorithms use matrix E as input in various computations based on an iteration of matrix multiplication. In general, we assume G is a sparse graph, where $m = O(n)$. Some harder problems, not tackled in this paper, include the traveling salesman problem (TSP) and clique detection. Detecting cliques with k vertices when k is not constant is a harder problem, under the usual assumption of computational complexity ($\mathbf{P} \neq \mathbf{NP}$), because there is an exponential search space for cliques.

2.2 Graph Problems Solved with Matrices and Vectors

We study the solution of two complementary classes of problems: (1) Iterating matrix-vector multiplication, where the solution is an n-vector S. For notational convenience S is a column vector manipulated as a matrix $n \times 1$. (2) Iterating matrix-matrix multiplication, where the result is an $n \times n$ matrix R.

These two classes account for most common problems in graph analytics to analyze social networks and the Internet. Some graph problems worth mentioning that do not fall into these categories are the traveling salesman problem (TSP, which requires visiting every vertex) and clique detection/enumeration, which requires exploring an even larger search space.

Iterating Matrix-Vector Multiplication

The core iteration is $S_1 = E^T \cdot S_0$, $S_2 = E^T \cdot S_1$, and so on, where S_0 has a different initialization depending on the graph algorithm. That is, each new solution is fed back into the same matrix-vector multiplication (somewhat similar to the Gauss-Seidel iterative method to solve linear equations). We would like to point out that for notational convenience it is better to use matrix-vector multiplication intead of vector-matrix multiplication, which would require using many transposition operators in equations. Representative problems include single-source reachability, single-source shortest path, weakly connected components and PageRank, among others. Single-source algorithms initialize the desired vertex i entry in S to 1 ($S[i] = 1$) and all other entries to zero ($S[j] = 0$ when $i \neq j$). Weakly connected components initialize $S[i] = i$ for $i = 1 \ldots n$. PageRank uses a transition matrix T with probabilities obtained from E, initialized as $T_{ij} = E_{ji}/outdegree(j)$, where outdegree($j$) $= E^T \cdot e_j$ (e_j is the canonical vector with 1 at j and zero elsewhere arriving from any vertex to i). Intuitively, in single-source reachabilty and shortest path each iteration gets further vertices reachable from chosen vertex i, starting with vertices reachable with one edge. For weakly connected components each vertex in a component gets labeled with the minimum id of common neighbors. PageRank iterates a special multiplication form $S_{I+1} = T \cdot S_I$ until probabilities of random walks stabilize following a Markov Chain process [7]. Although not central to graph analytics, it is noteworthy Topological Sort can also be expressed with this iteration.

Iterating Matrix-Matrix Multiplication

The basic iteration is to multiply E by itself: $E \cdot E$, $E \cdot E \cdot E$, and so on, which can be expressed as $R_j = R_{j-1} \cdot E$, where $R_0 = E$. Intuitively, we generalize single source reachability to any vertex to a broader reachability from any vertex to any vertex. An important observation is that for directed graphs multiplication is not commutative, because $E \cdot E \neq E \cdot E^T$. Only for undirected graphs $E \cdot E = E \cdot E^T$ because E is symmetric. When E is binary we can treat each vector-vector multiplication as boolean vectors or as real vectors. For each boolean vector dimension pair we compute a logical AND, then we compute a logical OR across all of them. For real vectors the power matrix E^k (E multiplied by itself k times) counts the number of paths of length k between each pair of vertices, and it is defined as: $E^k = \Pi_{i=1}^{k} E$ (i is a local variable here, not to be confused with vertex id i used in queries). Considering E a boolean matrix the transitive closure G^+ edge set is $\mathbf{E}^+ = E \vee E^2 \vee \ldots \vee E^n$. On the other hand, if E is treated as a real matrix, this iterative multiplication algorithm has these operator changes: scalar multiplication becomes scalar addition and the sum() aggregation becomes min() or max() aggregations (resulting in the shortest or longest path).

2.3 Graphs Stored and Processed in a Relational Database

To make exposition more intuitive, we prefer to use the term “table” instead of “relation”, to make the connection with SQL explicit. G is stored in table E as a list of edges, which is an efficient storage mechanism for sparse graphs. Let E be stored as a table with schema $E(i, j, v)$ with primary key (i, j), where v represents a numeric value (e.g. distance, weight). If there is no edge between two vertices or v is zero then the edge is not stored. Table E is the input for relational queries using columns i and j to compute joins, as explained below.

3 Query Processing to Solve Graph Problems

3.1 Iterating a Query Evaluating Matrix-Vector Multiplication

Solution vector S is stored on table $S(i, v)$, where v stores some value $v > 0$. Value v can be a binary flag, the shortest path length, vertex id, or page probability value, depending on the algorithm. From an algorithmic perspective all problems can be solved by a unified matrix-vector multiplication algorithm, under a semiring, exchanging mathematical operations. We use $|S_0|$ to denote the initial number of rows in table S. $|S_0|$ depends on the algorithm, which impacts query processing time. For reachability and single-source shortest path $|S_0| = 1$ (because only the ith entry is present in table S, with 1 and ∞ respectively), whereas for weakly connected components and PageRank $|S_0| = n$ (with vertex ids and probabilities of random walks respectively).

Then the iteration $S \leftarrow E^T \cdot S$ translates into relational queries as $S \leftarrow E \bowtie_{E.i=S.i} S$. For reachability and single-source shortest path $|S_0| = 1$, whereas for weakly connected components and PageRank $|S_0| = n$. Therefore, initialization divides time complexity into two complexity classes. Since the iteration has a linear right recursion the query plan is a right-deep tree with E on the left child, S_{I-1} on the right child and S_I as the parent node. In general, an aggregation is computed at each iteration resulting in this generalized query:

$$S_I \leftarrow \pi_{i:f(E.v \bullet S.v)}(E \bowtie_{E.i=S.i} S_{I-1}),$$

where $f()$ is an aggregation (sum(),min(),max()) and $\bullet$ is a scalar operation (addition $+$, multiplication *). Notice we use π as a general projection operator, capable of computing aggregations. Since $\bullet$ is commutative, associative and has an identity element (0 for $+$, 1 for *), and $f()$ is distributive over $\bullet$ both operators together acting on numbers (real, integer) represent a *semiring*. Such algebraic property allows solving all problems with the same template algorithm, by exchanging mathematical operators.

3.2 Iterating a Query Evaluating Matrix-Matrix Multiplication

We define R as a generalized recursive view which allows solving a broad class of problems. Let R be the result table returned by a linearly recursive query, with schema $R(d, i, j, p, v)$ and primary key (d, i, j), where d represents maximum number of edges in path, i and j identify an edge at a specific length (recursion depth), p and v are computed with aggregations. In this paper, p counts the number of paths and v is an aggregated numeric value (recursively computed, e.g. shortest distance). We assume a recursion depth threshold k is specified to make the problem tractable and to derive $O()$ bounds.

In general, the Seminaive algorithm is presented in the context of First Order Logic (FOL) and Datalog [1]. Here we revisit the Seminaïve algorithm [2, 3], with relational queries, using as input E, defined in Section 2. At a high level, we initialize $R_0 \leftarrow E$ and the basic iteration of Seminaive is $R_I \leftarrow R_{I-1} \bowtie E$, which is a linear recursion on R_I. That is, it stops when $\Delta = R_I \bowtie E = \emptyset$. Then Seminaive produces a sequence of partial tables $R_1, R_2, \ldots, R_k$ and the result R is the incremental union of all partial results: $R = \bigcup_j R_j$. Seminaive stops when R reaches a fixpoint, meaning no more edges were added [1]. This means $R_I = \emptyset$ or R is a complete graph. The join condition and required projection to make tables union-compatible are explained below. We emphasize that we use the most efficient version of Seminaïve algorithm [1] because we assume linear recursion (i.e., we only need Δ to stop). Since the number of iterations and time complexity required by Seminaive heavily depend on G structure we bound recursion with k s.t. $k \ll n$ (assuming G is sparse and n is large).

We now explain relational queries in more technical detail. Because recursion is linear and we have a bound k it can be transformed into an iteration of $k-1$ joins. The initialization step produces $R_1 = E$ and the recursive steps produce $R_2 = R_1 \bowtie_{R_1.j=E.i} E = E \bowtie E$, $R_3 = R_2 \bowtie_{R_2.j=E.i} E = E \bowtie E \bowtie E$, and so on. The general form of a recursive join is $R_{d+1} = R_d \bowtie_{R_d.j=E.i} E$, where the join condition $R_d.j = E.i$ creates a new edge between a source vertex and a destination vertex when they are connected by an intermediate vertex. Notice that at each recursive step a projection (π) is required to make the k partial tables union-compatible. We use R_k to represent the output table obtained by running the full iteration of $k-1$ joins on E, where E appears k times. The final result table is the union of all partial results: $R = R_1 \cup R_2 \cup \ldots \cup R_k$. The query plan is a deep tree with $k-1$ levels, k leaves (E) and $k-1$ internal nodes ($\bowtie$).

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

where $f()$ and $g()$ represent SQL aggregations (e.g. sum, count, max). To make notation from Equation 1 more intuitive we do not show either π or the join condition between R and E: $R_{d+1} = R_d \bowtie E$.

Proposition Transitive closure can be solved with matrix multiplication. Therefore, it can be solved with a unified algorithm. Let $G^+ = (V, E^+)$ be the transitive closure graph and let E have binary entries and $b(E) = F$ be a matrix function that transforms the E matrix to binary form: $F_{ij} = 1$ when $E_{ij} > 1$ and $F_{ij} = 0$ otherwise. Then $E^+ = b(E + E^2 + \ldots + E^k) = E \vee b(E^2) \ldots \vee b(E^k)$. The 2nd expression is evaluated more efficiently with bit operators: $\wedge$ (and) instead of *, $\vee$ (or) instead of + (i.e. like Warshall's algorithm [2]).
Proposition If $|E| = 1$ or $|E| = n(n-1)$ then $G = G^+$.
Proof: If $|E| = 1$ then $E^2 = 0$ so $E^+ = E$. For the second case $E^2 = E$. Therefore, in both cases $E^+ = b(E + E) = E$.

This proposition touches two extreme cases and it is important because it exhibits the optimal cases for Seminaïve, when it stops after just one iteration.

3.3 Time Complexity

Matrix-Vector Multiplication

Time to join E with S for a sparse graph is $O(n \log(n))$, where $|E| = \Theta(n)$. By a similar reasoning, time to compute the aggregation is the same. On the other hand, time complexity can reach $O(n^2 \log(n))$ with a very dense graph (i.e. similar to a complete graph, or having large cliques). Therefore, time complexity for each matrix-vector multiplication is $O(n \log(n))$ for a sparse graph and $O(n^2 \log(n))$ for a very dense graph.

Matrix-Matrix Multiplication

We now analyze time complexity iterating matrix-matrix multiplication considering different graphs of different structure and connectivity.

We focus on analyzing space and time complexity for the most challenging case: the join in iterative matrix-matrix multiplication. By a similar reasoning, $O()$ to compute the sum() aggregation is the same. We start with space complexity. Cardinality estimation is one of the hardest and most well-known problems in query processing [6]. We explore time complexity and query evaluation cost based on G characteristics. The goal is to understand how shape, density and connectivity impact $O()$ and cardinality of each partial result table.

Our first goal is to understand $|R_2| = |E \bowtie E|$. That is, the result of the first iteration of Seminaive. A major assumption is that G is connected. Otherwise, our analysis can be generalized to each connected component (subgraph) and the global time/cost is the sum of individual times/costs. We start with a minimally connected graph, which intuitively captures the best complexity case.
Lemma: Let G be a tree. Then $|E \bowtie E| = \Theta(n)$.
Proof: Since G is a tree $m = n - 1$. The main idea is that the best case is a balanced tree and the worst case is a list (chain). Without loss of generality assume G is a balanced binary tree and n is a power of 2. Then the number of paths of length 2 needs to exclude the leaves: $n/2$. Then the parents of the leaves are paths of length 1 and therefore should be excluded as well. Therefore, $|E \bowtie E| = n/2 - n/4 = \Theta(n)$. For the upper bound, assume G is a list, where each vertex (except the last one) has only one outcoming edge and no cycles. The number of paths of length 2 needs to exclude the last 2 vertices. Therefore, $|E \bowtie E| = n - 2 = \Theta(n)$.
Lemma: Let G be a complete graph without self-loops. Then $|E \bowtie E| = O(n^3)$
Proof: Intuitively, we need to count the number of paths of length 2. There are $n(n-1)$ pairs of vertices $(i, j), i, j \in 1 \ldots n$ Notice that since G is directed $(i, j) \neq (j, i)$. For each pair (i, j) there are $n - 2$ paths of length 2. Therefore, $|E \bowtie E| = n(n-1)(n-2) = O(n^3)$.

The previous result makes clear a complete graph is the worst case for $\bowtie$. Notice that if we make G cyclic by adding one edge so that $m = n$ $O()$ remains the same. Therefore, we propose the following result to characterize $O()$:
Proposition: Let H be a subgraph of G such that H is a complete subgraph (i.e. a clique). Let K be the number of vertices of H. By the previous lemma time is $O(K^2)$. Then the most important issue is K being independent from n, which leads to three cases: (1) If $K = \Theta(1)$ then H does not impact time. (2) if $K = \Theta(f(n))$ and $f(n) = o(n)$ then time is better than $\Theta(n^3)$. Prominent cases are $f(n) = \sqrt{n}$ or $f(n) = \log(n)$. (3) If $K = \Theta(n)$ then time is $\Theta(n^3)$. This result highlights cliques are the most important pattern impacting processing time.

Our ultimate goal is to understand $|R_k|$, where $R_k = E \bowtie E \bowtie \ldots \bowtie E$, multiplying E k times as $k \to n$. Notice that computing E^k is harder than G^+ because Seminaive can stop sooner to find G^+ (e.g. if G is complete). Our first time complexity result shows a list is the worst case for connected G. As explained above, the improved Seminaive algorithm computes the union of partial results at the end of the k iterations. Therefore, it would need $n-1$ iterations. On the other hand, if we computed G^+ after every iteration we can stop at $k = n/2 = \Theta(n)$. In short, if G is a "chain" the number of iterations is $\Theta(n)$. Therefore, we conjecture that the second aspect impacting time of Seminaive is the diameter κ of G (i.e., it stops in time $O(\kappa)$). To conclude this section, we stress that the most important aspect is detecting if the graphs behind $E^2, E^3, \ldots, E^k$ become denser as the k iterations move forward. In fact, $|R_k|$ can grow exponentially as k grows. Therefore, if E^k is dense duplicate elimination is mandatory, which is an optimization studied in the next section.

To simplify analysis, we assume a worst case $|R_d| = O(m)$, which holds at low k values and it is a reasonable assumption on graphs with skewed vertex degree distribution (e.g. containing cliques). Then time complexity for the join operator can range from $O(m)$ to $O(m^2)$ per iteration. 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 collisions, 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.

3.4 Parallel Processing

We assume there are P processors (nodes in a parallel cluster) in a distributed memory (shared-nothing) architecture, where processors communicate with each other via message passing. We consider both classes of graph algorithms introduced above. Recall $n = |V|$.

Parallel Matrix-Vector Multiplication

For matrix-vector multiplication the main issue is to partition E and S so that joins can be locally computed. There are two solutions: (1) Since $|S| = \Theta(1)$ or $|S| = \Theta(n)$ then S can be replicated across all nodes at low cost and in some cases it may be updated in RAM. (2) S can be partitioned by the vertex column to compute the join with E: $E.j$ and S is simply partitioned by $S.i$. We cannot guarantee an even distribution for E, but for S we actually can. Therefore, speedup will depend only on skewed degree vertices being in a subset of the nodes. Assuming $n \gg P$, having s high degree vertices and $s > P$ it is reasonable to assume such s vertices can be evenly distributed across the P nodes. For either solution, once the join result is ready the sum aggregation required by matrix-vector multiplication is computed locally with a local sort (perhaps with hashing in a best case), but without a parallel sort, resulting in optimal speedup going from $O(n/P \log(n/P))$ (sparse) to $O(n^2/P \log(n/P))$ (dense).

Parallel Matrix-Matrix Multiplication

For matrix-matrix multiplication the main issue is how to partition E to make multiplication faster, considering that the join operation accesses E in a different order from the previous iteration. If E is dense with $m = O(n^2)$ a partition by squared blocks is trivial if $n \gg P$, but large graphs are rarely dense. That is, E is generally sparse. Then for sparse graphs there are two major solutions: (1) partitioning edges by one vertex; (2) partitioning edges by their two vertices. Partitioning by vertex (i or j) will suffer when V degrees distribution is skewed: a few processors will contain most of the edges: parallel speedup will have a bottleneck. Hash-partitioning edges by both of their vertices can provide an even distribution, but in general the neighbors of a vertex will be assigned to a different processor, resulting in expensive data transfer during query processing. Therefore, this solution is detrimental to joins. The relative merits of each solution will depend on P, skewed degree distribution and network speed. On the other hand, the aggregation of the join result will require sorting in parallel by i, j, which can range from $O(n/P \log(n/P)))$ (sparse) to $O(n^3/P \log(n^3/P))$ (dense) assuming merge sort can evenly partition $E \bowtie E$ (a reasonable assumption for large n).

Notice the lower bounds match if $|S| = |E| = \Theta(n)$, but the upper bound for matrix-matrix multiplication has a much higher time complexity than matrix-vector multiplication when $|E| = \Theta(n^2)$.

4 Related Work

Research on analyzing graphs with relational queries has received little attention, compared to Big Data graph analytic systems (e.g. GraphX, Giraph, Neo4j). Important works include [11, 9, 12]. Reference [11] establishes a connection between graph analytics and SQL and justifies recursive queries, a classical problem, deserves to be revisited. Reference [9] shows a columnar DBMS is faster than graph analytic systems to compute some graph problems including reachability and PageRank. On the other hand, [12] compares different DBMS storage mechanisms to compute transitive closure and all-pairs shortest paths. This work shows columnar DBMSs have a performance advantage over rows and arrays. Neither work attempts to lay a theoretical foundation on the time and space complexity of relational queries to solve graph problems.

Being a classical topic in CS theory, research on recursive queries is extensive, especially with the Datalog language (which subsumes SQL) [1, 3, 8, 16, 13, 15, 14, 20, 18]. Comparatively, adapting such algorithms and optimizations to relational databases has received less attention [5, 10, 17]. Fundamental algorithms to evaluate recursive queries include Seminaïve [3] (the main algorithm used by us, linear number of iterations) and Logarithmic [17] (useful when most paths are long, logarithmic number of iterations). The Seminaïve algorithm solves the most general class of recursive queries, which eventually reach a fixpoint [2, 3]. Both algorithms iterate joining the input table with itself until no rows are added to the global result table (i.e., the iteration reaches a fixpoint). There is an independent line of research on adapting graph algorithms to compute transitive closure problem in a database system [2, 8]. The Direct algorithm [2] is an outstanding solution using in-place updates and bit operations in main memory instead of creating intermediate results on disk. Unfortunately, the Direct algorithm is incompatible with SQL query processing mechanisms and therefore has not made its way into relational DBMSs like Seminaïve.

5 Conclusions

We studied the evaluation of relational queries solving many fundamental graph problems iterating two forms of matrix multiplications: matrix-vector multiplication and matrix-matrix multiplication. Without loss of generality we focused on directed graphs. We studied two major aspects: (1) time and space complexity, considering basic relational operators (select, project, join) and (2) parallel processing, analyzing speedup and issues with unbalanced data partitioning. Parallel processing requires two major steps at each iteration: a parallel join and a parallel group-by agregation. Our results highlight a relational join is the most demanding operator, followed by group-by aggregations, expressed as an extended projection operator. Time complexity per iteration is best when the graph is sparse and its shape resembles a balanced tree and it is worst when the input graph is very dense (i.e. $m = O(n^2)$) or when intermediate tables gradually approximate complete subgraphs (i.e. subgraphs embedded in G become denser after each iteration). Parallel speedup is heavily impacted by a few vertices having a skewed degree distribution.

There are many open research issues, bridging theory and systems research. It is necessary to study graphs that have embedded cliques (complete subgraphs) in more depth. We need to study cardinality estimation of intermediate query results considering the graph is a union of subgraphs of diverse structure and density. Assuming the input graph is sparse but its transitive closure at some recursion depth is dense we need to characterize more precisely when a hash join or a sort-merge join is preferable. Finally, we aim to study harder graph problems requiring non-linear recursion (i.e. clique detection), which can help us identify SQL extensions to make it as powerful as Datalog.

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. W. Cabrera and C. Ordonez. Scalable parallel graph algorithms with matrix-vector multiplication evaluated with queries. *Distributed and Parallel Databases*, 35(3-4):335–362, 2017.
5. S. Dar and R. Agrawal. Extending SQL with generalized transitive closure. *IEEE Trans. Knowl. Eng.*, 5(5):799–812, 1993.
6. H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 1st edition, 2001.
7. T. Hastie, R. Tibshirani, and J.H. Friedman. *The Elements of Statistical Learning*. Springer, New York, 1st edition, 2001.
8. Y.E. Ioannidis, R. Ramakrishnan, and L. Winger. Transitive closure algorithms based on graph traversal. *ACM TODS*, 18(3):512–576, 1993.
9. A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker. Vertexica: Your relational friend for graph analytics! *Proc. VLDB Endow.*, 7(13):1669–1672, 2014.
10. I.S. Mumick, S.J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic conditions. *ACM TODS*, 21(1):107–155, 1996.
11. C. Ordonez. Optimization of linear recursive queries in SQL. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 22(2):264–277, 2010.
12. C. Ordonez, W. Cabrera, and A. Gurram. Comparing columnar, row and array dbmss to process recursive queries on graphs. *Information Systems*, 63:66–79, 2017.
13. R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. Implementation of the CORAL deductive database system. In *Proc. ACM SIGMOD*, pages 167–176, 1993.
14. 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.
15. S.Sakr, S.Elnikety, and Y.He. Hybrid query execution engine for large attributed graphs. *Inf. Syst.*, 41:45–73, 2014.
16. J.D. Ullman. Implementation of logical query languages for databases. *ACM Trans. Database Syst.*, 10(3):289–321, 1985.
17. P. Valduriez and H. Boral. Evaluation of recursive queries using join indices. In *Expert Database Systems*, pages 271–293, 1986.
18. M.Y. Vardi. Decidability and undecidability results for boundedness of linear recursive queries. In *ACM PODS Conference*, pages 341–351, 1988.
19. N. Yakovets, P. Godfrey, and J. Gryz. Evaluation of SPARQL property paths via recursive SQL. In *Proc. AMW*, 2013.
20. 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.