All-in-One: Graph Processing in RDBMSs Revisited

Kangfei Zhao, Jeffrey Xu Yu
The Chinese University of Hong Kong
Hong Kong, China
{kfzhao,yu}@se.cuhk.edu.hk

ABSTRACT

To support analytics on massive graphs such as online social networks, *RDF*, Semantic Web, etc. many new graph algorithms are designed to query graphs for a specific problem, and many distributed graph processing systems are developed to support graph querying by programming. In this paper, we focus on *RDBMS*, which has been well studied over decades to manage large datasets, and we revisit the issue how *RDBMS* can support graph processing at the *SQL* level. Our work is motivated by the fact that there are many relations stored in *RDBMS* that are closely related to a graph in real applications and need to be used together to query the graph, and *RDBMS* is a system that can query and manage data while data may be updated over time. To support graph processing, in this work, we propose 4 new relational algebra operations, MM-join, MV-join, anti-join, and union-by-update. Here, MM-join and MV-join are join operations between two matrices and between a matrix and a vector, respectively, followed by aggregation computing over groups, given a matrix/vector can be represented by a relation. Both deal with the semiring by which many graph algorithms can be supported. The anti-join removes nodes/edges in a graph when they are unnecessary for the following computing. The union-by-update addresses value updates to compute *PageRank*, for example. The 4 new relational algebra operations can be defined by the 6 basic relational algebra operations with group-by & aggregation. We revisit *SQL* recursive queries and show that the 4 operations with others are ensured to have a fixpoint, following the techniques studied in DATALOG, and enhance the recursive with clause in *SQL*'99. We conduct extensive performance studies to test 10 graph algorithms using 9 large real graphs in 3 major *RDBMS*s. We show that *RDBMS*s are capable of dealing with graph processing in reasonable time. The focus of this work is at *SQL* level. There is high potential to improve the efficiency by main-memory *RDBMS*s, efficient join processing in parallel, and new storage management.

1. INTRODUCTION

Graph processing has been extensively studied to respond the needs of analyzing massive online social networks, *RDF*, Semantic Web, knowledge graphs, biological networks, and road networks. A large number of graph algorithms have been used/proposed/revisited. Such graph algorithms include *BFS* (Breadth-First Search) [17], *Connected-Component* [48], shortest distance computing [17], topological sorting [31], *PageRank* [36], *Random-Walk-with-Restart* [36], *SimRank* [28], *HITS* [36], *Label-Propagation* [46], *Maximal-Independent-Set* [40], and *Maximal-Node-Matching* [43], to name a few. In addition to the effort to design efficient graph algorithms to analyze large graphs, many distributed graph processing systems have been developed using the vertex-centric programming on *BSP* (Bulk Synchronous Parallel). A recent survey can be found in [38]. Such distributed graph processing systems provide a framework on which users can implement graph algorithms to achieve high efficiency. Both new graph algorithms and distributed graph processing systems focus on efficiency. On the other hand, graph query languages have also been studied [60, 12, 21, 25]. As surveyed in [60], such graph query languages include *Lorel*, *StruQL*, *UnQL*, **G**, $\mathbf{G}^+$, *GraphLog*, *G-Log*, *SoSQL* that express the conjunctive query, regular path query, or combination of the both. Extending regular expressions to query both path and data is recently studied [35]. And *GraphQL* is a graph query language proposed based on graph algebra [25]. In addition, DATALOG has been revisited to support graph analytics. The representative systems are *DeALS* [54, 53, 52] and *SociaLite* [50, 51]. The scaling DATALOG for machine learning is studied in [14].

In this paper, we focus on *RDBMS*, which has been well studied over decades to manage large datasets, and we revisit the issue how *RDBMS* can support graph processing. Our work is motivated by the following. First, *RDBMS* is to manage various application data in relations as well as to query data in relations efficiently using the sophisticated query optimizer. A graph may be a labeled graph with node/edge label, and it is probable that additional information is associated with the graph (e.g. attributed graphs). A key point is that we need to provide a flexible way for users to manage and query a graph together with many relations that are closely related to the graph. The current graph systems are developed for processing but not for data management. We need a system to fulfill both. Second, there is a requirement to query graphs. In the literature, many new graph algorithms are studied to query a specific graph problem. And the current graph processing systems developed do not have a well-accepted graph query language for querying graphs. In other words, it needs coding, when there is a need to compute a graph algorithm based on the outputs of other graph algorithms. Instead of designing a new graph query language, a question to be asked is why *SQL* cannot be used to query graphs, given the techniques developed to process *SQL* queries. In this work, we revisit recursive *SQL* queries [19] and show that a large class of graph algorithms can be supported by *SQL* in *RDBMS*s. To the best of

SIGMOD'17, May 14-19, 2017, Chicago, Illinois, USA
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3035918.3035943

our knowledge, it is not well discussed on what graph analytics can be supported and how to support them by *SQL*. We give our solution in this work. Our focus is on supporting graph processing at the *SQL* level. There is high potential to improve the efficiency by main-memory *RDBMS*s, efficient join processing in parallel [8], and new storage management [30].

The issue of supporting graph algorithms in *RDBMS* at *SQL* level is the issue how recursive *SQL* can be used to support graph algorithms. There are two main concerns regarding recursive *SQL* queries. One is a set of operations that are needed to support a large pool of graph algorithms and can be used in recursive *SQL* queries. The other is the way to ensure the recursive *SQL* queries can obtain a unique answer. The two concerns are interrelated.

The main contributions of this work are summarized below.

First, for supporting graph algorithms, we propose to use a set of 4 relational algebra operations, MM-join, MV-join, anti-join, and union-by-update. Here, MM-join and MV-join are join operations between two matrices and between a matrix and a vector, respectively, followed by aggregation computing over groups, given a matrix/vector can be represented by a relation. Both are used to deal with the semiring by which many graph algorithms can be supported [34]. The anti-join removes nodes/edges in a graph when they are unnecessary for the following computing, and it serves as a selection. The union-by-update addresses value updates which are needed in many iterative graph computing tasks, like *PageRank*, *SimRank*, and *HITS*. It is worth mentioning that [41, 33] discuss MV-join, MM-join is similar to MV-join, anti-join is the complement of the semi-join, and union-by-update is a new operation proposed in this work. We show that the 4 relational algebra operations can be defined by the 6 basic relational algebra operations together with group-by & aggregation. However, none of them can be used in recursive *SQL* queries, in general, as specified by *SQL*'99, because they are negation-like operations.

Second, we revisit recursive queries defined in *SQL*'99. *SQL*'99 supports linear recursion and mutual recursion, but it does not support nonlinear recursion. The existing *RDBMS*s support linear recursion for monotonic *SQL* queries only, and do not support mutual recursion. There are two separated issues. One is how to support non-monotonic recursive *SQL* queries since the 4 operations are all non-monotonic in nature. By adopting the DATALOG techniques given in [62, 63, 11], we show that a recursive *SQL* query using the relational algebra operations, including MM-join, MV-join, anti-join, and union-by-update, has a fixpoint. That implies such 4 operations can be used in a recursive *SQL* query, which can be linear, nonlinear, or have mutual recursion. The other is related to linear, nonlinear, and mutual recursion. Even though many graph algorithms can be specified by linear recursion, some well-known graph algorithms such as *HITS* cannot be specified by linear recursion, where nonlinear recursive queries are easy to follow and converge fast, but they are hard to implement as pointed by Widom (https://www.youtube.com/watch?v=0n9kScLFyIo). To support a large pool of graph algorithms, we allow nonlinear and mutual recursion. The efficiency issue can be addressed by incorporating the new algorithms and implementations in main-memory and on new storage systems.

Third, we enhance the recursive with clause in *SQL*'99. Such with in *SQL*'99 is not designed for iterative graph algorithms to update node/edge values in every iteration. We propose with+ which supports iterative graph algorithms by making use of the operation of union-by-update, and it supports linear, nonlinear and mutual recursion.

Fourth, we conduct extensive performance studies to test 10 graph algorithms using 9 large real graphs in 3 main *RDBMS*s. All the testing is done by translating the enhanced recursive with statement to *SQL/PSM*, which is an *SQL* standard to support procedures and functions defined using looping and condition checking. By our testing, we show that *RDBMS*s are capable of dealing with graph processing in reasonable time.

The paper is organized as follows. Section 2 reviews the related works. Section 3 discusses the recursion handling by *SQL* in *RDBMS*s. In Section 4, we present our approach to support graph processing by *SQL* followed by the discussion on how to ensure the fixpoint semantics in Section 5. We give the enhanced with clause and its implementation details in Section 6. We report our extensive performance studies using graph algorithms over real datasets by *RDBMS*s, and conclude our work in Section 8.

2. RELATED WORK

Graph Query Languages: Graph query languages have been studied. A survey on query languages for graph databases can be found in [60], which covers conjunctive query (CQ), regular path query (RPQ), and CRPQ combining CQ and RPQ. Also, it surveys a large number of languages including *Lorel*, *StruQL*, *UnQL*, **G**, $\mathbf{G}^+$, *GraphLog*, *G-Log*, *SoSQL*, etc. Barceló investigates the expressive power and complexity of graph query languages [12]. Libkin et al. in [35] study how to combine data and topology by extending regular expressions to specify paths with data. There are several new attempts to query graphs. Gao et al. in [21] propose a graph language *GLog* on Relational-Graph, which is a data model by mixing relational and graph data. A *GLog* query is converted into a sequence of MapReduce jobs to be processed on distributed systems. Jindal and Madden propose *graphiQL* in [29] by exploring a way to combine the features of *Pregel* (vertex-centric programming) and *SQL*. He and Singh in [25] propose the language *GraphQL* on graph algebra which deals with graphs with attributes as a basic unit. The operations in the graph algebra include selection, Cartesian product, and composition. Salihoglu and Widom in [49] propose *HeLP*, a set of basic operations needed in many graph processing systems.

Recursive SQL Queries: *SQL*'99 supports recursive queries [39, 19]. As mentioned, in supporting graph algorithms, there are two main issues regarding recursive *SQL* queries: a set of operations that can be used in recursive *SQL* queries, and a way to ensure unique solution by recursive *SQL* queries. For the former, Cabrera and Ordonez in [15] and Kang et al. in [33] discuss an operation to multiply a matrix with a vector using joins and group-by & aggregation. Cabrera and Ordonez in [15] also discuss semiring for graph algorithms, and give a unified algorithm which is not in *SQL*. For the latter, recursive query processing is well discussed in [9]. Ordonez et al. in [42] compare *SQL* recursive query processing in columnar, row and array databases. The main issue to be studied in this work is that many graph algorithms need to use aggregation and negation to get an answer, but aggregations and negations cannot be used within a recursive *SQL* query for ensuring that an *SQL* query can get a unique solution. Recently, Aranda et al. in [10] study broadening recursion in *SQL*, but they do not deal with negation and aggregation. The *SQL* level optimizations for computing transitive closures are discussed in [41], with its focus on monotonic aggregation for transitive closures. However, aggregation and negation in general are needed for a large pool of graph algorithms. Ghazal et al. propose an adaptive query optimization scheme for the recursive query in *Teradata*, which employs multi-iteration pre-planning and dynamic feedback to take advantage of global query optimization and pipelining [23].

Graph Processing in RDBMSs: Supporting graph processing in

```
1. with
2. TC (F, T) as (
3.       (select F, T from E)
4.       union all
5.       (select TC.F, E.T from TC, E where TC.T = E.F))
```

Figure 1: The recursive with **statement**

*RDBMS*s have been studied. Srihari et al. in [55] introduce an approach for mining dense subgraphs in a *RDBMS*. Gao et al. in [20] leverage the window functions and the merge statement in *SQL* to implement shortest path discovery in *RDBMS*. Zhang et al. in [65] provide an *SQL*-based declarative query language *SciQL* to perform array computation in *RDBMS*s. Fan et al. in [18] propose *GRAIL*, a syntactic layer converting graph queries into *SQL* script. *GraphGene* [61] is a system for users to specify graph extraction layer over relational databases declaratively. *MADLib* is designed and implemented to support machine learning, data mining and statistics on database systems [16, 26]. In [30], *Vertica* relational database is studied as the platform for vertex-centric graph analysis. In [56], a graph storage system *SQLGraph* is designed, which combines the relational storage for adjacency information with *JSON* for vertex and edge properties. It shows that it can outperform popular *NoSQL* graph stores. Aberger et al. in [8] develop a graph pattern engine, called *EmptyHead*, to process graph patterns as join processing in parallel.

Deductive Database Systems: DATALOG systems have been developed that use DATALOG as its language to process graphs. However, DATALOG is not for data management. A survey of early deductive database systems can be found in [47]. *LDL++* is a deductive database system in which negation and aggregation handling in recursive rules are addressed [62, 11]. Based on *LDL++*, a new deductive application language system *DeALS* is developed to support graph queries [54], and the optimization of monotonic aggregations is further studied [53]. *SociaLite* [50] allows users to write high-level graph queries based on DATALOG that can be executed in parallel and distributed environments [51]. DATALOG for machine learning is studied with *Pregel* and map-reduce-update style programming [14], and it is investigated for big data analytics on *Spark* [52]. In this work, we introduce the DATALOG techniques into *RDBMS*s to deal with recursive *SQL* queries, since DATALOG has greatly influenced the recursive *SQL* query handling.

3. THE RECURSION IN RDBMS

Over decades, *RDBMS*s have provided functionality to support recursive queries, based on *SQL*'99 [39, 19], on which DATALOG has significant influence. The recursive queries are expressed using with clause in *SQL*. We introduce the with clause following the discussions given in [22].

with R **as** $\langle$ R initialization $\rangle$ $\langle$ recursive querying involving R $\rangle$

Here, the recursive with clause defines a temporary recursive relation R in the initialization step, and queries by referring the recursive relation R iteratively in the recursive step until R cannot be changed. As an example, the edge transitive closure can be computed using with over the edge relation $E(F, T)$, where F and T are for "From" and "To". As shown in Fig. 1, the recursive relation is named TC. Initially, the recursive relation TC is defined to project the two attributes, F and T, from the relation E (line 3). Then, the query in every iteration is to union TC computed and a relation with two attributes $TC.F$ and $E.T$ by joining the two relations, TC and E, over the join condition $TC.T = E.F$ (line 5).

SQL'99 restricts recursion to be a linear recursion and allows mutual recursion in a limited form [39]. In brief, a linear recursion

Features		*PostgreSQL*	*DB2*	*Oracle*
A	Linear Recursion	✓	✓	✓
	Nonlinear Recursion	✗	✗	✗
	Mutual Recursion	✗	✗	✗
B	Initial Step	✓	✓	✓
	Recursive Step	✗	✓	✗
C	Between initial queries	✓	✓	✓
	Across initial & recursive queries	✓	✗	✗
	Between recursive queries	–	✗	–
D	Negation	✗	✗	✗
	Aggregate functions	✗	✗	✗
	group by, having	✗	✗	✗
	partition by	✓	✓	✓
	distinct	✓	✗	✗
	General functions	✓	✗	✓
	Analytical functions	✓	✗	✓
	Subqueries without recursive ref	✓	✓	✓
	Subqueries with recursive ref	✗	✗	✗
E	Infinite loop detection	✗	✗	✓
	Cycle detection	✗	✗	✓
	cycle	✗	✗	✓
	search	✗	✗	✓

Table 1: The with **Clause Supported by** ***RDBMS*s**

means that a recursive relation is invoked at most once in an iteration, and a nonlinear recursion means that a recursive relation is referred more than once in the from clause. A mutual recursion indicates the situation that there are two recursive relations, R_A and R_B, where R_A invokes R_B directly or indirectly, and R_B invokes R_A directly or indirectly at the same time. Among the linear recursion, *SQL*'99 only supports monotonic queries, which is known as the monotonicity. In the context of recursion, a monotonic query means that the result of a recursive relation in any iteration does not lose any tuples added in the previous iterations. Such monotonicity ensures that the recursion ends at a fixpoint with a unique result. The definition of monotonicity can be found in [57], which we also give in the Appendix. As given in Theorem 3.3 in [57], union, select, projection, Cartesian product, natural joins, and θ-joins are monotone. On the other hand, negation is not monotone [22]. In *SQL*, the operations such as except, intersect, not exists, not in, $<>$ some, $<>$ all, distinct are the operations leading to negation. Also, aggregation can violate the monotonicity. It is worth mentioning that *SQL*'99 does not prohibit using negation in recursive queries completely, as long as the monotonicity of queries is maintained. In other words, the monotonicity is ensured if the negation is only applied to the relations that are completely known or computed prior to processing the result of the recursion. This is known as stratified negation.

We introduce stratification over the dependency graph defined in Definition 9.1 in the Appendix, where an edge exists from g to h if h depends on g to compute. The dependency graph (Definition 9.1) is defined over *SQL*, and it is equivalent to the predicate dependency graph defined over DATALOG [63, 57] over which the stratification is discussed. Recall that a DATALOG program consists of a set of rules. Here, a rule r is in the form of "h :- $g_1, g_2, \cdots, g_n$", where h is the head of the rule r, g_i, for $1 \leq i \leq n$, is a subgoal of the rule r, and a comma between subgoals is a logical conjunction $\wedge$. The head h is P and a subgoal g_i is either P or $\neg P$, where P is a predicate and $\neg$ is for negation. The predicate can be either a user-defined predicate, which is the head of another rule, or a built-in predicate over terms (constants, variables, functions). A built-in predicate can be a base relation that exists in the database or a predicate like the one used in selection (σ). The dependency graph (Definition 9.1) is a predicate dependency graph by treating nodes in the dependency graph as predicates. In the following, we use relation and predicate used in the context of DATALOG interchangeably. The definition of stratification can be found in [63], which we give it in the Appendix.

SQL Recursion Handling in RDBMS: *SQL*'99 supports stratified negation. Below, we discuss *SQL* recursion handling in *RDBMS*s, following the similar discussions given in [44] in which Przymus et al. survey recursive queries handling in *RDBMS*s. We focus on *Oracle* (11gR2) [6], *IBM DB2 10.5 Express-C* [4], and *PostgreSQL* (9.4) [7], where *Oracle* is not listed in [44]. We discuss the features related to recursive query processing in 5 categories. (A) linear/nonlinear/mutual recursion. (B) multiple queries used in the with clause, (C) the set operations other than union all that can be used to separate queries in the with clause, (D) the restrictions on group by, aggregate function, and general functions in the recursive step, and (E) the function to control the looping. Table 1 shows the summary, where "✓", "✗", and "–" denote the corresponding functionality is supported, prohibited, and not applicable, respectively, in the with clause.

We discuss the 5 categories. For (A), all the 3 *RDBMS*s support linear recursion but do not support nonlinear and mutual recursion. For (B) multiple queries used in the with clause, all the 3 *RDBMS*s support multiple queries in the initialization step without restrictions. In the recursive step, *DB2* allows multiple queries, whereas *PostgreSQL* and *Oracle* do not. The category (C) indicates which set operations other than union all can be used to separate 2 queries. Such set operations are union, except, and intersect. There are 3 cases. For 2 queries in the initialization step, all the set operations can be used. For 2 queries that one is in the initialization step and one is in the recursive step, *PostgreSQL* is the only one that can use union instead of union all. Note that union is the union operation that eliminates duplicates. For 2 queries in the recursive step, none except for union all can be used. We further discuss the restrictions in the recursive query (D). None of the 3 *RDBMS*s support negation and aggregation as well as group by and having, since they violate the monotonicity. Although aggregate functions are forbidden, analytical functions (partition by) specified in a window for aggregate functions can be used in *PostgreSQL* as well as *Oracle*. Here, when new tuples are added, analytical functions do not lose accumulated result of previous iterations. The analytical functions are applied only to the subset of data used in the current iteration, not the entire set of data used in the recursive querying step. *DB2* does not allow any general arithmetic functions and analytical functions used in the recursive querying step. With the concern of the monotonicity, among the 3 *RDBMS*s, *PostgreSQL* supports distinct which is to remove duplicates in the select clause. We show how *PostgreSQL* supports *PageRank* using partition by and distinct in Fig. 9 in the Appendix. It is important to note that partition by without distinct cannot be used to support graph processing in general, since the answer may be incorrect. That is because that every tuple in a group has a tuple in the resulting relation if partition by is used, which is different from group by that only one tuple per group is in the result. All the 3 *RDBMS*s allow subqueries (including subqueries with exists and not exists) in the recursive querying step on the condition that the subqueries cannot refer to the recursive relation. Finally, we discuss the control mechanisms used in *Oracle* to control the looping (E). *Oracle* provides users with two auxiliary clauses, namely, search and cycle, for the recursive with clause. The search specifies a tuple search order, and the cycle marks a cycle in the recursion. When a cycle is detected for a certain tuple, the recursion will terminate for this tuple but will continue for other noncyclic tuples. *Oracle* explicitly reports a warning if a cycle is discovered. On the contrary, *PostgreSQL* and *DB2* do not provide automatic cycle detection. Table 1 summarizes the with clause supported by *RDBMS*s.

Handing Recursion by PSM in RDBMS: There is another way to implement recursion, which is *SQL/PSM* (Persistent Stored Modules) included in *SQL* standard [57]. By *SQL/PSM* (or *PSM*), users can define functions/procedures in *RDBMS*s, and call such functions when querying. In a function/procedure definition, users can declare variables, create temporary tables, insert tuples, and use looping where conditions can be specified to exit (or leave) the loop. *PSM* provides users with a mechanism to issue queries using a general-purpose programming language.

4. THE POWER OF ALGEBRA

In this paper, we model a graph as a weighted directed graph $G = (V, E)$, where V is a set of nodes and E is a set of edges. A node is associated with a node-weight and an edge is associated with an edge-weight, denoted as $\omega(v_i)$ and $\omega(v_i, v_j)$, respectively. In the following, we use n and m to denote the number of nodes and the number of edges for a graph G. A graph can be represented in matrix form. The nodes with node-weights can be represented as a vector of n elements, denoted as V. The edges with edge-weights can be represented as a $n \times n$ matrix, denoted as M, where its M_{ij} value can be 1 to indicate that there is an edge from v_i to v_j, or the value of the edge weight. Such matrices and vectors have their relation representation. Let V and M be the relation representation of vector V and matrix M, such that $V(ID, vw)$ and $M(F, T, ew)$. Here, ID is the tuple identifier in V. F and T, standing for "From" and "To", form a primary key in M. vw and ew are the node-weight and edge-weight respectively.

4.1 The Four Operations

We discuss a set of 4 relational algebra operations, MM-join, MV-join, anti-join, and union-by-update. Here, MM-join and MV-join support the semiring by which many graph algorithms can be supported. The anti-join is used to remove nodes/edges in a graph when they are unnecessary in the following computing and serves as a selection. The union-by-update is used to deal with value updates in every iteration to compute a graph algorithm, e.g., *PageRank*. It is worth noting that there is no such an operation like union-by-update in relational algebra.

We show that all the 4 relational algebra operations can be defined using the 6 basic relational algebra operations (selection (σ), projection (Π), union ($\cup$), set difference ($-$), Cartesian product ($\times$), and rename (ρ)), together with group-by & aggregation. For simplicity, below, we use "$R_i \rightarrow R_j$" for the rename operation to rename a relation R_i to R_j, and use "$\leftarrow$" for the assignment operation to assign the result of a relational algebra to a temporal relation.

We explain why we need the 4 operations which can be supported by the relational algebra because they do not increase the expressive power of relational algebra. First, it is known that relational algebra can support graph algorithms. However, it is not well discussed how to support explicitly. The set of 4 operations is such an answer. Second, it is known that recursive query is inevitable. In other words, new operations cannot function if they cannot be used in recursive *SQL* queries in *RDBMS*. The 4 operations are the non-monotonic operations that cannot be used in recursive *SQL* queries allowed in *SQL*'99. With the explicit form of the 4 operations, in this work, we show that they can be used in recursive *SQL* queries which lead to a unique answer (fixpoint) by adopting the DATALOG techniques. Third, with the explicit form as a target, we can further study how to support them efficiently.

In this work, we do not include some matrix operations in the set, if they can be used in recursive *SQL* queries. For example, the transpose of a matrix M, denoted as M^{T}, is such an operation, which can be easily handled by renaming (ρ) in the relational algebra as $\rho_M(\Pi_{T,F,ew}M)$. In other words, this is to rename the F (T)

value to be T (F) value, respectively, in the relation representation M for matrix M. In addition, we do not include some matrix operations if they need recursive computing. For example, the matrix inverse, denoted as M^{-1} is such an operation. We discuss the 4 operations below.

To support graph analytics, the algebraic structure, namely semiring, is shown to have sufficient expressive power to support many graph algorithms [34, 15]. The semiring is a set of $\mathcal{M}$ including two identity elements, **0** and **1**, with two operations: addition $(+)$ and multiplication $(\cdot)$. In brief, (1) $(\mathcal{M}, +)$ is a commutative monoid with **0**, (2) $(\mathcal{M}, \cdot)$ is a monoid with **1**, (3) the multiplication $(\cdot)$ is left/right distributes over the addition $(+)$, and (4) the multiplication by **0** annihilates $\mathcal{M}$. Below, A and B are two 2×2 matrix, and C is a vector with 2 elements.

$$\mathsf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \quad \mathsf{B} = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}, \quad \mathsf{C} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$$

The matrix-matrix (matrix-vector) multiplication $(\cdot)$, and matrix entrywise sum $(+)$ are shown below.

$$\begin{aligned} \mathsf{A} \cdot \mathsf{B} &= \begin{pmatrix} a_{11} \odot b_{11} \oplus a_{12} \odot b_{21} & a_{11} \odot b_{12} \oplus a_{12} \odot b_{22} \\ a_{21} \odot b_{11} \oplus a_{22} \odot b_{21} & a_{21} \odot b_{12} \oplus a_{22} \odot b_{22} \end{pmatrix} \\ \mathsf{A} + \mathsf{B} &= \begin{pmatrix} a_{11} \oplus b_{11} & a_{12} \oplus b_{12} \\ a_{21} \oplus b_{21} & a_{22} \oplus b_{22} \end{pmatrix} \\ \mathsf{A} \cdot \mathsf{C} &= \begin{pmatrix} a_{11} \odot c_1 \oplus a_{12} \odot c_2 \\ a_{21} \odot c_1 \oplus a_{22} \odot c_2 \end{pmatrix} \end{aligned}$$

We focus on the multiplication $(\cdot)$, since it is trivial to support the addition $(+)$ in relational algebra. Let A and B be two $n \times n$ matrices, and C be a vector with n elements. For the multiplication $\mathsf{AB} = \mathsf{A} \cdot \mathsf{B}$, and $\mathsf{AC} = \mathsf{A} \cdot \mathsf{C}$, we have the following.

$$\mathsf{AB}_{ij} = \bigoplus_{k=1}^{n} \mathsf{A}_{ik} \odot \mathsf{B}_{kj} \tag{1}$$

$$\mathsf{AC}_{i} = \bigoplus_{k=1}^{n} \mathsf{A}_{ik} \odot \mathsf{C}_{k} \tag{2}$$

Here, M_{ij} is the value at the i-th row and j-th column in the matrix M, and V_i is the element at the i-th row in the vector V.

Let A and B be the relation representation for a $n \times n$ matrix, and C be a relation representation for a vector with n elements. The relations, A, B, and C are shown in Table 8 for 2×2 matrices A and B, and 2-element vector C, in the Appendix. To support matrix-matrix multiplication (Eq. (1)) and matrix-vector multiplication (Eq. (2)), we define two aggregate-joins, namely, MM-join and MV-join. The first aggregate-join, called MM-join, is used to join two matrix relations A and B, to compute $\mathsf{A} \cdot \mathsf{B}$. The MM-join is denoted as $A \overset{\oplus(\odot)}{\underset{A.T=B.F}{\bowtie}} B$, and it is defined by the following relational algebra.

$$A \overset{\oplus(\odot)}{\underset{A.T=B.F}{\bowtie}} B = {}_{A.F,B.T}\,\mathcal{G}_{\oplus(\odot)}(A \underset{A.T=B.F}{\bowtie} B) \tag{3}$$

The second aggregate-join, called MV-join, is used to join a matrix relation and a vector relation, A and C, to compute $\mathsf{A} \cdot \mathsf{C}$. The MV-join is denoted as $A \overset{\oplus(\odot)}{\underset{T=ID}{\bowtie}} C$, and it is defined by the following relational algebra.

$$A \overset{\oplus(\odot)}{\underset{T=ID}{\bowtie}} C = {}_{F}\,\mathcal{G}_{\oplus(\odot)}(A \underset{T=ID}{\bowtie} C) \tag{4}$$

Here, ${}_X\mathcal{G}_Y$ is a group-by & aggregation operation to compute the aggregate function defined by Y over the groups by the attributes specified in X. Note that MV-join is discussed in [41, 33], and MM-join is similar to MV-join.

There are two steps to compute MM-join. The first step is to join A and B by the join condition $A.T = B.F$. This step is to join the k value in order to compute $\odot$ for $\mathsf{A}_{ik} \odot \mathsf{B}_{kj}$ as given in Eq. (1). The second step is to do group-by & aggregation, where the group-by attributes are the attributes that are in the primary key but do not appear in the join condition, namely, $A.F$ and $B.T$, and the aggregate function is to compute Eq. (1). In a similar fashion, there are two steps to compute MV-join. The first step is to join A and C by the join condition $A.T = C.ID$. This step is to join the k value in order to compute $\odot$ for $\mathsf{A}_{ik} \odot \mathsf{C}_k$ as given in Eq. (2). The second step is to do group-by & aggregation, where the group-by attribute is the attribute $A.F$, and the aggregate function is to compute Eq. (2).

We adopt the anti-join, $R \bar{\ltimes} S$, which is defined as the result of R that cannot be semi-joined by S, such that $R - (R \ltimes S)$.

In addition, we propose a new union operation, called union-by-update, for the purpose of updating values in either a matrix or a vector, denoted as $\uplus$. Let $R(A, B)$ and $S(A, B)$ be two relations, where A and B are two sets of attributes. $R \uplus_A S$ is a relation, $RS(A, B)$. Let r be a tuple in R and s be a tuple in S. Different from the conventional union operation $(\cup)$ where two tuples are identical if $r = s$, with $R \uplus_A S$, two tuples, r and s, are identical if $r.A = s.A$. The union-by-update is to update the B attributes values of r by the B attributes values of s if $r.A = s.A$. In other words, if $r.A = s.A$, then s is in RS but not r. There are 2 cases that r and s do not match. If s does not match any r, then s is in RS. If r does not match any s, then r is in RS. It is worth noting that there can be multiple r match multiple s on the attributes A. We allow multiple r to match a single tuple s, but we do not allow multiple s to match a single r, since the answer is not unique. When A attributes in both R and S are defined as the primary key, there is at most one pair of r and s matches.

The 4 operations are independent among themselves, since we can discover a property that is possessed by one operation but is not possessed by the composition of the other three only [57]. The property of anti-join ($R \bar{\ltimes} S$) is that the resulting relation must not contain tuples in S. The property of union-by-update ($R \uplus S$) is that the resulting relation must contain tuples in S. The MM-join returns a relation with the same arity of the edge table, while the MV-join returns a relation with the same arity of the node table.

4.2 Relational Algebra plus While

To support graph processing, a control structure is needed in addition to the relational algebra operations discussed. We follow the "algebra + while" given in [9].

```
initialize $R$
**while** ($R$ changes) { $\cdots$; $R \leftarrow \cdots$ }
```

In brief, in the looping, R may change by the relational algebra in the body of the looping. The looping will terminate until R becomes stable. As discussed in [9], there are two semantics for "algebra + while", namely, noninflationary and inflationary. Consider the assignment, $R \leftarrow \mathcal{E}$, which is to assign relation R by evaluating the relational algebra expression $\mathcal{E}$. By the noninflationary, the assignment can be destructive in the sense that the new value will overwrite the old value. By the inflationary, the assignment needs to be cumulative. For the termination of the looping, as pointed out in [9], explicit terminating condition does not affect the expressive power. In this work, the conventional union $(\cup)$ is for the inflationary semantics, whereas union-by-update $(\uplus)$ is for the noninflationary semantics.

In this work, the expressive power, and the complexity remain unchanged as given in [9] under the scheme of "algebra + while", because the 4 operations added can be supported by the existing relational algebra operations. From the viewpoint of relational algebra, we can support all basic graph algorithms, including those that need aggregation (Table 2) but excluding those complicated algorithms for spectral analytics that need matrix inverse. The 4 operations make it clear how to support graph algorithms in relational algebra. In particular, all graph algorithms, that can be expressed by the semiring, can be supported under the framework of "algebra + while" and hence *SQL* recursion to be discussed in this work.

4.3 Supporting Graph Processing

We show how to support graph algorithms by the "algebra + while" approach, using MM-join and MV-join, anti-join, union-by-update, as well as other operations given in relational algebra. For simplicity, we represent a graph $G = (V, E)$ with n nodes by an $n \times n$ E and a vector V with n elements. We represent the vector V by a relation $V(ID, vw)$, where ID is the tuple identifier for the corresponding node with value vw associated. Moreover, we represent the matrix E by a relation $E(F, T, ew)$, where F and T, standing for "From" and "To", form a primary key in E, which is associated with an edge value ew. Below, to emphasize the operations in every iteration, we omit the while looping. Some graph algorithms can be computed by either union or union-by-update. We focus on union-by-update.

First, consider *BFS* (Breadth First Search). In the relation E, the value $ew = 1$ if $\mathsf{E}_{ij} = 1$ for an edge from v_i to v_j, and otherwise 0. Initially, suppose that V is a relation where the vw value is 1 for the tuple representing the source node to start *BFS*, and 0 for the other tuples. The following relational algebra is for $\mathsf{E}^\mathsf{T} \cdot \mathsf{V}$.

$$V \leftarrow \rho_V(E \overset{max(vw*ew)}{\underset{F=ID}{\bowtie}} V) \tag{5}$$

In the resulting relation, a tuple with $vw = 1$ indicates that the corresponding node can be traversed by *BFS*. It is worth noting that $E \overset{\oplus(\odot)}{\underset{T=ID}{\bowtie}} V$ is for computing $\mathsf{E} \cdot \mathsf{V}$, whereas $E \overset{\oplus(\odot)}{\underset{F=ID}{\bowtie}} V$ is for computing $\mathsf{E}^\mathsf{T} \cdot \mathsf{V}$. Here, to deal with the semiring for computing *BFS*, the multiplication $\odot$ and the addition $\oplus$ can be defined as $*$ and max. In other words, if v_i is visited and there is an edge from v_j to v_i, then v_j's value should be 1 as $1 * 1$. There are multiple nodes that have an edge to v_j, the max is to take one of them over the group-by attribute $E.T$. Note that in Eq. (5), $\leftarrow$ means assignment, which updates the values in V (union-by-update) rather than inserting new tuples into V.

In a similar fashion, *Connected-Component* can be computed. Here, in a graph G, we show that all nodes in a connected-component are with the same vw value. In order to do so, we have a relation V, where the vw value is the ID of the tuple in V, initially. By the following relational algebra,

$$V \leftarrow \rho_V(E \overset{min(vw*ew)}{\underset{F=ID}{\bowtie}} V) \tag{6}$$

in the resulting relation V, a connected component is determined by a unique value, which is the smallest ID of the nodes in the same connected component.

We show the relational algebra for two shortest distance algorithms. One is the *Bellman-Ford* algorithm to compute the single source shortest distances, and the other is the *Floyd-Warshall* algorithm to compute all source shortest distances. To implement *Bellman-Ford*, initially, let V be a relation where the vw is 0 for the tuple representing the source node to start, and ∞ for the other tuples. Here, vw value of a tuple indicates the distance from the source node to itself.

$$V \leftarrow \rho_V(E \overset{min(vw+ew)}{\underset{F=ID}{\bowtie}} V) \tag{7}$$

To implement *Floyd-Warshall*, the relational algebra is given as follows.

$$E \leftarrow \rho_E((E \rightarrow E_1) \overset{min(E_1.ew+E_2.ew)}{\underset{E_1.T=E_2.F}{\bowtie}} (E \rightarrow E_2)) \tag{8}$$

Next, we show the relational algebra for computing *PageRank*, *Random-Walk-with-Restart*, *SimRank*, and *HITS*. The relational algebra for *PageRank* is given below.

$$V \leftarrow \rho_V(E \overset{f_1(\cdot)}{\underset{T=ID}{\bowtie}} V) \tag{9}$$

Here, $f_1(\cdot)$ is a function to calculate $c*sum(vw*ew)+(1-c)/n$, where c is the damping factor and n is the total number of tuples in V. Note, $vw * ew$ is computed when joining the tuples from E and V, regarding $\odot$, and the aggregate function sum is computed over groups, given $vw * ew$ computed, along with other variables in $f_1(\cdot)$, regarding $\oplus$. Consider *Random-Walk-with-Restart*. Let $P(ID, vw)$ be a relation, where a non-zero ew value indicates its probability of the corresponding node to restart.

$$V \leftarrow \rho_V(\Pi_{V.ID, f_2(\cdot)+(1-c)*P.vw}(E \overset{f_2(\cdot))}{\underset{S.T=ID}{\bowtie}} V) \underset{V.ID=P.ID}{\bowtie} P) \tag{10}$$

Here, $f_2(\cdot)$ is a function of $c*sum(vw*ew)$, since *Random-Walk-with-Restart* is the general case of *PageRank*.

$$\begin{aligned}
R_1 &\leftarrow \rho_K(E \overset{sum(E.ew*K.ew)}{\underset{E.T=K.T}{\bowtie}} K) \\
R_2 &\leftarrow \rho_K(R_1 \overset{sum(R_1.ew*E.ew)}{\underset{R_1.F=E.T}{\bowtie}} E) \\
K &\leftarrow \Pi_{R_2.F, R_2.T, max((1-c)*R_2.ew, I.ew)}(R_2 \underset{\substack{R_2.T=I.T \\ R_2.F=I.F}}{\bowtie} I)
\end{aligned} \tag{11}$$

For *SimRank*, let $I(F, T, ew)$ be a relation, where $ew = 1$ if $F = T$, and otherwise 0. Let $K(F, T, ew)$ be I initially. It can be computed by Eq. (11). We discuss *HITS* below in detail. Let $H(ID, h, a)$ be a relation, where h and a are the hub and authority value of the ID. Initially, all the values of h and a are 1.

$$\begin{aligned}
H_h &\leftarrow \Pi_{ID,h} H \\
R_a &\leftarrow \rho_{ID,a}(H_h \overset{sum(h*ew)}{\underset{ID=T}{\bowtie}} E) \\
R_h &\leftarrow \rho_{ID,h}(R_a \overset{sum(a*ew)}{\underset{ID=F}{\bowtie}} E) \\
R_{ha} &\leftarrow \Pi_{R_a.ID,h,a}(R_a \underset{R_a.ID=R_h.ID}{\bowtie} R_h) \\
R_n &\leftarrow \rho_{nh,na}(\mathcal{G}_{sum(h*h),sum(a*a)} R_{ha}) \\
H &\leftarrow \rho_H(\Pi_{ID,h/sqrt(nh),a/sqrt(na)}(R_{ha} \times R_n)
\end{aligned} \tag{12}$$

Here, we first project H_h which is a relation keeping the hub values in the previous iteration. With H_h, we compute the current authority values, R_a, in this iteration. And we compute the current hub values, R_h, using R_a computed. Next, we combine the authority values and hub values computed into a single table R_{ha}. Finally, we update H by the newly computed authority and hub values after normalization. Note that R_n is a relation with a single tuple for the normalization purpose.

Below, we show how to support *TopoSort* (Topological Sorting) for *DAG* (Directed Acyclic Graph) using anti-join. To compute *TopoSort*, we assign a level L value to every node. For two nodes,

u and v, if $u.L < v.L$, then $u < v$ in the *TopoSort*; if $u.L = v.L$, then either $u < v$ or $v < u$ is fine, since the *TopoSort* is not unique. Let $Topo(ID, L)$ be a relation that contains a set of nodes having no incoming edges with initial L value 0. The initial $Topo$ can be generated by $\Pi_{ID,0}(V \bar{\ltimes}_{ID=E.T} E)$. In the recursive part, it is done by several steps.

$$\begin{aligned} L_n &\leftarrow \rho_L(\mathcal{G}_{max(L)+1} Topo) \\ V_1 &\leftarrow V \underset{V.ID=T.ID}{\bar{\ltimes}} Topo \\ E_1 &\leftarrow \Pi_{E.F,E.T}(V_1 \underset{ID=E.F}{\bowtie} E) \\ T_n &\leftarrow \Pi_{ID,L}(V_1 \underset{V_1.ID=E_1.T}{\bar{\ltimes}} E_1) \times L_n \\ Topo &\leftarrow Topo \cup T_n \end{aligned} \tag{13}$$

Here, first, we compute the L value to be used for the current iteration, which is the max L value used in the previous iteration plus one. It is stored in L_n. Next, we remove those nodes that have already been sorted by anti-join and obtain V_1. With $V_1 \subseteq V$, we obtain the edges among nodes in V_1 as E_1. T_n is the set of nodes that are sorted in the current iteration. Finally, we get the new $Topo$ by union of the previous $Topo$ and the newly sorted T_n. It repeats until T_n is empty.

Table 2 shows some representative graph algorithms that can be supported by the 4 operations including MM-join, MV-join, anti-join and union-by-update. As a summary, MV-join together with union-by-update can be used to implement *PageRank*, weakly *Connected-Component*, *HITS*, *Label-Propagation*, *Keyword-Search* and *K-core*, whereas MM-join together with union-by-update can be used to support *Floyd-Warshall*, *SimRank* and *Markov-Clustering*. The anti-join serves as a selection to filter nodes/edges which are unnecessary in the following iterations. It is important to note that anti-join is not only for efficiency but also for the correctness. Equipped with anti-join, *TopoSort* is easy to be implemented. The combination of MV-join and anti-join support *Maximal-Independent-Set* and *Maximal-Node-Matching*.

We discuss the efficiency of supporting graph algorithms. The efficiency is closely related to the behaviors of the graph algorithms in every iteration. In general, there are groups of graph algorithms. One is always-active and one is path-oriented (or graph traversal). First, for the always-active algorithms, a node, v, needs to compute its own value using the nodes in its neighbor, and all nodes in a graph need to do so iteratively. Such algorithms include *PageRank*, *HITS*, *Maximal-Independent-Set* and *Label-Propagation*, for example. Here, the time complexity in one iteration in such algorithms is in $O(m + n)$. In *RDBMS*s, a hash join together with aggregation can be used to support such algorithms in every iteration and possibly achieve the similar performance. Second, for the path-oriented algorithms, an algorithm focuses on a specific local subgraph in an iteration, and such a local subgraph changes in different iterations. Take reachability query as an example, which is to find whether a node is reachable by another and can be done by either *BFS* or *DFS*. It only needs $O(m + n)$ to finish the overall computing, but it needs to perform join iteratively in *RDBMS*s. *RDBMS*s cannot support such path-oriented algorithms well in general. To address it, there are some techniques. In [41], several *SQL* level optimizations are discussed in *Teradata*, among them one is early selection. In [59], some labeling/indexing approaches are discussed. In addition, new efficient join processing in-parallel/on-cloud and new storage techniques have been recently studied, which can further improve the efficiency. It is important to note that *RDBMS*s are capable of handling large datasets when they cannot be easily handled in main memory.

Graph Algorithm	Aggregation	linear	nonlinear
TC [17]	–	✓	✓
BFS [17]	max	✓	
Connected-Component [48]	min/max	✓	
Bellman-Ford [17]	min	✓	
Floyd-Warshall [17]	min		✓
PageRank [36]	sum	✓	
Random-Walk-with-Restart [36]	sum	✓	
SimRank [28]	sum	✓	
HITS [36]	sum		✓
TopoSort [31]	–		✓
Keyword-Search [13]	max	✓	
Label-Propagation [46]	count	✓	
Maximal-Independent-Set [40]	max/min		✓
Maximal-Node-Matching [43]	max/min		✓
Diameter-Estimation [32]	–	✓	
Markov-Clustering [58]	sum		✓
K-core [37]	count		✓
K-truss [45]	count		✓
Graph-Bisimulation [27]	–		✓

Table 2: Graph Algorithms

5. XY-STRATIFIED

As discussed in Section 3, *SQL*'99 supports stratified negation in recursion, which means it is impossible to support graph processing that needs the functions beyond stratified negation. Recall that the 4 operations are not monotone and are not stratified negation. To address this issue, we discuss relational algebra operations in the context of DATALOG. The rules for the relational algebra are shown below, where the rules for selection, projection, Cartesian product, union, θ-join are given in [57].

Selection $\sigma_P R$ where P is the predicate used in the selection σ over the relation R. The DATALOG rule is given in Eq. (14).

$$R'(X_1, ...X_n) \quad :- \quad R(X_1, ...X_n), P \tag{14}$$

Projection $\Pi_X(R)$ where X is a set of attributes projected from the relation $R(Y)$ for $X \subseteq Y$.

$$R'(X) \quad :- \quad R(Y) \tag{15}$$

Cartesian product $R \times S$ where $R(X)$ and $S(Y)$ are two relations over X and Y attributes such that $X \cap Y = \emptyset$.

$$R'(X, Y) \quad :- \quad R(X), S(Y) \tag{16}$$

Union $R \cup S$ where $R(X)$ and $S(X)$ are two relations over X attributes.

$$\begin{aligned} R'(X) &\quad :- \quad R(X) \\ R'(X) &\quad :- \quad S(X) \end{aligned} \tag{17}$$

θ-Join $R \bowtie_\theta S$ where θ is the join condition between two relations $R(X)$ and $S(Y)$. Here, for simplicity, we assume that X and Y are two sets of attributes such that $X \cap Y = \emptyset$.

$$R'(X, Y) \quad :- \quad R(X), S(Y), \theta \tag{18}$$

MV**-Join and** MM**-join** are $A \underset{T=ID}{\overset{\oplus(\odot)}{\bowtie}} C$ and $A \underset{A.T=B.F}{\overset{\oplus(\odot)}{\bowtie}} B$, respectively. The DATALOG rules for MV-Join and MM-join are given in Eq (19) and Eq (20).

$$R'(Y, W) :- A(X, Y, W_1), C(X, W_2), W = \oplus(W_1 \odot W_2) \tag{19}$$

$$R'(X, Y, W) :- A(X, Z, W_1), B(Z, Y, W_2), W = \oplus(W_1 \odot W_2) \tag{20}$$

Difference ($R - S$) **and Anti-join** ($R \bar{\ltimes} S$) are given in Eq. (21).

$$R'(X, Y) \quad \text{:-} \quad R(X, Y), \neg S(X, -) \tag{21}$$

Union by Update ($R \uplus S$). The rules are given in Eq. (22).

$$\begin{aligned} R'(X, W_1) \quad &\text{:-} \quad R(X, W_1), \neg S(X, -) \\ R'(X, W_2) \quad &\text{:-} \quad S(X, W_2) \end{aligned} \tag{22}$$

As union, selection, projection, Cartesian product and θ-joins are monotone, recursive queries using such operations are stratified. But, MM-join, MV-join, anti-join, and union-by-update are not monotonic. The approach we take is based on *XY*-stratification [62, 63, 11]. An *XY*-stratified program is a special class of locally stratified programs [24]. As proposed by Zaniolo et al. in [62], it is a syntactically decidable subclass for non-monotonic recursive programs to handle negation and aggregation, and it captures the expressive power of inflationary fixpoint semantics [9].

We discuss the locally stratified programs using Example 4.8 given in [24]. Consider a DATALOG program with two rules, namely, (r1) $p(a)$:- $\neg p(c)$ and (r2) $p(b)$:- $\neg p(c)$. This DATALOG program is not stratified since there is negation in a cycle in the corresponding dependency graph for this DATALOG program. Note that a predicate can be treated as a relation. The locally stratified program is a fine-grained version which considers stratification at the level of ground atoms rather than at the level of predicates. By treating the single predicate p over a, b, and c, as three different predicates, p_a, p_b, and p_c, the above DATALOG program becomes (r1') p_a :- $\neg p_c$ and (r2') p_b :- $\neg p_c$, which is stratified. By Theorem 10.8 in [63], *every locally stratified program has a stable model that is equal to the result of the iterated fixpoint computation*. However, the problem is that there is no simple way to decide whether a DATALOG program is locally stratified in general because the locally stratified is to deal with atoms instead of predicates, which cannot be easily checked at compile-time.

An *XY*-program is a locally stratified DATALOG program that can be checked at compile-time by syntax. The main idea behind *XY*-program for dealing with possible infinite atoms is to use temporal (or stage) arguments over a discrete temporal domain: $\{0, 1, 2, \cdots\}$, that is represented by an initial number 0 and a successor function $s(i) = i + 1$ for a number i in the temporal domain. Given the successor function $s(\cdot)$, any number i can be represented by repeating $s(0)$ i times in the form of $s(s(\cdots s(0)))$. Here, consider a predicate p with a temporal argument T, for example, $p(s(T))$:- $p(T)$, where for simplicity p only has one temporal argument. By combining the predicate name p with the temporal argument, *XY*-program can represent an infinite possible set of atoms, $p_0, p_1, p_2, \cdots$, and whether a DATALOG program is *XY*-stratified can be checked at compile-time by syntax. For easy reference, we give *XY*-program definition in Appendix.

There is a simple test to check whether an *XY*-program P is *XY*-stratified. This is done by transforming P to a bi-state version of P, denoted as P_{bis}, where all temporal arguments are removed. An *XY*-program P is *XY*-stratified if its bi-state version P_{bis} is stratified [63]. In addition, P is locally stratified if P is an *XY*-stratified program. We give the procedure on how to transform an *XY*-program P to its bi-state program P_{bis} following the discussion in [63]. For each rule r in P, it conducts the 3 steps. First, it removes all the recursive predicates in r that have the same temporal argument as the head of r with the distinguished prefix new_. Second, it removes all other occurrences of recursive predicates in r with the distinguished prefix old_. Third, it removes the temporal arguments from the recursive predicates. For an *XY*-stratified DATALOG program, it computes fixpoint for each stratum with temporal argument T before it computes fixpoint for each stratum with temporal argument $s(T)$. It repeats until the fixpoint of the DATALOG program is reached.

```
with R as
    select ··· from R_{1,j}, ··· (Q_1)
    union all
    ···
    union all
    select ··· from R_{i,j}, ··· (Q_i)
    union all
    ···
    union all
    select ··· from R_{n,j}, ··· (Q_n)
```

Figure 2: The general form of recursive with in *SQL*'99

In this paper, we extend the with clause in *SQL*'99, "**with** R **as** $\langle$ R initialization $\rangle$ $\langle$ recursive querying involving R $\rangle$", to support a class of recursive query that can be used to support many graph analytical tasks. To minimize such extension, we restrict that the with clause only has one recursive relation R, and there is only one cycle in the corresponding dependency graph. The extension is to allow negation as well as aggregation in a certain form for a recursive query Q. In the following discussion, we focus on "$\langle$ recursive querying involving R $\rangle$". We give a theorem to show that the recursive queries using the 4 operations discussed can have a fixpoint by which a unique answer can be obtained.

Theorem 5.1: *A recursive query Q with a single recursive relation, specified by the relational algebra operations, selection, projection, Cartesian product, union, θ-join,* MM*-join,* MV*-join, anti-join (difference), and union-by-update, is XY-stratified if there is only one cycle in the corresponding dependency graph.*

The proof sketch is given in the Appendix.

6. TO ENHANCE THE WITH CLAUSE

In this section, we present our approach to enhance with clause. The general form of the recursive with clause in *SQL*'99 is shown in Fig. 2. It consists of several select statements connected by union all. Here, the i-th select is indicated as the i-th *SQL* query Q_i in which it accesses several relations as $R_{i,j}$. It is important to mention that such $R_{i,j}$ may appear in a nested *SQL* query in Q_i. Among all such Q_i queries, for $1 \leq i \leq n$, there is a specific k such that all queries Q_i, for $i < k$, are for initialization and do not refer to the recursive relation R among $R_{i,j}$, and all queries Q_i, for $i \geq k$, are for recursive querying involving the recursive relation R and refer to R among $R_{i,j}$. Below, we call Q_i an initial subquery if it does not refer to R, and call Q_i a recursive subquery if it does refer to R. We focus on the recursive subqueries that refer to the recursive relation R. We enhance the *SQL*'99 with clause as follows. (1) To support negation/aggregation, we support anti-join, MV-join, and MM-join in a recursive query Q_i. (2) We support union-by-update in addition to union all. (3) For each query Q_i, we support computed by, which allows us to specify how a relation is computed by a sequence of queries. By allowing it, we can express complex queries in an easy way. (4) We support nonlinear recursion as well as mutual recursion. (5) Like some *RDBMS*s, we control the looping.

We explain why nonlinear/mutual recursion is allowed in this work. From the viewpoint of *SQL*, it is understood that linear recursion can support many applications, and it seems unnecessary to adopt nonlinear recursion, which is hard to implement in order to achieve high efficiency. From the viewpoint of supporting graph algorithms, as shown in Table 2, many graph algorithms need to be supported using nonlinear recursion. In other words, on one hand, it is to support certain queries efficiently in *RDBMS*, and on the

```
1.  with
2.  $P(ID, W)$ as (
3.    (select $R.ID$, $0.0$ from $R$)
4.    union by update $ID$
5.    (select $S.T$, $c * \mathbf{sum}(W * ew) + (1-c)/n$ from $P$, $S$
6.    where $P.ID = S.F$ group by $S.T$)
7.    maxrecursion 10)
8.  select $ID, W$ from $P$
```

Figure 3: The recursive with **for** *PageRank* **by ours**

other hand, it is to support as many as possible graph algorithms in *RDBMS*. In fact, the recursive queries have not been discussed since *SQL*'99. In this work, we take the position to support a large pool of graph algorithms and we will study how to support nonlinear recursion by exploring the possibility of utilizing the graph algorithms/implementation as an access method inside *RDBMS*s.

In the following, we highlight the main points but omit the syntax details.

MM-join and MV-join: An MM-join/MV-join is supported by an *SQL* query, which joins two relations followed by group by and aggregation. Take MM-join, $A \overset{\oplus(\odot)}{\underset{A.T=B.F}{\bowtie}} B$, as an example. This is supported by the following *SQL*.

> **select** $A.F$, $B.T$, $\oplus(\odot)$ **from** A, B
> **where** $A.T = B.F$ **group by** $A.F$, $B.T$

Anti-join: There are several ways to support anti-join using not in, not exists, and left outer join. We show how to support anti-join by left outer join. Suppose there are two relations $R(ID)$ and $S(ID)$, the anti-join, $R \bar{\ltimes} S$ can be supported by "select ID from R left outer join S on $R.ID = S.ID$ where $S.ID$ is null". In addition, several *RDBMS*s like *Oracle* uses its internal implementation of anti-join as a way to support not in. In other words, when an *SQL* looks like "select * from R where $R.A$ not in (select $\cdots$)", the *RDBMS* will use its internal anti-join to execute this query instead.

Union-by-update is to update the values held in the recursive relation R. There are restrictions on when union-by-update is used. There is only one recursive query Q_i in the with clause, because the semantics is undefined when multiple union-by-update and union are used together. Consider the case when two union-by-update are used. It is unclear how to update a value since the new value cannot be uniquely determined. Then, consider when union-by-update is used with union all. It is unclear what it means in doing so. We allow two ways to specify union-by-update: with/without attributes. With attributes given, it specifies that two tuples in the previous iteration and the current iteration are identical if the values of the attributes specified are identical. Without attributes, it is to replace the previous recursive relation R by the currently generated result as a whole.

Fig. 3 shows our enhanced with to support *PageRank* (Eq. (9)). The recursive relation is given as $P(ID, W)$. The initialization is line 3. Here, the initial W attribute values are zero. The MV-join is specified by line 5-6, and the union-by-update is specified in line 4, where it specifies the attribute ID to check whether two tuples are identical. Alternatively, the attribute ID (line 4) can be omitted. Without ID attribute, it is to replace the previous recursive relation P by the one newly generated. The recursive relation P, specified by the with clause, is generated by the *SQL* query given in line 8.

The implementation of union-by-update by *SQL* is given below. Suppose we need to update the values in a relation, $V(ID, vw)$, by the values in another relation, using union-by-update ($V \uplus V'$). One way is to use the merge statement introduced in recent *SQL* (https://en.wikipedia.org/wiki/Merge_(SQL)). We illustrate its syntax in brief.

> **merge** V **using** V' **on** ID
> **when matched then update set** $V.vw = V'.vw$
> **when not matched then**
> **insert** $(V.ID, V.vw)$ **values** $(V'.ID, V'.vw)$

Here, merge is to insert/update tuples in the relation V by the relation V'. The attribute ID appears after "on" is the condition to check if two tuples from V and V' are matched. The following specifies the actions when tuples are matched and not matched. We omit the details. Another implementation is to use outer join [30] as follows.

> **select** coalesce$(V.ID, V'.ID)$ **as** ID,
> coalesce$(V'.vw, V.vw)$ **as** vw
> **from** V **full outer join** V' **on** ID

The resulting relation by the full outer join is in the form of $(V.ID, V.vw, V'.ID, V'.vw)$. All V tuples and V' tuples appear in the result even if a tuple in one fails to match any tuple in the other. If a tuple $t \in V$ cannot match any $t' \in V'$, the $(V'.ID, V'.vw)$ value in t is null. If a tuple $t' \in V'$ cannot match any $t \in V$, the $(V.ID, V.vw)$ value in t' is null. Here, coalesce(x, y) is a function, supported by all the 3 *RDBMS*s, which takes x value if it is non-null, otherwise y value.

Computed by: Recall that in *SQL*, a with statement can be expressed using multiple "as" shown below.

> **with** R_1 **as select** $\cdots$ **from** $R_{1,j}, \cdots$
> $\cdots$
> R_i **as select** $\cdots$ **from** $R_{i,j}, \cdots$

In brief, it allows relations used in a select for R_i to refer to other relations used in another select for R_j that appears before R_i. In other words, such references must be forwarded, which means that it cannot lead to recursion. In addition, the relations defined using as cannot be unioned using union all, since it mixes "as" to define relations and "union all" to do query. However, in many cases, since graph processing is rather complicated, on one hand, it needs to use a sequence of *SQL* statements to query; on the other hand, it needs to use union all or union-by-update. In this work, we propose to separate "as" which is to define relations and "union all" which is to query by introducing a computed by clause as shown in Fig. 4. As shown Fig. 4, in the main body of the enhanced with, either union all or union-by-update is allowed to union subqueries, and no as is allowed. When union-by-update is used, it cannot be used more than once, and cannot be used with other union all together as discussed. By computed by, it specifies a collection of cycle-free relations using as for computing relations $R_{i,j}$ used in a single Q_i in the main body of with locally. We allow recursion on R by recursive query Q_i. The restriction is that the queries inside the computed by part of Q_i must be cycle free, as it can be proved that it is *XY*-stratified. This condition can be relaxed. But it is beyond the focus of this work. The with statement for *TopoSort* (Eq. (13)) is shown in Fig. 5.

Nonlinear/Mutual Recursion: We support nonlinear and mutual recursion in the enhanced with clause. Here, *Floyd-Warshall* is an example of nonlinear recursion (Eq. (8)), which needs to be implemented as "select * from E as E_1, E as E_2, $\cdots$". Some graph processing tasks need nonlinear recursion, and a nonlinear recursion is *XY*-stratified by Theorem 5.1 since renaming (ρ) does not violate *XY*-stratification. In addition, nonlinear queries are easy to understand and can converge faster, whereas it is difficult to implement efficiently. The efficiency issues can be addressed by exploring if some nonlinear recursion needed in its limited form can

```
with $R$ as
    select $\cdots$ from $R_{1,j}, \cdots$ computed by $\cdots$ ($Q_1$)
    union all
    $\cdots$
    union all
    select $\cdots$ from $R_{i,j}, \cdots$ computed by $\cdots$ ($Q_i$)
    union all
    $\cdots$
    union all
    select $\cdots$ from $R_{n,j}, \cdots$ computed by $\cdots$ ($Q_n$)
```

Figure 4: The general form of the enhanced recursive with

```
with
$Topo(ID, L)$ as (
    (select $ID$, $0$ from $V$
    where $ID$ not in select $E.T$ from $E$)
    union all
    (select $ID$, $L$ from $T_n$
    computed by
        $L_n(L)$ as select $max(L) + 1$ from $Topo$;
        $V_1$ as
            select $V.ID$ from $V$
            where $ID$ not in select $ID$ from $Topo$;
        $E_1$ as
            select $E.F$, $E.T$ from $V_1$, $E$
            where $V_1.ID = E.F$;
        $T_n$ as
            select $ID$, $L$ from $V_1$, $L_n$
            where $ID$ not in select $T$ from $E_1$;))
select from $Topo$;
```

Figure 5: The recursive with **for** *TopoSort*

be linearized [64], which we leave it as our future work. For mutual recursion, even it is allowed by *SQL*'99, none of the *RDBMS*s support it. We show the way of supporting *HITS* following the example given https://www.youtube.com/watch?v=0n9kScLFyIo.

```
with recursive
    $Hub(\cdots)$ as select $\cdots$ from $Author \cdots$
    $Author(\cdots)$ as select $\cdots$ from $Hub \cdots$
select $*$ from $Hub$;
```

Some observations can be made. First, to deal with mutual recursion for two relations to refer to each other, the with statement is not written as "with (recursive) R" for a specific single recursive relation R. Second, the two relations involved in mutual recursion, such as Hub and $Author$, are defined using as followed by select, and refer to each other mutually. Such references are not all forwarded, and cannot be supported by the *RDBMS*s. We take a different approach. (i) We use a single recursive relation in the with statement. (ii) To deal with mutual recursion, instead of letting two relations, say Hub and $Author$, to refer to each other mutually, we get a relation Hub' that contains tuples of Hub in the previous iteration, compute the current $Author$ using Hub', and then compute the current Hub using the current $Author$. The with statement for *HITS* (Eq. 12) is shown in Fig. 6.

Looping control: In general, it is difficult to decide whether a recursive query will terminate. The following simple recursive query

with $R(n)$ **as** ((**select values**(0)) **union all** (**select** $n + 1$ **from** R))

leads to an infinite recursion in *PostgreSQL*. We adapt maxrecursion which is used in *SQL-Server* [5] to allow users to limit the number of iterations by setting the query hint maxrecursion (between 0 to 32,767).

With vs Enhanced With: By *SQL*'99, negation including group by & aggregation is not allowed in the with clause. Therefore, none of the 4 operations can be allowed by *SQL*'99. Some *RDBMS*s (e.g., *PostgreSQL* and *Oracle*) support partition by. Like group by, partition by is used to divide tuples into groups, and compute aggregation for every group. Unlike group by, partition by does it

```
with
$H(ID, h, a)$ as (
  (select $ID$, $1.0$, $1.0$ from $V$)
  union by update
  (select $ID$, $h/sqrt(nh)$, $a/sqrt(na)$
  from $R_{ha}$, $R_n$
  computed by
      $H_h$ as select $ID$, $h$ from $H$;
      $R_a(ID, a)$ as select $ID$, $sum(h * ew)$ from $H_h$, $E$
                  where $H.ID = E.T$
                  group by $E.F$;
      $R_h(ID, a)$ as select $ID$, $sum(a * ew)$ from $R_a$, $E$
                  where $R_a.ID = E.F$
                  group by $E.T$;
      $R_{ha}$ as select $R_a.ID$, $h$, $a$ from $R_a$, $R_h$
              where $R_a.ID = R_h.ID$;
      $R_n(nh, na)$ as select $sum(h * h)$, $sum(a * a)$
                  from $R_{ha}$))
select from $H$;
```

Figure 6: The enhanced recursive with **for** *HITS*

for every tuple in a group, and does not result a single tuple for a group. By partition by & aggregation, it can support some graph algorithms (e.g., *Bellman-Ford*). But, it needs to keep the result of a tuple in every iteration using an additional attribute to indicate the value of a tuple in a specific iteration (e.g., $L + 1$ used to compute *PageRank* in Fig. 9), because it cannot delete tuples by the constraint on negation. The cost is that the number of tuples increases exponentially in iterations. *PostgreSQL* allows distinct along with partition by & aggregation and can compute *PageRank* (Fig. 9). With distinct, the number of tuples increases linearly in iterations. The cost of computing limited graph algorithms is too high. Note that distinct is a kind of negation, there is no reported study why it can be allowed, even though it only removes duplicates. By with+, we support the 4 new operations which cannot be used in with, where union-by-update is a powerful operation that can be used to update values directly in iterations. In addition, in with+, we support computed by, which makes it easy to support graph algorithms (e.g., *HITS*). Note that *DFS* can be supported in an indirect way by with/with+ as given in [7].

The implementation: We sketch how to support recursive queries using the enhanced with over the *RDBMS*s. We process a recursive query, Q, as given in Fig. 4. First, for each subquery Q_i used in Q including those defined by the computed by statement, we construct a local dependency graph G_i. The graph G_i constructed must be cycle free. We ensure that it is *XY*-stratified. Second, we create a *PSM* (Persistent Stored Model) in the recent *SQL* standard. With *PSM*, we create a unique procedure F_Q for the recursive query Q to be processed, as illustrated below.

```
create procedure $F_Q$ (
  declare $C_1, \cdots, C_i, \cdots$;
  create table $R_{i,j}$ for all tables defined by as in a subquery $Q_i$;
  create SQL statement to compute the initial $R$ by union of
  all initial subqueries;
  loop
    insert into $R_{i,j}$ select $\cdots$ for every $R_{i,j}$ used in $Q_i$;
    compute condition $C_i$ for each recursive subquery $Q_i$;
    if all $C_i$ for the recursive subqueries are false then exit
    compute the recursive relation $R$ for the current iteration;
    union the current $R$ with the previous $R$ computed;
  end loop)
```

In the procedure, F_Q, first we declare variables $C_1, \cdots, C_i, \cdots$ for every subquery Q_i, which are used to check the condition to exit from the looping. Second, we create all the tables needed for the relations defined by as in the computed by statements. Third, we include *SQL* statements to compute the initial recursive relation R.

Fourth, we create a looping. In the looping, in every iteration, we use an insert to compute $R_{i,j}$, and use an *SQL* to check whether a Q_i is empty. Note that different systems use different syntax to do it. For example, in *Oracle*, it can be written as "select count (*) C_i from $\cdots$" for Q_i, where C_i is an integer variable declared. If the resulting relation is empty, $C_i = 0$. By using such variables declared, we can determine when it exits from the looping. When the condition does not hold, the recursive relation computed in this iteration will be unioned with the one computed in the previous iteration by either union all or union-by-update. With the procedure defined, we can run the statements in the procedure F_Q by issuing "call F_Q". The algorithm is given in Algorithm 1 in the Appendix.

7. PERFORMANCE STUDIES

We report our performance studies on a PC with Intel Core i7-4770 (3.40 GHz) and 32GB RAM running Linux CentOS 6.7 64bit. We tested the enhanced with in 3 *RDBMS*s: *Oracle* (11gR2 Enterprise Edition) [6], *IBM DB2 10.5 Express-C* [4], and *PostgreSQL* 9.4 [7]. For *Oracle*, we enable the *Oracle* Auto Memory Management (AMM) by setting memory_target and memory_max_target to 24GB, following *Oracle* recommendation to use 2/3 main memory. For *PostgreSQL*, we enable the non-durable setting and adjust the following key parameters (shared_buffers = 8GB, temps_buffers = 2GB and work_mem = 512M). Here, non-durable enabled reduces the overhead of durability by allowing the risk of data loss. For *DB2*, self_tuning_mem is activated, and database_memory is set as automatic, which is similar to AMM supported by *Oracle*.

Graph Algorithms: The 10 graph algorithms implemented by the enhanced with statement include the single source shortest path (SSSP) by *Bellman-Ford* (Eq. (7)), weakly *Connected-Component* (WCC) (Eq. (6)), *PageRank* (PR) (Eq. (9)), *HITS* (HITS) (Eq. (12)), *TopoSort* (TS) (Eq. (13)), as well as K-*core* (KC) [37], *Maximal-Independent-Set* (MIS) [40], *Label-Propagation* (LP) [46], *Maximal-Node-Matching* (MNM) [43], *Keyword-Search* (KS) [13]. We briefly discuss how to process KC, MIS, MNM, LP, and KS. For KC, we let E' be the edge relation E initially. In every iteration, (1) we obtain V' with nodes whose degree is $> k$, (2) we compute E' if u can reach v via w for those nodes in V'. The result is obtained when E' cannot be changed. In testing, k is set to 10 for the dense graph Orkut and 5 for the others. MIS is the random-priority parallel algorithm [40]. In each iteration, the algorithm repeats three steps. (1) each node v generates a real number $r(v)$ in $[0, 1]$ randomly and sends it to its neighbors by join to E. (2) the node v with smallest $r(v)$ among their neighbors is picked up and removed into I. (3) the neighbors of I and associated edges are removed from the graph. Note that *RDBMS*s have a Rand function to generate a random number. We repeat 10 times to report the average time. For MNM, each node collects the weight of its neighbors first. Among these nodes, the one with maximum weight is chosen for this node. If two nodes choose each other, they form a pair of matching, and they will be excluded in the next iteration. This algorithm stops if no matching pairs are found. For LP, in every iteration, each node collects the label of its neighbors, finds the label which has the maximum count, and uses it as its new label. For KS, it finds the roots of Steiner trees, for a given keyword search query. Initially, each node generates an indicated vector (composed by 0 or 1) for itself. In every iteration, each node collects the indicated vector of its neighbors and updates its indicated vector by logic pair-wise OR. When the iteration reaches a given depth, the nodes whose indicated vector without 0 elements are reported as the roots. In our testing, we search for 3 labels with depth 4. For PR, HITS and LP, the number of iterations is fixed to 15.

Graphs	$\lvert V\rvert$	$\lvert E\rvert$	Diameter	Avg. Degree
Youtube (YT)	1,134,890	2,987,624	20	5.27
LiveJournal (LJ)	3,997,962	34,681,189	17	17.35
Orkut (OK)	3,072,441	117,185,083	9	76.22
Wiki Vote (WV)	7,115	103,689	7	29.14
Twitter (TT)	81,306	1,768,149	7	51.69
Web Google (WG)	875,713	5,105,039	21	11.66
Wiki Talk (WT)	2,394,385	5,021,410	9	4.19
Google+ (GP)	107,614	13,673,453	6	254.12
U.S. Patent Citation (PC)	3,774,768	16,518,948	22	8.75

Table 3: The Real Datasets

Time (ms)	*Oracle*	*DB2*	*PostgreSQL*
update from	–	–	123,466
merge	187,030	178,079	–
full outer join	55,870	80,473	113,060
drop/alter	54,760	79,415	102,233

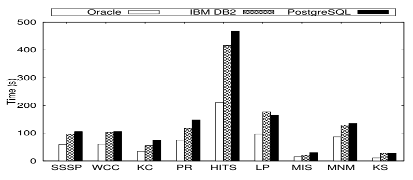
Table 4: union-by-update in Web Google

Datasets: We conducted testing using 9 real datasets, as shown in Table 3. The datasets are obtained from SNAP (https://snap.stanford.edu/data). In Table 3, the top 3 are undirected graphs and the remaining 6 are directed graphs. An undirected graph is maintained as a directed graph by including two directed edges for an undirected edge. We randomly generate labels for nodes which are needed for testing LP and KS. We also randomly generate a node-weight for every node in a graph in the range of [0, 20], when node-weight is needed for example in MNM.

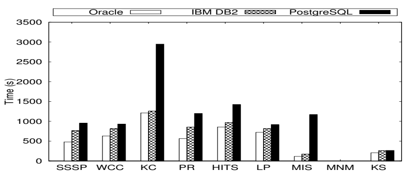
7.1 Exp-1: Union-By-Update and Anti-Join

Union-by-update: We test 4 union-by-update implementations. Two are to use merge and full outer join with function coalesce. As *PostgreSQL* does not support merge until version 9.5+, we use update from in testing *PostgreSQL* instead of merge, which updates the tuples of one table by referencing tables in the from lists. *Oracle* and *DB2* do not support it since *SQL* does not support it. In addition, a special way to support union-by-update is to replace the relation in the previous iteration with the relation computed in the current iteration. This can be done by drop a table and alter table in *SQL*. We test union-by-update by performing *PageRank* in 15 iterations on two graphs, Web Google and U.S. Patent Citation. Table 4 and Table 5 show the results. The full outer join outperforms merge, as it essentially does join instead of real update. The update from clause is only supported by *PostgreSQL*. Different from merge, the update from implementation does not check and report duplicates in the source table, which shows the similar performance as full outer join does. It is interesting to know that replacing the previous table by drop and alter shows the similar performance as full outer join. In the following, we use full outer join to support union-by-update.

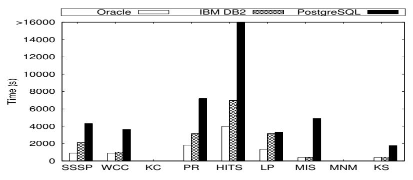
Anti-Join: We test anti-join ($\bar{\ltimes}$) using 3 implementations, not in, not exists and left outer join with "is null" condition. It is worth noting that although not in is more intuitive to write than the other two, their logics are not equivalent so that *RDBMS*s generate different query plans. Considering $R_O \bar{\ltimes} R_I$, where R_O and R_I indicate the outer table used in the top *SQL* and the inner table used in the nested *SQL*, respectively, since a nested *SQL* is needed. Here, not exists and left outer join will generate the same query plan. On the other hand, for not in, in *PostgreSQL* and *DB2*, its query plan is scanning R_O by filtering the tuples in R_I. It is an extension of the regular anti-join with building data structures for sorting/hashing the null values. *Oracle* optimizer hints can control the specific query plan of NAAJ and anti-join. We test the 3 implementations of anti-join by processing TS on two graphs, Web Google and U.S. Patent Citation. Table 6 and Table 7 show



(a) Youtube

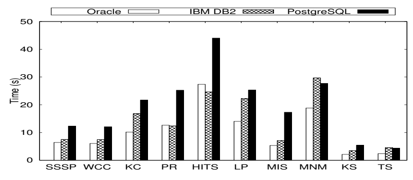


(b) LiveJournal

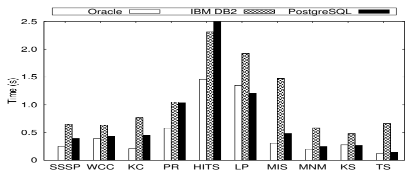


(c) Orkut

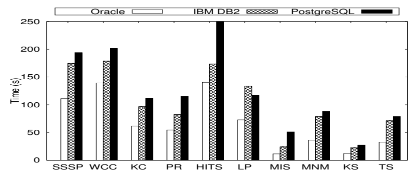
Figure 7: Testing 9 Graph Algorithms over 3 Undirected Graphs



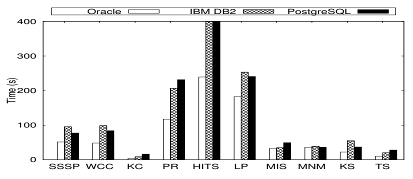
(a) Twitter



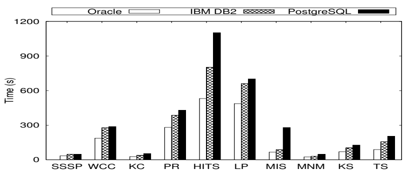
(b) Wiki Vote



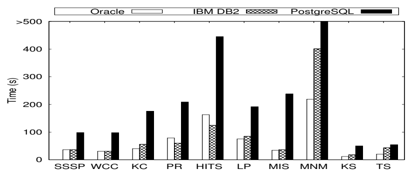
(c) Web Google



(d) Wiki Talk



(e) U.S. Patent Citation



(f) Google+

Figure 8: Testing 10 Graph Algorithms over 6 Directed Graphs

Time (ms)	*Oracle*	*DB2*	*PostgreSQL*
update from	–	–	643,965
merge	853,180	739,414	–
full outer join	281,060	386,573	430,066
drop/alter	298,300	366,032	429,696

Table 5: union-by-update in U.S. Patent Citation

Time (ms)	*Oracle*	*DB2*	*PostgreSQL*
not exists	31,190	76,747	74,594
left outer join	32,470	76,198	74,560
not in	32,050	76,297	78,792

Table 6: Antijoin in Web Google

the results. There are marginal differences among the 3 implementations. Both not exists and left outer join perform similarly and outperform not in. In the following testing, we use left outer join to support anti-join.

7.2 Exp-2: Graph Algorithms Testing

We conducted extensive testing using 10 graph algorithms over 9 large graphs. Fig. 7 shows the performance of the 9 algorithms (without *TopoSort*) over 3 undirected graphs, and Fig. 8 shows the performance of the 10 algorithms over 6 direct graphs. For *PostgreSQL*, the results showed include the time needed to construct indexes. We omitted the result for a graph algorithm if it fails to finish in 5 hours. Overall, *Oracle* performs best, while *DB2* outperforms *PostgreSQL*. This is because *PostgreSQL* does not generate the optimal plan for temporary tables due to the lack of sufficient statistical information.

The number of operations, such as join, aggregation, and union-by-update, in an iteration, plays an important role. For example, in an iteration PR executes 1 MV-join and 1 union-by-update, whereas HITS executes 2 MV-joins, 1 union-by-update, 1 θ-join, and an extra aggregation for normalization. HITS needs much time than PR, as observed in Fig. 7, and Fig. 8. Also, the number of iterations is an important factor in determining the performance. The number of iterations needed by MNM has a significant variance in testing, over different graphs. For U.S. Patent Citation (Fig. 8(e)) it ends after only one iteration. For Google+ (Fig. 8(f)), it needs 18 iterations which lead to a long running time. As a comparison, MIS requires the similar number of iterations over different graphs, and the average number 4-6. In addition, the system resources utilization cannot be neglected. It is known that graph analytics are CPU intensive work. Consider the graph algorithms over LiveJournal (Fig. 7(b)), the CPU utilization ratio is 70%-80%. However, the CPU utilization ratio for the same graph algorithms over Orkut (Fig.7(c)) is only 40%-50%. Low CPU utilization indicates plenty of system resources are spent on I/O, e.g., reading and writing data, accessing indexes and logs. Note that, even though *RDBMS*s can bypass the redo-log for temporary tables, it still needs to log. As can be observed, if the whole graphs can be held in main memory, it can achieve high efficiency.

Time (ms)	*Oracle*	*DB2*	*PostgreSQL*
not exists	87,030	152,814	178,693
left outer join	88,910	154,330	181,285
not in	88,870	156,524	204,712

Table 7: Antijoin in U.S. Patent Citation

8. CONCLUSION

To support a large pool of graph algorithms, we propose 4 operations, namely, MM-join, MV-join, anti-join and union-by-update, that can be supported by the basic relational algebra operations, with group-by & aggregation. Among the 4 operations, union-by-update plays an important role in allowing value updates in iterations. The 4 non-monotonic operations are not allowed to be used in a recursive query as specified by *SQL*'99. We show that the 4 operations proposed together with others have a fixpoint semantics based on its limited form, based on DATALOG techniques. In other words, a fixpoint exists for the 4 operations that deal with negation and aggregation. We enhance the recursive with clause in *SQL* and conduct extensive performance studies by translating the enhanced recursive with into *SQL/PSM* to process in *RDBMS*s. The testing is done for 10 graph algorithms using 9 real graphs on *Oracle*, *DB2*, and *PostgreSQL*. As future work, we will study the efficiency issues and explore parallel processing techniques.

ACKNOWLEDGES: This work was supported by the Research Grants Council of Hong Kong SAR, China, No. 14209314 and No. 14221716.

9. REFERENCES

[1] https://github.com/dato-code/PowerGraph.

[2] http://socialite-lang.github.io.

[3] http://giraph.apache.org.

[4] IBM DB2 10.5 for linux, unix and windows documentation. http://www.ibm.com/support/knowledgecenter/#!/SSEPGG_10.5.0/com.ibm.db2.luw.kc.doc/welcome.html.

[5] Microsoft WITH common_table_expression. https://msdn.microsoft.com/en-us/library/ms175972.aspx.

[6] Oracle database SQL language reference. http://docs.oracle.com/cd/E11882_01/server.112/e41084/toc.htm.

[7] Postgresql 9.4.7 documentation. http://www.postgresql.org/files/documentation/pdf/9.4/postgresql-9.4-A4.pdf.

[8] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *Proc. o SIGMOD'16*, 2016.

[9] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[10] G. Aranda-López, S. Nieva, F. Sáenz-Pérez, and J. Sánchez-Hernández. Formalizing a broader recursion coverage in SQL. In *Proc. of PADL'13*, 2013.

[11] F. Arni, K. Ong, S. Tsur, H. Wang, and C. Zaniolo. The deductive database system LDL++. *TPLP*, 3(1), 2003.

[12] P. Barceló. Querying graph databases. In *Proc. of PODS'13*, 2013.

[13] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Proc. of ICDE'02*, 2002.

[14] Y. Bu, V. R. Borkar, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Scaling datalog for machine learning on big data. *CoRR*, abs/1203.0160, 2012.

[15] W. Cabrera and C. Ordonez. Unified algorithm to solve several graph problems with relational queries. In *Proc of AMW'16*, 2016.

[16] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. Mad skills: new analysis practices for big data. *PVLDB*, 2(2), 2009.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, 3 edition, 2009.

[18] J. Fan, A. Gerald, S. Raj, and J. M. Patel. The case against specialized graph analytics engines. In *Proc. of CIDR'15*, 2015.

[19] S. J. Finkelstein, N. Mattos, I. Mumick, and H. Pirahesh. Expressing recursive queries in SQL. *ISO-IEC JTC1/SC21 WG3 DBL MCI*, (X3H2-96-075), 1996.

[20] J. Gao, R. Jin, J. Zhou, J. X. Yu, X. Jiang, and T. Wang. Relational approach for shortest path discovery over large graphs. *PVLDB*, 5(4), 2011.

[21] J. Gao, J. Zhou, C. Zhou, and J. X. Yu. Glog: A high level graph analysis system using mapreduce. In *Proc. of ICDE'14*, 2014.

[22] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems The Complete Book*. Prentice Hall, 2002.

[23] A. Ghazal, D. Seid, A. Crolotte, and M. Al-Kateb. Adaptive optimizations of recursive queries in teradata. In *Proc. of SIGMOD'12*, 2012.

[24] S. Greco and C. Molinaro. *Datalog and Logic Databases*. Morgan & Claypool Publishers, 2015.

[25] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proc. of SIGMOD'08*, 2008.

[26] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The madlib analytics library: or mad skills, the sql. *PVLDB*, 5(12), 2012.

[27] M. R. Henzinger, T. Henzinger, P. W. Kopke, et al. Computing simulations on finite and infinite graphs. In *Proc. of FOCS'95*, 1995.

[28] G. Jeh and J. Widom. Simrank: a measure of structural-context similarity. In *Proc. of SIGKDD'02*, 2002.

[29] A. Jindal and S. Madden. Graphiql: A graph intuitive query language for relational databases. In *Proc. of BigData'14*, 2014.

[30] A. Jindal, S. Madden, M. Castellanos, and M. Hsu. Graph analytics using the vertica relational database. *arXiv preprint arXiv:1412.5263*, 2014.

[31] A. B. Kahn. Topological sorting of large networks. *CACM*, 5(11), 1962.

[32] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Hadi: Mining radii of large graphs. *TKDD*, 5(2), 2011.

[33] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: mining peta-scale graphs. *Knowledge and information systems*, (2), 2011.

[34] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. SIAM, 2011.

[35] L. Libkin, W. Martens, and D. Vrgoc. Querying graphs with data. *J. ACM*, 63(2), 2016.

[36] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge University Press, 2008.

[37] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *JACM*, 30(3), 1983.

[38] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2), 2015.

[39] J. Melton and A. R. Simon. *SQL: 1999: understanding relational language components*. Morgan Kaufmann, 2001.

[40] Y. Métivier, J. M. Robson, N. Saheb-Djahromi, and A. Zemmari. An optimal bit complexity randomized distributed MIS algorithm. *Distributed Computing*, 23(5-6), 2011.

[41] C. Ordonez. Optimization of linear recursive queries in sql. *TKDE*, 22(2), 2010.

[42] C. Ordonez, W. Cabrera, and A. Gurram. Comparing columnar, row and array dbmss to process recursive queries on graphs. *Information Systems*, 63, 2017.

[43] R. Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *Proc. of STACS'99*, 1999.

[44] P. Przymus, A. Boniewicz, M. Burzańska, and K. Stencel. Recursive query facilities in relational databases: a survey. In *Proc. of DTA/BSBT'10*, 2010.

[45] L. Quick, P. Wilkinson, and D. Hardcastle. Using pregel-like large scale graph processing frameworks for social network analysis. In *Proc. of ASONAM'12*, 2012.

[46] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

[47] R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *J. Log. Program.*, 23(2), 1995.

[48] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. Das Sarma. Finding connected components in map-reduce in logarithmic rounds. In *Proc. of ICDE'13*, 2013.

[49] S. Salihoglu and J. Widom. Help: High-level primitives for large-scale graph processing. In *Proc. of Workshop on GRAph Data management Experiences and Systems*, 2014.

[50] J. Seo, S. Guo, and M. S. Lam. Socialite: Datalog extensions for efficient social network analysis. In *Proc. of ICDE'13*, 2013.

[51] J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *PVLDB*, 6(14), 2013.

[52] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo. Big data analytics with datalog queries on spark. In *Proc. of SIGMOD'16*, 2016.

[53] A. Shkapsky, M. Yang, and C. Zaniolo. Optimizing recursive queries with monotonic aggregates in DeALS. In *Proc. of ICDE'15*, 2015.

[54] A. Shkapsky, K. Zeng, and C. Zaniolo. Graph queries in a next-generation datalog system. *PVLDB*, 6(12), 2013.

[55] S. Srihari, S. Chandrashekar, and S. Parthasarathy. A framework for sql-based mining of large graphs on relational databases. In *Proc. of PAKDD'10*, 2010.

[56] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie. Sqlgraph: An efficient relational-based property graph store. In *Proc. of SIGMOD'15*, 2015.

[57] J. D. Ullman. *Principles of Database and Knowledge Base Systems (Vol I)*. Computer Science Press, 1988.

[58] S. M. van Dongen. Graph clustering by flow simulation. *PhD Thesis, University of Utrecht*, 2000.

[59] H. Wei, J. X. Yu, C. Lu, and R. Jin. Reachability querying: An independent permutation labeling approach. *PVLDB*, 7(12), 2014.

[60] P. T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1), 2012.

[61] K. Xirogiannopoulos, U. Khurana, and A. Deshpande. Graphgen: exploring interesting graphs in relational data. *PVLDB*, 8(12), 2015.

[62] C. Zaniolo, N. Arni, and K. Ong. Negation and aggregates in recursive rules: the LDL++ approach. In *Proc. of DOOD*, 1993.

[63] C. Zaniolo, S. Stefano, Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced database systems*. Morgan Kaufmann, 1997.

[64] W. Zhang, C. T. Yu, and D. Troy. Necessary and sufficient conditions to linearize double recursive programs in logic databases. *ACM Trans. Database Syst.*, 15(3), 1990.

[65] Y. Zhang, M. Kersten, and S. Manegold. Sciql: Array data processing inside an rdbms. In *Proc. of SIGMOD'13*, 2013.

Appendix

Algorithm 1 WITH+

```
Input: A recursive SQL query, Q, using the enhanced with (Fig. 4)
Output: SQL/PSM to execute
1: for each subquery $Q_i$ in the enhanced with do
2:     create a local dependency graph $G_i$ for the relations used in $Q_i$,
       where the computed by must be cycle free;
3: end for
4: create a SQL/PSM procedure to process $Q$;
5: call the SQL/PSM function;
```

F	T	ew
1	1	a_{11}
1	2	a_{12}
2	1	a_{21}
2	2	a_{22}

(a) Relation A

F	T	ew
1	1	b_{11}
1	2	b_{12}
2	1	b_{21}
2	2	b_{22}

(b) Relation B

ID	vw
1	c_1
2	c_2

(c) Relation C

Table 8: The relation representations

```
1. with
2. P (ID, W, L) as (
3.    (select V.ID, 0.0, 0 from V)
4.    union all
5.    (select distinct E.T,
6.       c * (sum(P.W * ew) over(partition by E.T))
7.       +(1 − c)/n, P.L + 1
8.    from P, E where P.ID = E.F and P.L < 10))
9. select ID, W from P where L = 10
```

Figure 9: The recursive with for *PageRank* by *PostgreSQL*

Monotonicity: Consider any operation as a function $f(\cdot)$ which generates a relation over relations $R_1, R_2, \cdots, R_n$ such as $f(R_1, R_2, \cdots, R_n)$. Also, consider two assignments of $f(\cdot)$, namely, $f(\mathcal{R}^{(1)})$ and $f(\mathcal{R}^{(2)})$, for $\mathcal{R}^{(1)} = R_1^{(1)}, R_2^{(1)}, \cdots, R_n^{(1)}$ and and $\mathcal{R}^{(2)} = R_1^{(2)}, R_2^{(2)}, \cdots, R_n^{(2)}$. The function $f(\cdot)$ is monotone if $f(\mathcal{R}^{(1)}) \subseteq f(\mathcal{R}^{(2)})$ for any $\mathcal{R}^{(1)} \subseteq \mathcal{R}^{(2)}$. Here $\mathcal{R}^{(1)} \subseteq \mathcal{R}^{(2)}$ means $R_i^{(1)} \subseteq R_i^{(2)}$ for $1 \leq i \leq n$.

Definition 9.1: (**Dependency Graph**) A dependency graph is an edge-labeled directed graph $G = (V, E)$ for a recursive query Q specified by the recursive with clause. Here, V is a set of nodes, $V = \{v_1, v_2, \cdots\}$, representing relations in Q as follows. (a) A node represents the recursive relation called the recursive-node. (b) Every select clause has a corresponding node called a select-node. (c) Every base relation or its alias appearing in the from clause has a corresponding node, called a base-node. E is a set of edges. (i) There is an edge from every top select-node to the recursive-node. Note that a select-node may represent a nested *SQL* in the with clause. (ii) There is an edge from a base-node to a select-node if the base-node appears in the from clause of the select-node. (iii) There is an edge from a select-node, v_j, to a select-node, v_i, where v_j represents an immediate nested *SQL* appears in the *SQL* represented by v_i. Regarding edge label, by default the edge label is "$+$" which represents monotone. The edge label is "$-$" (negation) for an edge from v_j to v_i where v_j is a negated node. Here, a negated node is a node before which a negation condition (except, intersect, not exists, not in, $<>$ all, $<>$ some appears.

Definition 9.2: (**Stratification**) A recursive query Q is stratifiable if there is no edge with "$-$" label appearing in a cycle in the dependency graph G for Q. By applying a topological sorting over G for the stratifiable query Q, the nodes in G can be partitioned into m strata from S_1 to S_m, for $S_k < S_l$ if $k < l$. Q is stratified if every edge (v_j, v_i) directed to node v_i in G satisfies the following 2 conditions. Suppose v_i is in S_k, and v_j is in S_l. (1) $S_k \geq S_l$ if v_j is a non-negated node. (2) $S_k > S_l$, if v_j is a negated node.

Definition 9.3: (**XY-programs**) (Definition 10.13 in [63]) Let P be a DATALOG program with a set of rules defining mutually recursive predicates. Then P is an *XY*-program if it satisfies the following 2 conditions.

- (**X-rule**) Every recursive predicate of P has a distinguished temporal argument.

- (**Y-rule**) Every recursive rule r is either an *X*-rule or a *Y*-rule. First, a rule r is an *X*-rule when the temporal argument in every recursive predicate in r is the same variable (e.g. T). Second, a rule r is a *Y*-rule when (i) the head of the rule r has as temporal argument $s(T)$ where T denotes any variable, (ii) some subgoals of r have temporal argument T, and (iii) the remaining recursive goals have either T or $s(T)$ as their temporal arguments.

The proof sketch of Theorem 5.1: Since selection, projection, Cartesian product, and union are monotone, they are stratified, and therefore are *XY*-stratified. We focus on MM-join, MV-join, anti-join (difference), and union-by-update that are not stratified because they deal with either negation or aggregation over group-by. We prove it in two steps. In the first step, we prove that a recursive query with only one MM-join, MV-join, anti-join (difference), and union-by-update is *XY*-stratified. In the second step, we prove that any combination of the relational algebra operations is *XY*-stratified. Below R_q indicates a recursive relation (predicate), and T is a temporal argument.

In the first step, first, consider MV-join, the DATALOG rule is given in Eq. (19). The recursive query Q_1 by DATALOG is given below.

$$R_q(Y, W) \text{ :- } R_q(X, W_1), S(X, Y, W_2), W = \oplus(W_1 \odot W_2)$$

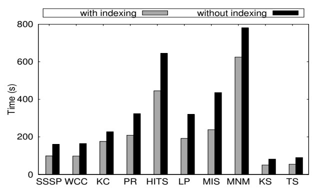
Q_1 is not stratified, since $\oplus$ and $\odot$ do not preserve monotone. Following *XY*-program, we add T to the R_q, and the corresponding *XY*-program is given below with one *Y*-rule.

$$R_q(Y, W, s(T)) \text{ :- } S(X, Y, W_2, T), R_q(X, W_1), W = \oplus(W_1 \odot W_2)$$

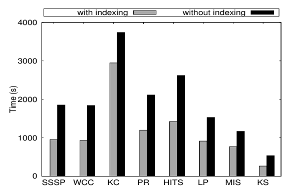
This *XY*-program is *XY*-stratified since its bi-state program is stratified. Second, we consider two possible recursive queries for MM-join. One is linear and the other is non-linear.

$$R_q(X, Y, W) \text{ :- } R_q(X, Z, W_1), S(Z, Y, W_2), W = \oplus(W_1 \odot W_2)$$
$$R_q(X, Y, W) \text{ :- } R_q(X, Z, W_1), R_q(Z, Y, W_2), W = \oplus(W_1 \odot W_2)$$

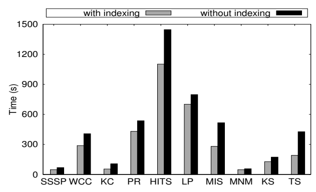
The *XY*-programs for the two DATALOG programs are shown be-



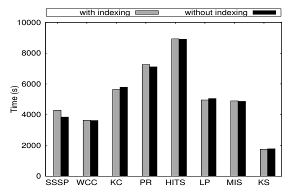
(a) Google+



(b) Live Journal



(c) U.S. Patent Citation



(d) Orkut

Figure 10: The Effectiveness of Indexing

low, which are XY-stratified since their bi-state programs are stratified.

$$R_q(X,Y,W,s(T)) \text{ :- } R_q(X,Z,W_1,T), S(Z,Y,W_2), W = \oplus(W_1 \odot W_2)$$
$$R_q(X,Y,W,s(T)) \text{ :- } R_q(X,Z,W_1,T), R_q(Z,Y,W_2,T),$$
$$W = \oplus(W_1 \odot W_2)$$

Third, consider the difference (anti-join). For the DATALOG program, $R_q(X,Y)$:- $R_q(X,Y), \neg S(X,-)$, since the negation is not a recursive predicate, this DATALOG program is stratified and therefore is XY-stratified. On the other hand, the DATALOG program, $R_q(X,Y)$:- $R(X,Y), \neg R_q(X,-)$, is not stratified since the negation is the recursive predicate. However, its XY-program, $R_q(X,Y, s(T))$:- $R(X,Y)$, $\neg R_q(X,-,T)$, is XY-stratified. Fourth, consider union-by-update, like the difference (anti-join), there are two recursive queries. One is stratified and therefore XY-stratified. We show the one that is not stratified below.

$$\begin{aligned} R_q(X,W_1) &\text{ :- } R(X,W_1), \neg R_q(X,-) \\ R_q(X,W_2) &\text{ :- } R_q(X,W_2) \end{aligned}$$

Its XY-program is given below with 2 Y-rules, which is XY-stratified.

$$\begin{aligned} R_q(X,W_1,s(T)) &\text{ :- } R(X,W_1), \neg R_q(X,-,T) \\ R_q(X,W_2,s(T)) &\text{ :- } R_q(X,W_2,T) \end{aligned}$$

In the second step, for the recursive query Q with only one recursive relation and one cycle in the corresponding dependency graph. We can obtain a DATALOG program as follows.

$$\begin{aligned} R_1(\cdots) &\text{ :- } R_q(\cdots), \cdots, B(\cdots) \\ R_2(\cdots) &\text{ :- } R_1(\cdots), \cdots, B(\cdots);\ \text{for } j = 1 \\ R_i(\cdots) &\text{ :- } R_j(\cdots), \cdots, B(\cdots);\ \text{for } i > j \\ &\quad \cdots \\ R_q(\cdots) &\text{ :- } R_j(\cdots), \cdots, B(\cdots);\ \text{for any } 1 \leq j < n \end{aligned}$$

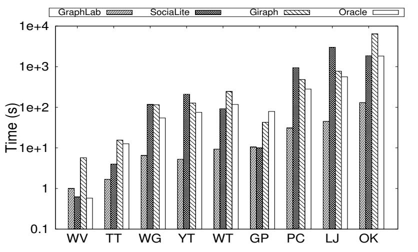
Here, R_q is the recursive relation (predicate). For simplicity, we assume that R_q has already been initialized before the recursive DATALOG program. R_i and R_j are 2 temporal relations generated by processing the recursive query or by the DATALOG program, where $R_i \neq R_q$ and $R_j \neq R_q$. Rules are used to support the relational algebra discussed above. To ensure there is only one cycle of R_q, a rule with R_i as the head has subgoals R_j for $i > j$. In other words, a temporal relation R_i is generated by some temporal relation R_j generated in an iteration in the recursive processing. B denotes any relations that are stored in the database to process this recursive query, for simplicity. Such DATALOG program can be rewritten as an XY-program as follows.

$$\begin{aligned} R_1(\cdots, s(T)) &\text{ :- } R_q(\cdots, T), \cdots, B(\cdots) \\ R_2(\cdots, s(T)) &\text{ :- } R_1(\cdots, s(T)), \cdots, B(\cdots);\ \text{for } j = 1 \\ R_i(\cdots, s(T)) &\text{ :- } R_j(\cdots, s(T)), \cdots, B(\cdots);\ \text{for } i > j \\ &\quad \cdots \\ R_q(\cdots, s(T)) &\text{ :- } R_j(\cdots, s(T), \cdots, B(\cdots);\ \text{for any } 1 \leq j < n \end{aligned}$$

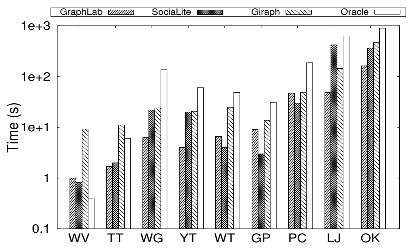
Such an XY-program is XY-stratified since its bi-state program is stratified. □

Some implementation details: The overall framework to process recursive *SQL* queries is shown in Algorithm 1. We give some implementation details. In computed by, when there is a need to create a relation, we create a temporary table, because (1) it preserves the concurrency and consistency at the session level, (2) the redo logs are bypassed for insert operations, and (3) the independent tablespace and buffer pool are used for improving performance. In a loop, we use insert to compute $R_{i,j}$. In *Oracle*, the optimizer hints provide a mechanism to instruct the optimizer to generate a certain query plan. For example, we use the "/*+APPEND*/" hint to perform a direct-path insert for the insert operation, and disable the optimizer feedback for reducing the overhead by "/*+OPT_PARAM*/" hint. We also check whether or not the table defined in computed by is empty. If one R_{ij} is empty and it is not a negated node in the dependency graph, subsequent query of Q_i will end. To check whether a Q_i is empty, in *PostgreSQL*, C_i is a variable of record to receive the first tuple by the limit clause since a temporary table lacks statistical information. At the end of each recursion, the intermediate result of Q_i is cleaned up by the truncate table clause. It is a *DDL* clause introduced in *SQL* to remove all the tuples in a relation swiftly.

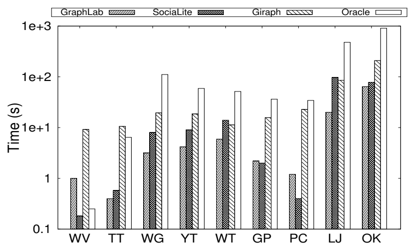
Exp-A: The Effectiveness of Indexing: To process a recursive query using the enhanced with statement, we use *PSM* to generate a sequence of *SQL* statements to process in which temporary tables are used. We test the effectiveness of indexing for MV-join or MM-join, over such temporary tables. In *Oracle* and *DB2*, the query plan produced by the optimizer is hash join and hash aggregation. In other words, the optimizers do not choose a new query plan for temporary tables, even when an index (either B+-tree or hash) is constructed in *PSM* for the temporary tables. In other words, the query plans are the same with/without an index constructed for temporary tables. In *PostgreSQL*, the optimizer generates a sub-optimal query plan using merge join and hash aggregation. The reason is that the optimizer does not have sufficient statistics of join attributes, in particular for temporary tables. Given merge join is used, the optimizer uses the indexing constructed on the joined attribute, and does the index scanning instead of the sequential table scanning. We test *PostgreSQL* since there is no difference between *Oracle* and *DB2*. Fig. 10 shows the performance with and without indexing for 4 larger datasets using *PostgreSQL*. Fig. 10(a), Fig. 10(b), Fig. 10(c) show that indexing improves 10%-50% performance. It is worth noting that in Fig 10(d), for dataset Orkut, the performance with indexing is similar or even slightly worse than that without indexing. This is because index scanning is random disk access. For a very large dataset, frequent index scanning makes I/O be the performance bottleneck. For example, the algorithms tested on the dataset Orkut only have CPU utilization 40%-50%. Overall, the indexing helps to improve efficiency in *PostgreSQL*. In the testing, we build indexes over temporary tables for *PostgreSQL*.



(a) PR

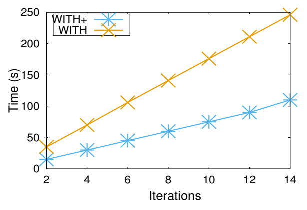


(b) WCC

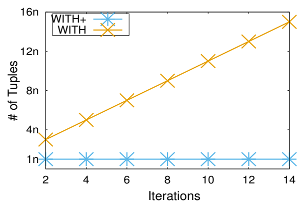


(c) SSSP

Figure 11: Comparison with *PowerGraph*, *SociaLite*, and *Giraph*

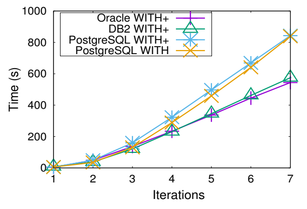


(a) Running Time

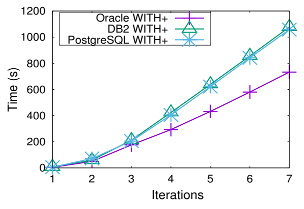


(b) # of Tuples

Figure 12: PR **on *PostgreSQL***



(a) TC



(b) APSP

Figure 13: Linear TC **and** APSP

Exp-B: RDBMS vs Other Systems: We compare our with+ at *SQL* level with 3 representative graph systems, *PowerGraph* [1], *SociaLite* [2] and *Giraph* [3]. *PowerGraph* is a parallel graph system (*C/C++* version of *GraphLab*), which adopts vertex-centric, and follows the GAS model (*Gather*, *Apply*, *Scatter*). *SociaLite* is a DATALOG based graph analysis framework. *Giraph* is a graph processing system based on the *BSP* model. We compare our approach in *Oracle*, with *PowerGraph*, *SociaLite* and *Giraph* to test 3 graph algorithms, PR, WCC and SSSP, over 9 real datasets (Table 3). The 3 algorithms are provided in *PowerGraph*, *SociaLite* and *Giraph*. Note that PR is always-acive whereas SSSP and WCC are path-oriented (or graph traversal). The results are shown in Fig. 11. For PR, as shown in Fig. 11(a), *PowerGraph* is the best. Our approach in *Oracle* can outperform *PowerGraph* if the dataset is small, and outperform *SociaLite* and *Giraph* in several datasets. For WCC (Fig. 11(b)), our approach in *Oracle* performs the best when the dataset is small (WV), and can outperform *Giraph* for TT. However, since our approach needs to do it by joins iteratively, there is a gap between the performance when datasets become large, which is similar to SSSP (Fig. 11(c)). It is important to note *Oracle* is a disk-based system. The performance of our approach can be enhanced using in-memory *RDBMS* techniques, optimizations among joins in a recursive query where computing cost can be reduced, and supporting graph algorithms as access methods inside *RDBMS*.

Exp-C: More on With and Enhanced With: We conduct 3 more testing of *PageRank* (PR), transitive closure (TC) and all pairs shortest paths (APSP). It is worth noting that the with of TC cannot stop if there are cycles in the dataset. To avoid infinite recursion, a threshold of recursive depth d needs to be specified in the where clause [41].

For PR, we compare PR based on our with+ (Fig. 3) over PR based on the with clause on *PostgreSQL* (Fig. 9). The dataset used is Web Google in Table 3, and the threshold of recursive depth d is set to 14, (in Fig. 9, L is used as d). Note that *DB2* cannot support PR, because *DB2* does not support partition by to compute aggregation in with. *Oracle* cannot support PR because *Oracle* does not support distinct to eliminate duplicates to get a correct answer in with, even though it supports partition by. Among the 3 *RDBMS*s, *PostgreSQL* can support it because it supports both partition by and distinct in with. We run tests in *PostgreSQL*. Fig. 12(a) and Fig. 12(b) show the running time and number of tuples accumulated in iterations, respectively. Here, the number of tuples is shown as xn, where x is a number and n is the number of nodes in the graph, which is 875,713 for Web Google. Over iterations, the with+ version of PR significantly outperforms the with version of PR. For running time, as shown in Fig. 12(a), the running time using with+ is 2 times faster than that using with. The speedup is mainly because the former uses group by, whereas the latter uses partition by and distinct. For the number of tuples accumulated in iterations, as shown in Fig. 12(b), the number of tuples using with+ keeps n, whereas the number of tuples using with increases linear. At the end of the 14-th iteration, the number of tuples accumulated using with is 15 times larger. The reason is that the with+ version uses union-by-update whereas the with version uses union all.

TC can be supported by both with and with+ using linear recursion. We test the implementation taken in with+ and that used behind with (e.g., Seminaive) without aggregation computing. The dataset used is Wiki Vote in Table 3, and the threshold of recursive depth d is set to 7. In Fig. 13, we only show *PostgreSQL* using with. We explain it below. *DB2* and *Oracle* can only use union all in the with clause, and cannot eliminate duplicates over iterations. As a result, they take too long to compute TC. *PostgreSQL* allows union instead of union all and can remove duplicates. As shown in Fig. 13(a), our with+ implementation performs in a similar way like the with implementation in *PostgreSQL*.

APSP can be done by linear recursion using MM-join based on *Bellman-Ford* for all nodes. The dataset used is Wiki Vote in Table 3, and the threshold of recursive depth d is set to 7. For APSP, we test MM-join in linear recursion. A higher cost occurs due to the extra aggregation operation in MM-join. Similar to TC, the cost of each iteration increases, because the matrix is no longer sparse over iterations by edge-to-edge join operations.