# Graph Processing in RDBMSs

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. A main issue to be addressed is how RDBMS can support graph processing. And the first thing is 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, 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 we enhance the recursive* with *clause in SQL'99. RDBMSs are capable of dealing with graph processing in reasonable time.*

## 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) [20], *Connected-Component* [57], shortest distance [20], topological sorting [34], *PageRank* [42], *Random-Walk-with-Restart* [42], *SimRank* [31], *HITS* [42], *Label-Propagation* [55], *Maximal-Independent-Set* [46], and *Maximal-Node-Matching* [52], 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. A recent survey can be found in [44]. Such distributed graph processing systems


**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**

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 [69, 15, 24, 28, 38, 28]. In addition, DATALOG has been revisited to support graph analytics [63, 62, 61, 59, 60, 17].

In this work, we investigate how *RDBMS* can support graph processing at *SQL* level for the following reasons. 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.

We revisit recursive *SQL* queries [22] and show that a large class of graph algorithms can be supported by *SQL* in *RDBMS*s in [74]. 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 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 conclude our work in Section 6.

## 2 Related Works

**Graph Processing and Management.** Many graph systems have been extensively studied recent years. Distributed graph processing systems like *Pregel* [41], *Giraph* [1], *GraphLab* [2] and *GraphX* [3] adopt 'think like a vertex' programming model. Program logic is expressed by vertex-centric push/pull based message passing, which is scalable for very large graph data. Apart from using some common built-in algorithms, users need to implement algorithms using the system APIs provided. The optimization techniques are system-dependent and at individual algorithm level. Such systems compute graphs imported and do not support graph maintenance. Different from graph processing systems, graph management systems e.g. *Neo4j* [4], *AgensGraph* [5], *Titan* [6] are designed for the transactional graph-like query. *Neo4j* adopts the property graph model and supports a visual UI for querying by the declarative language *Cypher*. *AgensGraph* is a multi-model graph database based on *PostgreSQL RDBMS*. *Titan* uses the functional language *Gremlin* to query and update graph.

**Graph Query Languages**. Graph query languages have been studied. A survey on query languages for graph databases can be found in [69], 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 [15]. Libkin et al. in [38] study how to combine data and topology by extending regular expressions to specify paths with data. The declarative, *SQL*-inspired query language *Cypher* [4], and functional language *Gremlin* [6] are integrated into transactional graph databases to describe graph patterns and traversal queries. There are several new attempts to query graphs. The *PGQL* [68] is a property graph query language in the *Oracle* Parallel Graph AnalytiX (PGX) toolkit, and is an *SQL*-like query language for graph matching. *PGQL*

supports path queries and has the graph intrinsic type for graph construction and query composition. Moffitt and Stoyanovich in [47] present a declarative query language *Portal* for querying evolving graphs, which is based on temporal relational algebra and is implemented on *GraphX*. Gao et al. in [24] 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 [32] by exploring a way to combine the features of *Pregel* (vertex-centric programming) and *SQL*. He and Singh in [28] propose the language *GraphQL* on graph algebra which deals with graphs with attributes as a basic unit, where the operations in the graph algebra include selection, Cartesian product, and composition. Salihoglu and Widom in [58] propose *HeLP*, a set of basic operations needed in many graph processing systems.

**Recursive SQL Queries**: *SQL*'99 supports recursive queries [45, 22]. 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 [18] and Kang et al. in [36] discuss an operation to multiply a matrix with a vector using joins and group-by & aggregation. Cabrera and Ordonez in [18] 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 [12]. Ordonez et al. in [50] compare *SQL* recursive query processing in columnar, row and array databases. 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 [26]. Aranda et al. in [13] study broadening recursion in *SQL*. The *SQL* level optimizations for computing transitive closures are discussed in [49], with its focus on monotonic aggregation for transitive closures. All the works in [50, 26, 13, 49] do not deal with negation and aggregation. It is important to note that aggregation and negation in general are needed for a large pool of graph algorithms, but aggregations and negations cannot be used within a recursive *SQL* query for ensuring that an *SQL* query can get a unique solution.

**Graph Analytics by SQL**. Graph analytics in *RDBMS*s using *SQL* have been studied. Srihari et al. in [64] introduce an approach for mining dense subgraphs in a *RDBMS*. Gao et al. in [23] leverage the window functions and the merge statement in *SQL* to implement shortest path discovery in *RDBMS*. Zhang et al. in [73] provide an *SQL*-based declarative query language *SciQL* to perform array computation in *RDBMS*s. Fan et al. in [21] propose *GRAIL*, a syntactic layer converting graph queries into *SQL* script. *GraphGene* [70] 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 [19, 29]. Passing et al. in [51] propose a *PageRank* operator in main-memory *RDB*. This operator is implemented by extending the *SQL* recursive query by lambda expression. In [33], *Vertica* relational database is studied as the platform for vertex-centric graph analysis. In [65], 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. *GQ-Fast* [39] is an indexed and fragmented database which supports efficient *SQL* relationship queries for typed graph analytics. Ma et. al in [40] present *G-SQL*, an *SQL* dialect for graph exploration. Multi-way join operations are boosted by underlying graph processing engine. Aberger et al. in [11] develop a graph pattern engine, called *EmptyHead*, to process graph patterns as join processing in parallel.

**Datalog-based Systems**. DATALOG is a declarative query language used in early deductive databases. As its new applications derive from information extraction, data analytics and graph analytics, many DATALOG-based systems have been developed recently. A survey of early deductive database systems can be found in [56]. *LDL++* is a deductive database system in which negation and aggregation handling in recursive rules are addressed [71, 14]. Based on *LDL++*, a new deductive application language system *DeALS* is developed to support graph queries [63], and the optimization of monotonic aggregations is further studied [62]. *SociaLite* [59] allows users to write high-level graph queries based on DATALOG that can be executed in parallel and

distributed environments [60]. These systems support graph analytics, especially iterative graph analytics e.g. transitive closure, weakly connect components, single source shortest distance since DATALOG has a great expressive power for recursion. DATALOG for machine learning is studied with *Pregel* and map-reduce-update style programming [17]. The efficient evaluation of DATALOG is investigated on data flow processing system *Spark* [61] and *BSP*-style graph processing engines [48]. 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 [45, 22]. The recursive queries are expressed using with clause in *SQL*. It 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 below, initially, the recursive relation $TC$ is defined to project the two attributes, $F$ and $T$, from the relation $E$. 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 $E$.

**with** $TC$ $(F, T)$ **as** ( **select** $F$, $T$ **from** $E$ **union all select** $TC.F$, $E.T$ **from** $TC$, $E$ **where** $TC.T = E.F$)

*SQL*'99 restricts recursion to be a linear recursion and allows mutual recursion in a limited form [45]. 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 [66]. As given in Theorem 3.3 in [66], union, select, projection, Cartesian product, natural joins, and $\theta$-joins are monotone. On the other hand, negation is not monotone [25]. 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. *SQL*'99 does not prohibit negation in recursive queries completely since the monotonicity of recursive query 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 [72].

**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 [53] in which Przymus et al. survey recursive queries handling in *RDBMS*s. We focus on *Microsoft SQL Server* (2016) [8], *Oracle* (11gR2) [9], *IBM DB2 10.5 Express-C* [7], and *PostgreSQL* (9.4) [10], where *Oracle* is not covered in [53]. We investigate 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.

**Handing Recursion by PSM in RDBMS**: There is another way to implement recursion, which is *SQL*/*PSM* (Persistent Stored Modules) included in *SQL* standard [66]. 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.

| Features | | *PostgreSQL* | *DB2* | *Oracle* | *SQL Server* |
|---|---|---|---|---|---|
| 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

# 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. A graph can be represented in matrix form. The nodes with node-weights can be represented as a vector of $n$ elements, denoted as $\mathsf{V}$. The edges with edge-weights can be represented as a $n \times n$ matrix, denoted as $\mathsf{M}$, where its $\mathsf{M}_{ij}$ 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 $\mathsf{V}$ and matrix $\mathsf{M}$, 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 ($\sigma$), projection ($\Pi$), union ($\cup$), set difference ($-$), Cartesian product ($\times$), and rename ($\rho$)), 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. 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 [37, 18]. The semiring is a set of $\mathcal{M}$ including two identity elements, $\mathbf{0}$ and $\mathbf{1}$, with two operations: addition ($+$) and multiplication ($\cdot$). In brief, (1) $(\mathcal{M}, +)$ is a commutative monoid with $\mathbf{0}$, (2) $(\mathcal{M}, \cdot)$ is a monoid with $\mathbf{1}$, (3) the multiplication ($\cdot$) is left/right distributes over the addition ($+$), and (4) the multiplication by $\mathbf{0}$ annihilates $\mathcal{M}$. Below, $\mathsf{A}$ and $\mathsf{B}$ are two $2 \times 2$ matrix, and $\mathsf{C}$ 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 $\mathsf{A}$ and $\mathsf{B}$ be two $n \times n$ matrices, and $\mathsf{C}$ 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, $\mathsf{M}_{ij}$ is the value at the $i$-th row and $j$-th column in the matrix $\mathsf{M}$, and $\mathsf{V}_i$ is the element at the $i$-th row in the vector $\mathsf{V}$. Let $A(F, T, ew)$ and $B(F, T, ew)$ be the relation representation for a $n \times n$ matrix and $C(ID, vw)$ be a relation representation for a vector with $n$ elements. 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 [49, 36], 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 [66].

### 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 [12].

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 [12], 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 [12], 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 [12] 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 enhanced.

### 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 $n \times n$ $\mathsf{E}$ and a vector $\mathsf{V}$ with $n$ elements. We represent the vector $\mathsf{V}$ 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 $\mathsf{E}$ 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. Note that some graph algorithms can be computed by either union or union-by-update.

We use the examples of *PageRank*, *Floyd-Warshall* and *TopoSort* to illustrate how to support graph algorithms by the new algebra. The relational algebra for *PageRank* is given below.

$$V \leftarrow \rho_V(E \overset{f_1(\cdot)}{\underset{T=ID}{\bowtie}} V) \tag{5}$$

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 $\oplus$.

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

$$E \leftarrow \rho_E((E \rightarrow E_1) \overset{min(E_1.ew+E_2.ew)}{\underset{E_1.T=E_2.F}{\bowtie}} (E \rightarrow E_2)) \tag{6}$$

In Eq (6), we use one MM-join with $+$ and $min$ serving as the $\odot$ and $\oplus$ respectively. For *PageRank* and *Floyd-Warshall*, one union-by-update operation is performed to update recursive relation $V$ and $E$.

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.

$$\begin{aligned}
L_n &\leftarrow \rho_L(\mathcal{G}_{max(L)+1} Topo) \\
V_1 &\leftarrow V \underset{V.ID=T.ID}{\bar{\ltimes}} Topo \\
E_1 &\leftarrow \Pi_{E.F,E.T}(V_1 \underset{ID=E.F}{\bowtie} E) \\
T_n &\leftarrow \Pi_{ID,L}(V_1 \underset{V_1.ID=E_1.T}{\bar{\ltimes}} E_1) \times L_n \\
Topo &\leftarrow Topo \cup T_n
\end{aligned} \tag{7}$$

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*.

| **Graph Algorithm** | MV/MM-join | ⊎/∪ | anti-join | linear | nonlinear |
|---|---|---|---|---|---|
| *TC* [20] | – | ∪ | | ✓ | ✓ |
| *BFS* [20] | MV | ⊎ | | ✓ | |
| *Connected-Component* [57] | MV | ⊎ | | ✓ | |
| *Bellman-Ford* [20] | MV | ⊎ | | ✓ | |
| *Floyd-Warshall* [20] | MM | ⊎ | | | ✓ |
| *PageRank* [42] | MV | ⊎ | | ✓ | |
| *Random-Walk-with-Restart* [42] | MV | ⊎ | | ✓ | |
| *SimRank* [31] | MM | ⊎ | | ✓ | |
| *HITS* [42] | MV | ⊎ | | | ✓ |
| *TopoSort* [34] | – | ∪ | ✓ | | ✓ |
| *Keyword-Search* [16] | MV | ⊎ | | ✓ | |
| *Label-Propagation* [55] | MV | ⊎ | | ✓ | |
| *Maximal-Independent-Set* [46] | – | ∪ | ✓ | | ✓ |
| *Maximal-Node-Matching* [52] | MV | ∪ | ✓ | | ✓ |
| *Diameter-Estimation* [35] | – | ∪ | | ✓ | |
| *Markov-Clustering* [67] | MM | ⊎ | | | ✓ |
| $K$*-core* [43] | MV | ⊎ | | | ✓ |
| $K$*-truss* [54] | – | ⊎ | | | ✓ |
| *Graph-Bisimulation* [30] | – | ∪ | | | ✓ |

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 using rules. The rules for selection, projection, Cartesian product, union, $\theta$-join are given in [66]. In a similar way, we can express the 4 new operators using rules. As union, selection, projection, Cartesian product and $\theta$-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 [71, 72, 14]. An *XY*-stratified program is a special class of locally stratified programs [27]. As proposed by Zaniolo et al. in [71], 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 [12]. An *XY*-program is a locally stratified DATALOG program that can be checked at compile-time by syntax.

Based on *XY*-stratification, 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. We show that the recursive queries using the 4 operations discussed can have a fixpoint by which a unique answer can be obtained [74].

# 6 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 translate the enhanced recursive with into *SQL/PSM* to be processed in *RDBMS*s. In [74], we conduct extensive performance studies to test 10 graph algorithms

using 9 large real graphs on 3 major *RDBMS*s (*Oracle*, *DB2*, and *PostgreSQL*) at *SQL* level, and we show that *RDBMS*s are capable of dealing with graph processing in reasonable time. There is high potential to improve the efficiency by main-memory *RDBMS*s, efficient join processing in parallel, and new storage management.

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

# References

[1] `http://giraph.apache.org`.

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

[3] `http://spark.apache.org/graphx/`.

[4] `http://neo4j.com`.

[5] `http://www.agensgraph.com/`.

[6] `http://thinkaurelius.github.io/titan/`.

[7] 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`.

[8] Microsoft SQL documentation. `https://docs.microsoft.com/en-us/sql/`.

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

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

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

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

[13] 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.

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

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

[16] 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.

[17] 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.

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

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

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

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

[22] 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.

[23] 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.

[24] 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.

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

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

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

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

[29] 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.

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

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

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

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

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

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

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

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

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

[39] C. Lin, B. Mandel, Y. Papakonstantinou, and M. Springer. Fast in-memory SQL analytics on typed graphs. *PVLDB*, pages 265–276, 2016.

[40] H. Ma, B. Shao, Y. Xiao, L. J. Chen, and H. Wang. G-SQL: fast query processing via graph exploration. *PVLDB*, pages 900–911, 2016.

[41] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. of SIGMOD'10*, 2010.

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

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

[44] 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.

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

[46] 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.

[47] V. Z. Moffitt and J. Stoyanovich. Towards a distributed infrastructure for evolving graph analytics. In *Proc. of WWW'16 Companion Volume*, 2016.

[48] W. E. Moustafa, V. Papavasileiou, K. Yocum, and A. Deutsch. Datalography: Scaling datalog graph analytics on graph processing systems. In *2016 IEEE International Conference on Big Data*.

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

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

[51] L. Passing, M. Then, N. Hubig, H. Lang, M. Schreier, S. Günnemann, A. Kemper, and T. Neumann. SQL- and operator-centric data analytics in relational main-memory databases. In *Proc. of EDBT 2017.*, pages 84–95, 2017.

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

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

[54] 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.

[55] 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.

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

[57] 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.

[58] 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.

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

[60] 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.

[61] 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.

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

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

[64] 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.

[65] 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.

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

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

[68] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. PGQL: a property graph query language. In *Proc. of GRADES'16*, 2016.

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

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

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

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

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

[74] K. Zhao and J. X. Yu. All-in-one: Graph processing in rdbmss revisited. In *Proc. of SIGMOD'17*, 2017.