SQL-G: Efficient Graph Analytics by SQL

Kangfei Zhao, Jiao Su, Jeffrey Xu Yu, and Hao Zhang

Abstract—Querying graphs and conducting graph analytics become important in data processing since many real applications are dealing with massive graphs, such as online social networks, Semantic Web, knowledge graphs, etc. Over the years, many distributed graph processing systems have been developed to support graph analytics using various programming models, and many graph querying languages have been proposed. A natural question that arises is how to integrate graph data and traditional non-graph data in a distributed system for users to conduct analytics. There are two issues. One issue is related to expressiveness on how to specify graph analytics as well as data analytics by a querying language. The other issue is related to efficiency on how to process analytics in a distributed system. For the first issue, *SQL* is a best candidate, since *SQL* is a well-accepted language for data processing. We concentrate on *SQL* for graph analytics. Our early work shows that graph analytics can be supported by *SQL* in a way from "semiring + while" to "relational algebra + while" via the enhanced recursive *SQL* queries. In this article, we focus on the second issue on how to process such enhanced recursive *SQL* queries based on the *GAS* (*Gather-Apply-Scatter*) model under which efficient graph processing systems can be developed. To demonstrate the efficiency, we implemented a system by tightly coupling *Spark SQL* and *GraphX* on *Spark* which is one of the most popular in-memory data-flow processing platforms. First, we enhance *Spark SQL* by adding the capability of supporting the enhanced recursive *SQL* queries for graph analytics. In this regard, graph analytics can be processed using a distributed *SQL* engine alone. Second, we further propose new transformation rules to optimize/translate the operations for recursive *SQL* queries to the operations by *GraphX*. In this regard, graph analytics by *SQL* can be processed in a similar way as done by a distributed graph processing system using the APIs provided by the system. We conduct extensive performance studies to test graph analytics using large real graphs. We show that our approach can achieve similar or even higher efficiency, in comparison to the built-in graph algorithms in the existing graph processing systems.

Index Terms—Graph analytics, distributed graph processing, SQL recursive query, spark

1 Introduction

In recent years, researchers in both academia and industry are driven to develop new techniques for querying graphs and conducting graph analytics over massive online social networks, Semantic Web, knowledge graphs, biological networks, and road networks. A large number of graph algorithms have been used/proposed/revisited [7], [8], [13], [57]. In addition to the effort to design efficient graph algorithms to query/analyze large graphs, many distributed graph processing systems have been developed. Recent surveys can be found in [35], [54], [56]. Such distributed graph processing systems provide APIs or domain-specific languages for users to implement graph algorithms. Both existing graph algorithms and distributed graph processing systems focus on efficiency. On the other hand, graph query languages have also been studied [1], [2], [11], [23], [24], [37], [42], [51], [53]. There are languages to deal with pattern matching using conjunctive query, regular path query, or the combination of the both [53], and languages which are graph algebra-based [23], functional-based [2] and *SQL* related [1], [51] for graph pattern matching and graph traversals, not for graph analytics.

An important issue is the integration of distributed graph processing systems and *RDBMS*s to support (i) graph and relational data management and (ii) graph analytics and data analytics in a unified way. For (i), this is motivated by the fact that graph data is used as a part of data together with other non-graph data which are commonly stored as relations in *RDBMS*. In such a way, graph data can be maintained for users to update. However, graph data needs to be extracted into a graph processing system for graph analytic processing, which incurs additional costs. For (ii), in general, the ability of analytics shall enable both graph analytics and other non-graph analytics, where the *RDBMS*s are developed to support data (non-graph) analytics, and graph processing systems are developed for graph analytics. It is challenging to integrate the two systems, since there is a gap between graph processing systems, which use a programming model (e.g., vertex-centric), and *RDBMS*s, which use *SQL* backed up by the techniques developed over decades. It is worth noting that while-loops are not supported by *SQL* directly, whereas iterative computing is assumed by a graph programming model. It requires end-users to know both *SQL* and vertex-centric programming well when there are needs to support both analytics.

To integrate the two systems, we propose a single query language approach (e.g., *SQL*) with an attempt to separate "how" and "what". In other words, the approach we take is for end-users to write "what" analytics they want to do without knowing "how" to achieve the efficiency for such analytics. The focus becomes on how to support graph analytics by *SQL*. In [63], we proposed an *SQL* approach to handle iterative computing by *SQL* recursion. This is

• *The authors are with the Chinese University of Hong Kong, Sha Tin, Hong Kong SAR. E-mail: {kfzhao, jiaosu, yu, hzhang}@se.cuhk.edu.hk.*

Manuscript received 8 Sept. 2018; revised 28 Aug. 2019; accepted 6 Oct. 2019. Date of publication 31 Oct. 2019; date of current version 1 Apr. 2021.
(Corresponding author: Jeffrey Xu Yu.)
Recommended for acceptance by K. Selcuk Candan.
Digital Object Identifier no. 10.1109/TKDE.2019.2950620

```
1. with
2. P(ID, W) as (
3.   (select V.ID, 0.0 from V)
4.   union by update ID
5.   (select E.T, c * sum(W * ew) + (1 − c)/n from P, E
6.   where P.ID = E.F group by E.T)
7.   maxrecursion 10)
8. select ID, W from P
```

Fig. 1. The enhanced recursive *SQL* for *PageRank*.

```
1. with
2. HT(ID, H) as (
3.     (select V.ID, 0.0 from V)
4.     union by update ID
5.     (select F as ID, least(HT'.H', C.W) from
6.         (select E.F, sum(H * ew) + 1 as H' from HT, E
7.             where HT.ID = E.T group by E.F) as HT',
8.         C where F = C.ID)
9.     maxrecursion 10)
10. select ID, W from HT
```

Fig. 2. The enhanced recursive *SQL* for *Hitting-Time*.

challenging because the *SQL* recursion does not support non-monotonic operations (e.g., aggregation and group-by). As given in [63], to support the semiring by which many graph analytics can be supported [28], we proposed two aggregate-joins, namely, MM-join and MV-join. Together with MM-join and MV-join, we also proposed a new union by update relational algebra operation to handle value update. Furthermore, to conduct graph analytics iteratively, we proposed the enhanced recursive *SQL*, which can reach a fixpoint when using the non-monotonic operations (e.g., MM-join, MV-join, union by update) by *XY*-stratification [10], [59], [60]. In other words, we showed that graph analytics that can be supported by "semiring + while" can be supported by "relational algebra + while" via the enhanced recursive *SQL* queries. Below, we give an example to show how to conduct graph analytics by *SQL* (e.g., *PageRank*) using the enhanced recursive *SQL*.

Example 1.1. Consider a web graph G stored in *RDBMS* with two relations, namely, $V(ID, A)$ and $E(F, T, ew)$. Here, $V(ID, A)$ is a relation for vertices, where *ID* is for vertex IDs and A is a set of attributes (e.g., URL, time of creation, *click* rate) of the pages (vertices), and $E(F, T, ew)$ is a relation for links (edges) between pages (vertices), where F and T refer to *ID* in V as the foreign key, and ew is the link attribute (e.g., transition probability). Suppose a user needs to know the commute time (using the hitting time [18]) between the top two backbone vertices based on *PageRank*. First, the user can compute the *PageRank* using an enhanced recursive *SQL* query shown in Fig. 1, as proposed in [63]. It is worth noting that this *PageRank* query (Fig. 1) is not allowed in *SQL'99* since there are non-monotonic operations in the recursion, where *SQL'99* only supports stratified negation. Second, the top-2 *PageRank* tuples, u and v, can be selected using *SQL*. Third, commuting time between u and v can be computed by two hitting time, such as $c_{uv} = h_{uv} + h_{vu}$ where h_{uv} is the hitting time from u to v computed by $1 + \sum_{w \in N^+(v)} h_{uw} * e_{wv}$ for $u \neq v$ and equals to 0 for $u = v$ [18]. Fig. 2 shows the query to compute the hitting time from a fixed vertex u to all the others. Here, $C(ID, W)$ is a relation for vertices, where W is 0 for that fixed vertex u and positive infinity for the others. The function least returns the minimum of H' and $C.W$, which is used to reset the hitting time h_{uu} to 0. It is important to know that users can refer to recursive *SQL* queries using views, which hides the details, to deal with complex graph analytics by *SQL* in *RDBMS*s.

Our approach does not require users to know anything about programming that is needed to implement graph analytics in a distributed graph processing system. Along the direction of supporting graph analytics by *SQL* in [63], in this paper, we further study how to integrate *SQL* engine and graph processing engine and how to support the enhanced recursive *SQL* queries in distributed graph processing systems.

The main contributions of this work are summarized below. Along the direction shifting from "semiring + while" to "relational algebra + while" via the enhanced recursive *SQL* queries, we show how to support the enhanced recursive *SQL* queries with MV-join, MM-join, and union by update by the *GAS* (*Gather-Apply-Scatter*) decomposition [21] under which efficient distributed graph processing systems can be developed. We have implemented our system named *SQL-G* on top of an *SQL* processing engine *Spark SQL* and a graph processing system *GraphX* on *Spark*. First, we implement the enhanced recursive *SQL* [63] on *Spark SQL*. Second, we integrate *Spark SQL* and *GraphX* by converting enhanced recursive *SQL* queries to *GraphX* operators using transformation rules. The advantage is that users do not need to know how the underneath distributed graph processing system is used to support graph analytics specified by *SQL*. We conduct extensive performance studies to test graph algorithms using large real graphs. We show that it is possible to achieve similar performance like *GraphX* when recursive *SQL* queries are used for graph analytics.

The paper is organized as follows. Section 2 introduces the *SQL* recursion, the *SQL* engine and the graph processing engine (i.e., *Spark SQL* and *GraphX*) on which we implement the *SQL-G* system. In Section 3, we give an overview of our system. Section 4 focuses on query transformation to translate recursive *SQL* queries to logical operators to be processed by a graph engine. We discuss logical operators newly added into *Spark SQL* and query transformation rules to translate *SQL* recursive queries into graph-based operations, and query optimization. In Section 5, we further discuss physical transformation to translate a query plan based on the logical operators to a query plan based on the physical operators, under the support of *GraphX* iterative computing. We report our extensive performance studies using iterative graph analytics over real datasets in our system in Section 6. We review the related works in Section 7 and conclude our work in Section 8.

2 Preliminaries

In this section, we introduce the recursive *SQL* for handling graph algorithms [63] and review *Spark SQL* [9] and *GraphX* [22] built on *Spark* [58].

2.1 Graph Algorithms by Recursive SQL

In [63], we model a graph as a weighted directed graph, $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, where $\mathbb{V}$ is the set of vertices and $\mathbb{E}$ is the set of edges. Let $n = |\mathbb{V}|$ denote the number of vertices. Such a graph $\mathbb{G}$ can be represented by a matrix, where the vertices

with vertex-weights vw can be represented as an n-dimensional vector, and the edges with edge-weights ew can be represented as an $n \times n$ matrix. The work in [63] shows that many graph algorithms can be supported by a while-loop using a semiring with two binary operations: the multiplication $(\cdot)$ and the addition $(+)$ as follows.

while ($\mathbb{G}$ changes) {compute $\mathbb{G}$ using the semiring operations}

To support graph algorithms in the form of "while + semiring", in [63], it represents such matrix/vector as relations. Let $E(F, T, ew)$ and $E'(F, T, ew)$ be the relation representation for $n \times n$ matrix, where F and T are short for 'From' and 'To' for edges, and let $V(ID, vw)$ be the relation representation for a vector with n dimensions where ID is the index of the vector (the vertex identifier).

First, to support the semiring, [63] introduces two new operations; namely, MV-join and MM-join, to support the multiplication $(\cdot)$ between matrix and vector and between two matrices, respectively. Note that relational algebra can easily support the addition $(+)$ in the semiring. We show MV-join (Eq. (1)) and MM-join (Eq. (2)) below:

$$E \overset{\oplus(\odot)}{\underset{T=ID}{\bowtie}} V = {}_{F}\,\mathcal{G}_{\oplus(\odot)}\left(E \underset{T=ID}{\bowtie} V\right) \tag{1}$$

$$E \overset{\oplus(\odot)}{\underset{E.T=E'.F}{\bowtie}} E' = {}_{E.F,E'.T}\,\mathcal{G}_{\oplus(\odot)}\left(E \underset{E.T=E'.F}{\bowtie} E'\right). \tag{2}$$

The MV-join in Eq. (1) is used to support the multiplication $(\cdot)$ between a matrix and a vector in their relation representation in two steps. The first step is computing $\odot$ between a tuple (edge) in E and a tuple (vertex) in V under the join condition $E.T = V.ID$. The second step is aggregating all the $\odot$ results by the operation of $\oplus$ for every group by grouping-by the attribute $E.F$. Similarly, the MM-join is done in two steps. The first step computes $\odot$ between a tuple (edge) in E and a tuple (edge) in E' under the join condition $E.T = E'.F$. The second step aggregates all the $\odot$ results by the operation of $\oplus$ for every group by grouping-by the attributes $E.F$ and $E'.T$. In MV-join (Eq. (1)) and MM-join (Eq. (2)), the attribute F and T can be exchanged simultaneously.

Second, to support a while loop, [63] enhances the with clause in *SQL* for handling recursive *SQL* queries, based on *SQL*'99.

with R **as** $\langle R$ initial query $\rangle\langle$ recursive query involving $R\rangle$

In *SQL*'99, the recursive with defines an initial relation R in the initialization step, and queries by referring the R iteratively in the recursive step until R cannot be changed. *SQL*'99 restricts the recursion to be a linear recursion and only supports monotonic queries in the recursive step. Such monotonicity ensures the recursion ends at a fixpoint in the context of stratified negation. It is worth noting that both MV-join and MM-join are non-monotonic since they use aggregation. Therefore, *SQL*'99 cannot handle such non-monotonic operations in recursion. To address it, in [63], it shows that recursions involving MV-join and MM-join do have a fixpoint based on *XY*-stratification.

Example 2.1. As an example given in [63], *PageRank* can be computed using an MV-join until V cannot be changed as follows.

while (V changes) { $V \leftarrow \rho_V(E \overset{f_1(\cdot)}{\underset{F=ID}{\bowtie}} V)$ }

Here, ρ is the rename operator to rename the output of the MV-join as the attributes used in V. $f_1(\cdot)$ is a function to calculate $c * \mathsf{sum}(vw * ew) + (1-c)/n$, where c is the damping factor and n is the total number of tuples in V. Note that $vw * ew$ 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 $\oplus$. Fig. 1 shows the corresponding enhanced recursive *SQL* to compute *PageRank*. In line 3, P is initialized by the projection of *V.ID* with numeric value zero. The recursive step specified by line 5-6 uses an MV-join, where the $\odot$ and $\oplus$ correspond to numeric multiplication and sum, respectively. Instead of union all, the enhanced recursive *SQL* proposed in [63] adopts union by update to update values in a relation (line 4 in Fig. 1).

We further discuss union by update, which is a new relational operation, denoted as $\uplus$, defined in [63] as follows:

$$R \uplus_A S = (R - (R \underset{R.A=S.A}{\ltimes} S)) \cup S. \tag{3}$$

Suppose t_r is a tuple in R and t_s is a tuple in S. Different from the conventional union $(\cup)$, the union by update updates t_r by t_s if t_r and t_s are identical by some attributes A. If t_s does not match any t_r, t_s is merged into the resulting relation. Like MV-join and MM-join, $\uplus$ is a non-monotonic operation. In [63], it proves that $\uplus$ leads to a fixpoint in the enhanced recursive *SQL* queries by *XY*-stratification. Finally, maxrecursion (line 7 in Fig. 1) restricts the max iteration number, which is adopted by *SQL Server*. Such a recursive *SQL* query can be implemented by *SQL/PSM* (Persistent Stored Modules) and executed in major *RDBMS*s, where efficiency is a main issue.

2.2 Spark SQL and GraphX on Spark

Spark [58] has emerged as the de facto distributed big data processing system. It supports a one-stop resolution for big data-crunching needs by combining *MapReduce*-like capabilities for structured-data processing, stream processing, machine learning, and graph processing. The core data abstraction in *Spark* is *RDD* (Resilient Distributed Dataset), which is an immutable in-memory abstraction to be partitioned and computed on a cluster in a fault-tolerant manner. An *RDD* is manipulated by two kinds of operation, namely, transformations and actions. The *Spark* scheduler launches a job, where each job consists of stages to be computed in parallel.

Spark SQL [9] is a structured data processing module on *Spark*. It provides high-performance in-memory relational processing by *RDBMS*s techniques with the support of *SQL*'92 syntax. Note that *Spark SQL* does not support any recursive *SQL* queries as specified in *SQL*'99. For a user-given *SQL* query, the parser builds an unoptimized logical plan composed of logical operators (e.g., join, aggregate). The analyzer looks up meta-data and resolves the logical

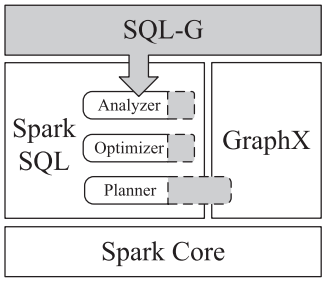


Fig. 3. The architecture of *SQL-G* on *Spark*.

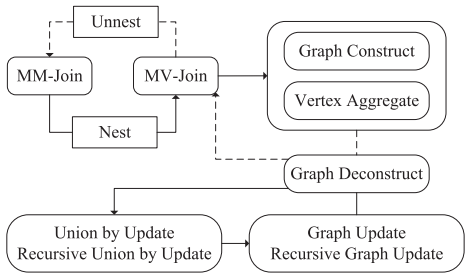


Fig. 4. The logical operators for handling graph.

plan by a collection of analyzing rules. The logical plan is further transformed by both rule-based and cost-based optimizations. In the physical planning phase, the optimized logical plan is transformed into physical plans by a collection of strategies. A physical plan is composed of physical operators (e.g., ShuffleHashJoin, BroadcastJoin) which are implemented by the basic operations on *RDD* s. The core component of *Spark SQL* is the *Catalyst* optimizer. It takes an unoptimized logical plan as input and returns an optimized logical plan. *Catalyst* is highly extensible for developers. *GraphX* [22] on *Spark* represents graph-structured data as a property graph, Graph, which is logically equivalent to a pair of special *RDD*s: VertexRDD and EdgeRDD. The operators supported by *GraphX* are classified into 4 main categories: 1) Property Operators, 2) Structural Operators, 3) Join Operators and 4) Neighborhood Aggregation. In addition, *GraphX* supports custom iterative graph algorithms by the *Pregel* and the *GAS* (*Gather-Apply-Scatter*) [21].

3 An Overview

In this section, we give an overview of *SQL-G*, which deals with both data analytics and graph analytics in a distributed system. For data analytics, it is the best to use *SQL* engine with the data structures designed for such purposes, and for graph analytics, it is the best to use graph processing engine with the data structures designed for such purposes. The two engines need to be developed independently since they deal with different processing tasks. We implement *SQL-G* on top of *Spark* since it has a distributed *Spark SQL* and a distributed graph processing system *GraphX*. We have two options in our system design. One option is to build the system on top of both *Spark SQL* and *GraphX*. In brief, consider the query "*with R as* $\langle R$ initial query $\rangle\langle$ recursive query involving $R\rangle$". We can process the initial query using *Spark SQL* or *GraphX* and process the recursive query in a while loop, where in every iteration the query is evaluated by *Spark SQL* or *GraphX*. This approach has limitations. First, since a recursive relation is maintained by a temporary table instead of *RDD* registered in *Spark SQL* or *GraphX*, in each iteration it needs to register the recursive temporary table as an *RDD* then convert the *RDD* to DataFrame, which is with high overhead. Second, the recursive query needs to be re-analyzed, re-optimized and re-planned in every iteration by *Spark SQL*. Third, *Spark SQL Catalyst* treats each *SQL* query independently. Therefore, it fails to optimize queries. The other option, which we adopt, is as follows. We implement our enhanced recursive *SQL* in *Spark SQL* and invoke *GraphX* from the extended *Spark SQL* for graph analytics when necessary. In such a way, we can resolve the three limitations discussed above. It is worth reemphasizing that, by invoking *GraphX* from the enhanced *Spark SQL* with an internal interface from *Spark SQL* to *GraphX*, we have the flexibility to upgrade our system when *GraphX* is upgraded or is extended with new functions.

The *SQL-G* architecture is illustrated in Fig. 3. In our design, *SQL-G* only accesses *Spark SQL* and does not access *GraphX* directly. We extend *Spark SQL* to support recursive *SQL* queries following our early work in [63]. In doing so, we enhance the *SQL* parser, analyzer, optimizer and query planning strategies of *Spark SQL*. For a user-given *SQL* query as shown in Fig. 1, we build an unoptimized logical plan composed of logical operators by extending *Spark SQL* parser. We provide new logical operators in two categories for *SQL* and graph to be added into *Spark SQL*. In the *SQL* category, we have 3 new logical operators. We add a logical operator for union by update and add two logical operators for recursive with where one is recursive union all, and the other is recursive union by update. In the graph category, we have 9 new logical operators: MV-join and MM-join, nest and unnest, graph-construct and graph-deconstruct, vertex-aggregate, graph-update, and recursive graph-update. The 9 logical operators are shown in Fig. 4. Here, the logical operators, MV-join and MM-join capture such joins in *SQL* explicitly. By the join and aggregation in MV-join, *SQL* fulfills neighborhood message passing and aggregation on vertex so that MV-join is the starting point of transforming the *SQL* logical plan into graph operators. The transformation first uses a graph-constructor to construct a graph based on the inputs of MV-join (i.e., a vertex table and an edge table); second, it computes the aggregate using vertex-aggregate over the graph. To handle MM-join which has two edge tables as input, we convert MM-join to MV-join by converting one edge table as vertex using nest, and then revert such nested table using unnest after the MV-join has been computed. The graph-update updates the vertex attributes for an input graph, and the recursive graph update is the graph version for recursive union by update. We will discuss the 12 logical operators in Section 4.1.

In the logical planning phase, to handle the 9 logical operators for graph computation, we add 6 new transformation rules as shown in Fig. 6. Here, Rule-1 and Rule-2 capture MV-join and MM-join in an *SQL* query. Rule-3 converts an MV-join to graph-construct followed by vertex-aggregate, and Rule-4 converts recursive union by update to recursive graph update. Rule-5 converts an MM-join to an MV-join, and Rule-6 handles recursive nest-update as a

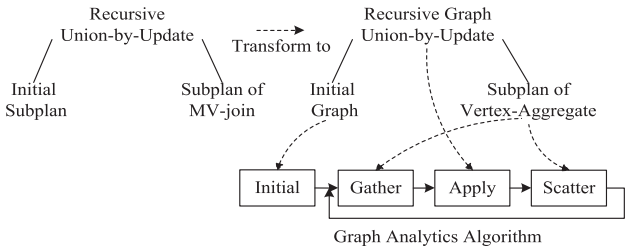


Fig. 5. From recursive *SQL* logical plan to *GAS*.

special union by update. We will discuss the logical transformation rules in Section 4.2.

In the physical planning phase, *SQL-G* further transforms the optimized logical plan with graph support to a physical plan containing *GraphX* operators, which will be executed by *GraphX*. Fig. 5 shows the connection from recursive *SQL* using MV-join to *GAS* (*Gather-Apply-Scatter*). The physical operations for *GraphX* to execute graph analytics are given in Section 5.

Our work is similar to but different from *BigDatalog* [45], which is a *Datalog* system built on *Spark SQL*. The recursion in *BigDatalog* is evaluated by parallel semi-naive evaluation with *RDD* transformation. *BigDatalog* only supports aggregate functions in a limited form (i.e., non-recursive traditional aggregations and recursive monotonic aggregations [46]), and it does not support *XY*-stratification [60] so that it cannot support *PageRank*.

4 Query Transformation

In this section, we discuss how to transform an enhanced recursive *SQL* query to a plan that a graph system can process. Below, we discuss the new logical operators newly added into *Spark SQL* in Section 4.1, the logical transformation from *SQL* to graph in Section 4.2, and new rules for query optimizations in Section 4.3.

4.1 New Logical Operators

We provide new logical operators in two categories for *SQL* and graph to be added into *Spark SQL*. We discuss the 3 *SQL* and 9 graph logical operators below with an overview shown in Table 1.

Union by Update. $R \uplus S$ is used to compute union by update (Eq. (3)).

Recursive Union All. $\mathcal{R}_{\cup}(P_I, P_R, Q)$ is a recursive operator to process a recursive query using union all in the with clause following *SQL'99*. Here, P_I and P_R represent the sub-plan for the initial query and the sub-plan for the recursive query, respectively. Both of the sub-plans are composed of existing *SQL* logical operators. A special logical node represents the recursive relation, which is referred to in the sub-plan P_R and is the output of the recursive operator when the recursion reaches the fixpoint or at the given maximum iteration. Q is a query sub-plan to access the output of the recursive relation.

Recursive Union by Update. $\mathcal{R}_{\uplus}(P_I, P_R, Q)$ is a recursive operator like the recursive operator of $\mathcal{R}_{\cup}(P_I, P_R, Q)$. The difference is that it uses union by update ($\uplus$) instead of union all ($\cup$). Initially, P_I and P_R are evaluated, and the result of $P_I \uplus P_R$ is regarded as the recursive relation to be used in the next iteration. The recursion terminates when $P_I \uplus P_R$ is not changed or at a given maximum iteration. Take *PageRank* (Fig. 1) as an example. The recursive relation is P, P_I is for the initial query (line 3), and P_R is for the recursive query (line 5-6) which refers to the recursive relation P, and Q is the query to trigger the recursion (line 8).

TABLE 1
The 12 Logical Operators

Logical Operator	Notation
Recursive union all	$\mathcal{R}_{\cup}$
union by update / Recursive union by update	$\uplus$ / $\mathcal{R}_{\uplus}$
MV-Join / MM-Join	$\overset{\oplus(\odot)}{\underset{\text{MV}}{\bowtie}}$ / $\overset{\oplus(\odot)}{\underset{\text{MM}}{\bowtie}}$
nest / unnest	ν / μ
Graph-Construct / Graph-Deconstruct	$\mathcal{C}_G$ / $\mathcal{D}_V, \mathcal{D}_E$
Vertex-Aggregate	$\mathcal{G}_{(\odot,\oplus)}$
Graph-Update / Recursive Graph-Update	$\uplus_G$ / $\mathcal{R}_{\uplus_G}$

MV-Join & MM-Join. $E \overset{\oplus(\odot)}{\underset{\text{MV}}{\bowtie}} V$ and $E \overset{\oplus(\odot)}{\underset{\text{MM}}{\bowtie}} E'$ compute matrix-vector (Eq. (1)) and matrix-matrix (Eq. (2)) multiplication respectively. Here, V represents a vector relation; E and E' represent two matrix relations. Two binary operations $\oplus$ and $\odot$ specify the corresponding addition and multiplication of the matrix multiplication, respectively.

Nest & Unnest. The nest and unnest operators are represented by $\nu(R)$ and $\mu(R)$, respectively. Assume the schema of the relation R is $(A_1, \ldots, A_n, B_1, \ldots, B_m)$, where all the attributes are atomic. The nest operator $\nu_{(B_1,\ldots,B_m)\rightarrow B}(R)$ generates a nested relation $(A_1, \ldots, A_n, B)$, where B is a relation with attributes $(B_1, \ldots, B_m)$. It groups all $B_1, \ldots, B_m$ values into a single nested relation if the corresponding $A_1, \ldots, A_n$ values are identical. The inverse operation of nest (ν) is unnest (μ) which makes a nested relation flat.

Graph-Construct & Graph-Deconstruct. The logical operator $V\,\mathcal{C}_G\,E$ builds a graph by two relations; V for vertices and E for edges. Graph-deconstruct is the inverse operation of graph-construct. There are two logical operators, $\mathcal{D}_V(G)$ and $\mathcal{D}_E(G)$, which take a graph G and return the vertex and edge relation as output, respectively.

Vertex-Aggregate. The logical operator $\mathcal{G}_{(\odot,\oplus)}(G)$ conducts vertex-centric aggregation for an input graph G. The output of this logical operator is a vertex relation where each row contains a vertex *ID* and an aggregated message.

Graph-Update. $G \uplus_G V$ updates the vertex attribute of graph G by a vertex relation $V(ID, A)$. The output of this operator is the graph being updated. In other words, for any vertex v in G, if there exists $u \in V$ such that $v.ID = u.ID$, the attribute of v in G is updated by $u.A$ in V.

Recursive Graph-Update. $\mathcal{R}_{\uplus_G}(I_G, R_G, Q_G)$ is a graph-based recursive operator for recursive union by update. Here, I_G, R_G, Q_G are the initial, recursive, and query sub-plans, respectively, and each sub-plan is composed of at least one graph-based logical operator. More precisely, I_G is the initial sub-plan returning a graph, and R_G is the recursive sub-plan returning a vertex relation. Different from $P_I \uplus P_R$ used in the recursive union by update, in the recursive graph-update,

Rule-1 : $_{E.F}\mathcal{G}_{\oplus(\odot)}(E \underset{E.T=R.ID}{\bowtie} R) \Rightarrow E \overset{\oplus(\odot)}{\underset{\text{MV}}{\bowtie}} R$

Rule-2 : $_{E.F,R.T}\mathcal{G}_{\oplus(\odot)}(E \underset{E.T=R.F}{\bowtie} R) \Rightarrow E \overset{\oplus(\odot)}{\underset{\text{MM}}{\bowtie}} R$

Rule-3 : $E \overset{\oplus(\odot)}{\underset{\text{MV}}{\bowtie}} R \Rightarrow \mathcal{G}_{(\odot,\oplus)}(R\, \mathcal{C}_G\, E)$

Rule-4 : $\mathcal{R}_{\uplus}(R_0, \mathcal{G}_{(\odot,\oplus)}(R\, \mathcal{C}_G\, E), Q(R))$
$\Rightarrow \mathcal{R}_{\uplus_G}(G_0, \mathcal{G}_{(\odot,\oplus)}(G), Q_G)$

Rule-5 : $E \overset{\oplus(\odot)}{\underset{\text{MM}}{\bowtie}} R \Rightarrow \mu(E \overset{\widetilde{\oplus}(\widetilde{\odot})}{\underset{\text{MV}}{\bowtie}} \nu(R))$

Rule-6 : $\mathcal{R}_{\uplus}(R_0, P_R, Q(R)) \Rightarrow \mathcal{R}_{\widetilde{\uplus}}(\nu(R_0), \nu(P_R), Q(\mu(\nu(R))))$

Fig. 6. SQL-to-graph transformation rules.

$I_G \uplus_G R_G$ generates a recursive graph for the next iteration, instead of a recursive relation. If $I_G \uplus_G R_G$ does not update any vertex or reaches a given maximum iteration; the recursion ends with evaluating the query Q_G finally.

4.2 Logical Transformation: SQL-to-Graph

In this section, we focus on query transformation from *SQL* to graph operators and mainly discuss the transformation rules regarding the 9 logical operators in the graph category. Fig. 6 shows the logical transformation rules supported by *SQL-G*. Here, a rule in the form of $P_1 \Rightarrow P_2$ indicates it transforms a plan P_1 to an equivalent plan P_2. Below, we discuss 6 new rules added in *Spark SQL*. Rule-1 and Rule-2 identify MV-join and MM-join in *SQL* explicitly. Rule-3 and Rule-4 execute MV-join in a recursive union by update in *SQL* by the vertex-aggregate in a recursive graph-update. And, Rule-5 and Rule-6 execute MM-join in a recursive union by update in *SQL*.

MV-Join and MM-Join Identification. Rule-1 and Rule-2 are shown in Fig. 6 to identify MV-join and MM-join in the *SQL* logical plan explicitly.

Execute MV-Join in Recursive Union by Update. Here, assume that by Rule-1 in Fig. 6, we have identified an MV-join ($E \overset{\oplus(\odot)}{\underset{\text{MV}}{\bowtie}} R$). By Rule-3, we first construct a graph G by the logical graph-construct operator $R\mathcal{C}_G\, E$ where R represents the vertex relation, and E represents the edge relation. Then we use the logical vertex-aggregate operator $\mathcal{G}_{(\odot,\oplus)}(G)$ to compute the MV-join, suppose an MV-join can be executed by $\mathcal{G}_{(\odot,\oplus)}(R\mathcal{C}_G\, E)$ using Rule-3. Next, we discuss how to compute the MV-join (i.e., $\mathcal{G}_{(\odot,\oplus)}(R\mathcal{C}_G\, E)$) in the logical operator of recursive union by update using the logical operator of recursive graph-update. We explain Rule-4. In Rule-4, the left side is the recursive union by update ($\mathcal{R}_{\uplus}$); and the right side is the recursive graph-update ($\mathcal{R}_{\uplus_G}$). In $\mathcal{R}_{\uplus}$, R is the recursive relation, R_0 is the initial R to be used in the recursion, and the recursive query is $\mathcal{G}_{(\odot,\oplus)}(R\mathcal{C}_G\, E)$, which is a graph-based logical operator for the MV-join (refer to Rule-3). To transform $\mathcal{R}_{\uplus}$ to $\mathcal{R}_{\uplus_G}$, R_0 is transformed to $G_0 = R_0\, \mathcal{C}_G\, E$, since $G = R\mathcal{C}_G\, E$. Also, in the transformation, $Q(R)$ is transformed to $Q_G = Q(\mathcal{D}_V(G))$. Here, $Q(R)$ indicates that a query Q accesses the recursive relation R. Recall *PageRank* in Fig. 1, where P is the recursive relation, and Q refers to line 8 in Fig. 1 to access the recursive relation P. On the other hand, $\mathcal{D}_V(G)$ deconstructs the graph G into a vertex relation which is the same as R in $Q(R)$.

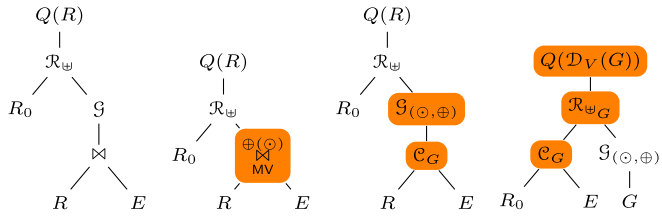


Fig. 7. MV-join in recursion.

We show how to transform *PageRank* by *SQL* in Example 2.1 (Fig. 1) using logical graph-based operators. In brief, *PageRank* uses an MV-join and union by update in recursion. We show the transformation step-by-step in Fig. 7. The *SQL* logical plan is shown in Fig. 7a. Here, we use R to indicate the recursive relation, which is P in Fig. 7, and R_0 to indicate the initial recursive relation. First, Rule-1 captures the MV-join and transforms the plan in Fig. 7a to the plan in Fig. 7b. Second, Rule-3 transforms the MV-join to the vertex-aggregate ($\mathcal{G}_{(\odot,\oplus)}$) over a graph which is constructed by the graph-construct $R\mathcal{C}_G\, E$ in Fig. 7c. Regarding the recursive union by update ($\mathcal{R}_{\uplus}$), as shown in Fig. 7c, its initial query is R_0 on left, its recursive query is rooted at the vertex-aggregate ($\mathcal{G}_{(\odot,\oplus)}$), and $Q(R)$ on top will get the final recursive relation R when the recursion terminates. Third, by applying Rule-4, the recursive union by update is transformed to the recursive graph-update as shown in Fig. 7d. Here, the recursive relation R becomes a graph G, R_0 is transformed to an initial graph G_0 by $R_0\, \mathcal{C}_G\, E$, and $Q(R)$ becomes $Q_G = Q(\mathcal{D}_V(G))$.

Execute MM-Join in Recursive Union by Update. To execute MM-join by graph processing systems (e.g., *GraphX*), we transform an MM-join to an MV-join using the two logical operators, nest and unnest. Here, nest turns a set of edges (pairs of vertices) from a single source vertex, v_s, to a single pair (v_s, N_s) where N_s is the neighbors of v_s, and unnest turns (v_s, N_s) to a set of edges from the same source vertex v_s in reverse. In other words, the transformation of an MM-join to an MV-join is done by treating a set of edges as (v_s, N_s) for every source vertex v_s. We give details below. Let the input relations of MM-join be $R(F, T, ew)$ and $E(F, T, ew)$, where $R(F, T, ew)$ is supposed to be the recursive relation. By nesting the attributes (T, ew) in $R(F, T, ew)$, we obtain a nested relation $R_N(F, SV)$, where SV is a relation on the schema (T, ew) in Eq. (4)

$$\nu_{(T,ew)\rightarrow SV}(R) = R_N(F, SV). \tag{4}$$

With the nested relation R_N, MM-join can be transformed into an MV-join between E and R_N by Eq. (5) followed by unnesting the MV-join result by Eq. (6)

$$E \overset{\widetilde{\oplus}(\widetilde{\odot})}{\underset{E.T=R_N.F}{\bowtie}} R_N = {}_{E.F}\,\mathcal{G}_{\widetilde{\oplus}(E.ew\widetilde{\odot}SV)}\left(E \underset{E.T=R_N.F}{\bowtie} R_N\right) \tag{5}$$

$$E \overset{\oplus(\odot)}{\underset{E.T=R.F}{\bowtie}} R = \mu_{SV\rightarrow(T,ew)}\left(E \overset{\widetilde{\oplus}(\widetilde{\odot})}{\underset{E.T=R_N.F}{\bowtie}} R_N\right). \tag{6}$$

Here, $\widetilde{\odot}$ and $\widetilde{\oplus}$ operate on the nested relation $R_N.SV$ for $\odot$ and $\oplus$ respectively as given in Eqs. (7) and (8), where $\rho_A(R)$

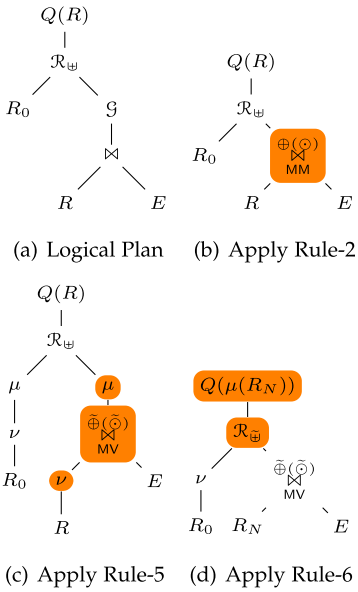


Fig. 8. MM-join to MV-join.

is the rename operation in the relational algebra to rename the relation R to A

$$E.ew \widetilde{\odot} SV = \rho_{SV'}(\Pi_{SV.T,\ E.ew \odot SV.ew}(SV)) \tag{7}$$

$$\widetilde{\oplus}(SV') = {}_{SV'.T}\,\mathcal{G}_{\oplus(ew)}(SV'). \tag{8}$$

Fig. 6 shows how Rule-5 is used to transform MM-join into MV-join based on Eqs. (4)-(8).

Example 4.1. The all pairs shortest distances are computed by the MM-join ${}_{E.F,R.T}\mathcal{G}_{\min(E.ew+R.ew)}(E \underset{E.T=R.F}{\bowtie} R)$, where $R(F,T,ew)$ is the recursive relation which maintains the distances from F to T, the operations $+$ and min correspond to the $\odot$ and $\oplus$ in $\overset{\oplus(\odot)}{\underset{\text{MM}}{\bowtie}}$ respectively. The nested relation R_N (Eq. (4)) denotes for each F, there is a relation $SV(T,ew)$ which preserves the distance ew from F to T. By applying Rule-5 to transform the MM-join into MV-join, $+$ is transformed to $\widetilde{+}$ (Eq. (7)) to perform $+$ on $E.ew$ and each ew in SV so that the result is also a relation. And min is transformed to $\widetilde{\min}$ (Eq. (8)) which groups SV for each F, computing min on $SV.ew$ for each $SV.T$.

The main reason to execute MM-join by MV-join is MV-join can be supported by the graph-based operators in a recursive union by update. However, there is a problem. We need to do nest (Eq. (4)) and unnest (Eq. (6)) in every iteration, which is inefficient. To resolve this problem, we allow union by update and recursive union by update to directly deal with nested relations as a special case. Let the union by update over two nested relations, denoted as $R_N \uplus S_N$, where SV in R_N and S_N is also a relation. For tuple r_N in R_N and s_N in S_N with identical F value, $r_N.SV \uplus s_N.SV$ will be conducted instead of replacing $r_N.SV$ by $s_N.SV$. Rule-6 (in Fig. 6) is used to avoid explicit nest (ν) and unnest (μ) in every iteration by (i) keeping the nested relation structure in the recursive union by update and (ii) evaluating the unnest only once at the end of the recursion but before evaluating the query Q.

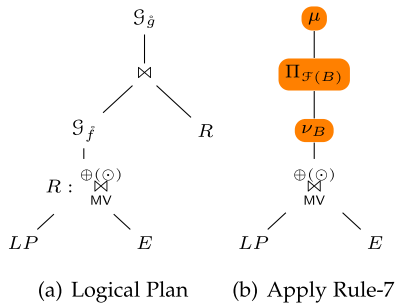


Fig. 9. A complex query.

We show the transformation from MM-join to MV-join in the recursive union by update using nest and unnest in Fig. 8. Here, in Fig. 8, R and E are two relations representing matrices (edges), R is the recursive relation, and R_0 is the initial recursive relation. Rule-2 in Fig. 6 captures the MM-join and transforms the plan in Fig. 8a to the plan in Fig. 8b. Then, Rule-5 transforms the plan to be a plan in Fig. 8c, where $R_0 = \mu(\nu(R_0))$. Let $R_N = \nu(R)$ be the recursive nested relation. $\nu(R_0)$ is the initial recursive nested relation. By applying Rule-6, the plan becomes Fig. 8d. Given the plan in Fig. 8d, we further use a similar way to transform MV-join in recursive union by update by nested relations to MV-join in the recursive union by update.

4.3 Query Optimization

In this section, we discuss two query optimization techniques to (i) remove some unnecessary joins and (ii) avoid computing a join in recursion.

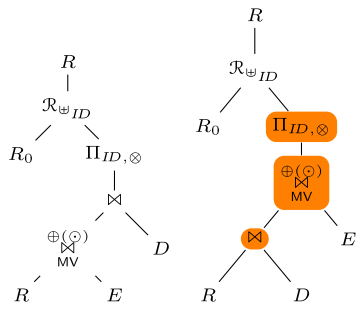
We explain how to remove some unnecessary joins using an example. Consider *Label-Propagation* [39] which is an algorithm to find communities in a network $G = (V, E)$. Initially, every vertex, u, is associated with a label which is its vertex ID. In every iteration, the label of every vertex, u, is reassigned to be the smallest label among the most frequent labels from u's neighbors' labels. The label updating repeats until all labels in G do not change or a maximum iteration number is reached. Then, all vertices that have the same label are in the same community. Let LP be a vertex relation on $LP = (ID, L, C)$ where L is the label, and C is the count. Initially, L's value is assigned to be ID's value, which is unique, and $C = 1$. *Label-Propagation* is implemented by computing and updating relation LP using the following 3 equations iteratively:

$$R \leftarrow E \overset{\text{sum}(C)}{\underset{E.T=ID}{\bowtie}} LP \tag{9}$$

$$R_1 \leftarrow \rho_{(ID,C')\,ID}\mathcal{G}_{\max(C)}(R) \tag{10}$$

$$LP \leftarrow {}_{ID}\mathcal{G}_{\min(L)}\left(R_1 \underset{R_1.ID=R.ID R1.C'=R.C}{\bowtie} R\right). \tag{11}$$

Here, $R(ID, L, C)$ is computed by Eq. (9), where L is the distinct label and C is the count of the neighbors of vertex that have the label L. Eq. (10) picks the vertex ID and its maximum count in R_1, where ρ is a rename operator. In Eq. (11), the new label of a vertex is determined by the min label in R whose count, which is the max, is in R_1. Note that Eq. (9) is



(a) Logical Plan (b) Apply Rule-8

Fig. 10. Avoid computing joins in recursion.

an MV-join, Eq. (10) is an aggregation, and Eq. (11) an equijoin followed by grouping-by and aggregation. Fig. 9a shows its query plan.

We show that we can compute *Label-Propagation* (Fig. 9a), without the equijoin in Eq. (11), by applying the two aggregate functions over a nested relation instead. We do so with Rule-7. Consider a relation $R = (A, B_1, \ldots, B_m)$, where A is a set of non-null atomic attributes and B_i is a non-null atomic attribute for $1 \leq i \leq m$. Let $\nu_B(R)$ be short for $\nu_{B_1,\ldots,B_m \to B}(R)$. The Rule-7 is given in Eq. (12)

$$\text{Rule-7}: {}_A\,\mathcal{G}^{\mathring{g}}(\rho_{R_1}({}_A\mathcal{G}^{\mathring{f}}(R)) \underset{\substack{R_1.A=R.A \\ R_1.\mathring{f}=R.B_i}}{\bowtie} R) \Rightarrow \mu_B(\Pi_{A,\mathcal{F}(B)}(\nu_B(R))). \tag{12}$$

Here, $\mathring{f}$ and $\mathring{g}$ are two aggregate functions that compute aggregation for every $(B_1, \ldots, B_m)$ tuple, such that $\mathring{f}(B) = \mathring{f}(B_1, \ldots, B_m)$ and $\mathring{g}(B) = \mathring{g}(B_1, \ldots, B_m)$. And $\mathcal{F}(B) = \mathcal{G}^{\circ}_{g(B)}(\sigma_{B_i=\mathcal{G}^{\mathring{f}}(B)}(B))$. For the query plan in Fig. 9a, the resulting query plan by Rule-7 is shown in Fig. 9b.

Next, we discuss how to avoid computing joins in recursion. As examples, in computing *Random-Walk-with-Restart* [34] and *Hitting-Time* [18], etc, it needs to compute an MV-join with another join in recursion. We illustrate such a query plan in Fig. 10a, which is a recursive union by update query, where R is a recursive relation and R_0 is the initial relation. There is an MV-join followed by an equijoin. The equijoin needs to be conducted in every iteration in the recursion with high overhead.

To avoid computing such joins in every iteration, we use the following Rule-8 conditionally

$$\text{Rule-8}: \left(E \overset{\oplus(\odot)}{\underset{\text{MV}}{\bowtie}} R\right) \bowtie D \Rightarrow E \overset{\oplus(\odot)}{\underset{\text{MV}}{\bowtie}} (R \bowtie D). \tag{13}$$

Assume R represents vertices, and E represents edges. The first condition is that there is an edge for every vertex in R. In other words, R will be updated in every iteration as a while. The second condition is that R is a subset of D, such that some additional attributes in D may be added by the equijoin, but all the tuples of R will be retained in the result. Such conditions can be done by some preprocessing. Under the two conditions, $(E \overset{\oplus(\odot)}{\underset{\text{MV}}{\bowtie}} R) \bowtie D$ can be done by $E \overset{\oplus(\odot)}{\underset{\text{MV}}{\bowtie}} (R \bowtie D)$. It is worth noting that all tuples in R will be involved in every iteration, and the equijoin $R \bowtie D$ only needs to be done once before the recursion. Accordingly, R_0 in Fig. 10a, will be replaced by $R_0 \bowtie D$ in Fig. 10b. This optimization reduces the processing cost significantly.

5 Physical Transformation

The logical plans above are translated into physical plans comprising *Spark RDD* transformations and *GraphX* operators by new planning strategies. We first discuss how to support *SQL*-based operators on *Spark SQL* followed by graph-based operators on *GraphX*. *SQL-G* transforms the recursive *SQL* execution to graph-based operators on *GraphX* under *GAS* model as Fig. 5 illustrates. Our transformation rules can be used when it needs to transform the logical graph operators to other models (e.g., *Pregel*, Scatter-Gather, etc.). Unlike *GAS* that strictly decomposes vertex and edge specific computation to address skewness of workload, *Pregel* uses message combiner to implement *Gather*, and merge *Apply* and *Scatter* to the vertex program [21], and Scatter-Gather expresses *Apply* used in *GAS* at the end of *Gather* when vertices gather messages from their in-coming edges. It is important to mention that we hide the models from end-users and make it possible to isolate *SQL* specifications on graph analytics and the efficient implementation of graph processing systems.

Union by Update. $R \uplus_A S$ (Eq. (3)) is computed in *Spark SQL* using a full outer join ($*\!*\bowtie$) [25], [63] as follows:

$$R \uplus_A S = \Pi_{\mathsf{coal}(R.A,S.A),\mathsf{coal}(S.B,R.B)}\left(R \underset{R.A=S.A}{*\!*\bowtie} S\right). \tag{14}$$

Here, the resulting relation is $(R.A, R.B, S.A, S.B)$, where B represents the attributes that are not involved in A. If one tuple $t_r \in R$ fails to join any tuple in S, the $(S.A, S.B)$ value of t_r is null, vice versa. Note that $\mathsf{coal}(x, y)$ is a function, supported by *Spark SQL*, which returns x if it is non-null, otherwise y.

Recursive Union All & Union by Update. The two logical operators, $\mathcal{R}_{\cup}$ and $\mathcal{R}_{\uplus}$, are implemented using *RDD* with row type. Let rdd_R and rdd_S be two *RDD*s for the result of the initial query and recursive query, respectively. For recursive union all, we use rdd_R.union (rdd_S) to union all the results of initial evaluation rdd_R and the result of recursive evaluation rdd_S. The generated *RDD* replaces out-dated recursive *RDD* in the catalog and rdd_S to be the input of the next iteration. The operator terminates recursion when rdd_S.count equals to 0. For recursive union by update, the generated *RDD* from union by update replaces the recursive *RDD* in the catalog and becomes the input of the next iteration. The recursion ends when the generated *RDD* is identical to rdd_R. To reduce the shuffling cost, the same partition strategy is used for all the recursive *RDD* s generated.

MV-Join & MM-Join. There is no need to support these two logical operators. Since MV-join is supported by the logical vertex-aggregate operator by Rule-3 and MM-join is further transformed to MV-join by Rule-5.

Graph-Construct & Graph-Deconstruct. $V\,\mathcal{C}_G\,E$ builds a graph by the constructor of *GraphX* Graph ($\mathsf{rdd}_V, \mathsf{rdd}_E$) where rdd_V and rdd_E are two *RDD* s of V and E. The attributes of vertex *ID*, edge source and destination *ID*, maintained in the logical operator, are mapped to the fields of the rdd_V and rdd_E, respectively. Graph-deconstruct $\mathcal{D}_V$ ($\mathcal{D}_E$)

is implemented by returning the vertices (edges) attribute of Graph. When a graph is built, *SQL-G* computes some statistics on the graph, (e.g., the degree distribution, self-loop, and edge direction), which is conducted by several *RDD* transformations efficiently, and is used in the vertex -aggregate to be discussed next. We explain degree distribution and edge direction below and will explain self-loop when we discuss the vertex-aggregate.

- We estimate the degree distribution in order to select a graph partitioning strategy to partition the graph over computational nodes in a distributed platform. When a graph is built, our system decides a partitioning strategy to be used in *GraphX*. According to [52], for *GraphX*, the Canonical Random works well with low degree graphs, and the 2D Edge partitioning works well with heavy-tail graphs. We use the Kurtosis to evaluate the skewness of the degree distribution [16], as defined by Eq. (15), where X is the random variable denoting the vertex degree, α and σ are its mean and standard deviation, respectively

$$Kurt[X] = E\left[\left(\frac{X-\alpha}{\sigma}\right)^4\right] = \frac{E[(X-\alpha)^4]}{(E[(X-\alpha)^2])^2}. \quad (15)$$

  If the kurtosis is negative ($Kurt[X] < 3$), we adopt the Canonical Random partitioning. Otherwise, 2D Edge partitioning is chosen. It is worth mentioning that this statistic can be computed efficiently by several *RDD* transformations. For large graphs, we sample a fixed number of vertex uniformly.
- We also check the edge direction in order to determine whether the graph to be constructed is a directed graph or an undirected graph. We observe that it is storage and communication efficient to perform computation on an undirected graph physically built in the system, if the graph to be constructed is undirected.

Vertex-Aggregate. This logical operator, $\mathcal{G}_{(\odot,\oplus)}(G)$ computes MV-join (Eq. (1)) over a graph constructed. This is a core operation of many graph algorithms to be executed in *GraphX*. Here, the two operations $\odot$ and $\oplus$ form the matrix-vector multiplication $(\cdot)$ in [63]. In the physical planning phase, *SQL-G* captures the algebra characteristic to enable optimization. Over a domain $\mathbb{D}$, if the binary $\odot$ has identity element $\mathbf{1}_\odot$, $a_i \odot \mathbf{1}_\odot$ equals to a_i for any $a_i \in \mathbb{D}$. As an aggregation operator, $\oplus$ is commutative and associative. If and only if for any $a_i, a_j \in \mathbb{D}$ $(a_i \neq a_j)$, $a_i \oplus a_j = a_i$, we say $\oplus$ has a *strict partial order* $\prec_\oplus$ and a_i *dominates* a_j, denoted as $a_i \prec_\oplus a_j$. Note that not all aggregation operators have strict partial order. For example, for min over real numbers, the strict partial order $\prec_{\mathsf{min}}$ is numeric $<$. The sum over nonnegative real numbers does not have a strict partial order.

We transform $\mathcal{G}_{(\odot,\oplus)}(G)$ to execute aggregateMessages on Graph, which is the core neighborhood aggregation operation in *GraphX*. To conduct vertex-centric computation, this operation sends a message along the edge by user-defined function sendMsg. The messages sent to one destination are aggregated to yield a single message by function mergeMsg. By taking advantage of the characteristic of $\oplus$, $\odot$ and the meta-data (e.g., edge direction and self-loop) collected during graph construction, *SQL-G* generates a specialized aggregateMessages operation. It avoids passing redundant messages and allows *GraphX* to select the optimized join strategy. We discussed the edge direction, and discuss self-loop below.

- We check self-loops in order to determine whether each vertex has self-loop so that message reduction can be applied in the Vertex-Aggregate physical operator. As the Vector-Aggregate keeps the semiring operation, i.e., $\odot$ and $\oplus$, unnecessary message passing can be avoided if the semiring has some specific algebraic characteristics when each vertex has self-loops. More specifically, if the $\oplus$ operation has a strict partial order and vertex with weight vw has self-loop of weight ew_{se}, any message dominated by $vw \odot ew_{se}$ can be reduced in the aggregation of $\oplus$. This optimization is reflected in the physical Vertex-Aggregate operator and the Scala code generated.

Graph-Update & Recursive Graph-Update. $\uplus_G$ updates the vertex attributes of G. Note that updating in place is much efficient than updating $\mathcal{D}_V(G) \uplus_{ID} V$ followed by reconstructing the graph. We use *GraphX* JoinVertices API to update the vertex attribute of a Graph. Here, rdd_V is the *RDD* abstraction of V and updateFunc is a function updating the vertex attribute of G to the value of rdd_V, if vertex *ID* equals to the key of rdd_V. Recursive graph-update is implemented in a similar manner as to implement the recursive union by update, except it is for the Graph. Recursive graph is maintained in the graph catalog. In each iteration, JoinVertices updates Graph until rdd_V.count is 0 or the specified iteration time is reached. The newly updated Graph replaces out-dated Graph for the next iteration.

We explain how vertex-aggregate $\mathcal{G}_{(\odot,\oplus)}(G)$ is related to the *Scatter* and *Gather* stages of the GAS vertex program [21] by *PageRank* (Example 2.1). In the optimized logical plan of *PageRank* query, which is Fig. 7d, *SQL-G* generates the *GraphX* operation aggregateMessages for logical operator $\mathcal{G}_{(\odot,\oplus)}(G)$ whose aggregation $\oplus$ is sum over nonnegative real values and $\odot$ is the numeric multiplication. On the edges of Graph, sendMsg performs the *Scatter* phase of *GAS* model by computing the *PageRank* value multiplying the transitive probability of the edge. The result of the multiplication is scattered on the edge unidirectionally for directed graph or bi-directionally for undirected graph. Then, the function mergeMsg performs the *Gather* phase by taking the messages on the edges and calculating the numeric sum to generate a Σ for destination, which is commutative and associative. The physical operation returns Σ for further *Apply* phase. The graph-update corresponds to the *Apply* phase of the *GAS* where a vertex program is evaluated to update the old value to new value. Consider the *PageRank*, a projection $0.85 * \Sigma + 0.15$ on Σ updates the old *PageRank* value on Graph.

Nest & Unnest. Two physical operators are implemented to support nest (ν) and unnest (μ). Consider a flat relation $R(A_1, \ldots, A_m, B_1, \ldots, B_n)$. Here, we implement nest, $\nu_{(B_1,\ldots,B_n)\rightarrow B}$, by using a new datatype SparseVector to store B as an $(n-1)$-dimensional array. On the other hand, unnest (μ) flattens a relation containing the SparseVector attribute to atomic attributes. Recall MM-join is

TABLE 2
The Real Datasets

Graphs	Source	$\|V\|$	$\|E\|$	$Kurt[X]$
roadNet-CA (CA)	[5]	1.9M	2.7M	2.1
roadNet-USA (US)	[5]	23.9M	29.1M	2.5
LiveJournal (LJ)	[30]	3.9M	34.6M	7.2E+3
Orkut (OK)	[30]	3.0M	117.1M	6.1E+3
Wiki Vote (WV)	[30]	7.1K	103.6K	49.47
U.S. Patent Citation (PC)	[30]	3.7M	16.5M	1.8E+2
arabic-2005 (AR)	[12]	22.7M	639.9M	3.7E+5
uk-2005 (UK)	[12]	39.4M	936.3M	1.1E+6
Twitter (TT)	[29]	41.6M	1.5B	3.5E+5

transformed to MV-join by Rule-5 in the form of $\mu(E \overset{\widetilde{\oplus}(\widetilde{\odot})}{\underset{\text{MV}}{\bowtie}} \nu(R))$, where $\widetilde{\oplus}$ and $\widetilde{\odot}$ are $\oplus$ and $\odot$ for handling nested relations, respectively. To support $\widetilde{\oplus}$ and $\widetilde{\odot}$, we add arithmetic and aggregate expressions for SparseVector. The algebra characteristics of $\widetilde{\oplus}$ and $\widetilde{\odot}$ are preserved on nested relations. Hence, transformation for vertex-aggregate can be applied to generate the physical operator for the nested version of MV-join using $\widetilde{\oplus}$ and $\widetilde{\odot}$.

6 Performance Studies

In this section, we present the performance study for *SQL-G*, which is implemented on top of *Spark SQL* and *GraphX* on *Spark*. The experiment is designed to show that graph analytics by *SQL* as we proposed can achieve a similar performance as done by vertex-centric programming in a distributed platform (e.g., *GraphX*). In other words, we show that a similar performance, to be achieved by an expert on vertex-centric programming, can be achieved by a user who only knows *SQL* and does not know anything about vertex-centric programming.

We compare *SQL-G* with *GraphX* since *SQL-G* uses *GraphX* as the basis to perform graph analytics when possible. In addition to *GraphX*, we also compare *SQL-G* with 2 systems that are developed on *Spark*, namely, *GraphFrames* [15] and *BigDatalog* [45]. Here, *GraphFrames* is a graph processing system, which maintains vertices and edges as DataFrame (instead of *RDD* used in *GraphX*), and processes graph analytics using DataFrame operations. *BigDatalog* is a distributed system that supports graph analytics at a high level using *Datalog*. *GraphX*, *GraphFrames* and our *SQL-G* are running on *Spark* 2.2, whereas *BigDatalog* is running on *Spark* 1.6. All of the systems are built on *Hadoop* 2.6.0.

Graph Algorithms. We select the graph algorithms to test that are *GraphX* and *GraphFrames* built-in graph algorithms including the single source shortest path (SSSP) by *Bellman-Ford* [14], weakly *Connected-Component* (WCC) [41], *PageRank* (PR) [34] and *Label-Propagation* (LP) [39]. We use the *Datalog* programs given in their paper to implement SSSP and WCC on *BigDatalog*. *BigDatalog* does not support PR and LP since *BigDatalog* cannot guarantee a fixpoint when aggregations need to be computed iteratively. For PR and LP, the number of iterations is fixed to 20. In addition, we set the upper bound of the number of iterations for WCC and SSSP to 100 to avoid excessive long time running.

Datasets. We conducted our testing using 9 real datasets, as shown in Table 2. The datasets are obtained from DIMACS [5], SNAP [30], WebGraph Datasets [12] and Twitter [29]. The first 4 datasets are undirected graphs and the remaining 5 are directed graphs. An undirected graph is represented as a directed graph by including two directed edges for an undirected edge. For algorithm WCC, a directed graph is treated as an undirected graph. The last column in Table 2 is the kurtosis (Eq. (15)) of the vertices degree which *SQL-G* uses to evaluate the skewness of the degree distribution [16]. Referring to the kurtosis of the normal distribution is 3, the larger the kurtosis, the more skewness of the degree distribution. When a graph is built, *SQL-G* on *Spark* explicitly specifies a partitioning strategy from *GraphX* partitioning strategies. According to [52], the Canonical Random works well with low degree graphs, and the 2D Edge partitioning works well with heavy-tail graphs. According to the kurtosis, *SQL-G* specifies Canonical Random partitioning for the road network CA and US and 2D Edge partitioning for other datasets.

Experimental Setup. We report our performance studies on an Amazon EC2 cluster consisting of 16 m4.2xlarge instances. Each instance has 8 vCPUs with Intel Xeon E5-2676 v3 (2.40 GHz) processor and 32 GB memory running RHEL 7.1 64bit. For each instance, the secondary storage is composed of 50 GB SSD and 500 GB HDD. Instances are connected by a 1 Gbps network.

6.1 *SQL-G*: Graph Workflow Versus *SQL* Workflow

There are two options to process graph analytics in *SQL-G*, namely, *SQL-G* Graph workflow (*SQL-G*/Graph) and *SQL-G SQL* workflow (*SQL-G*/*SQL*), where the former transforms MV-join, MM-join, and union by update in the enhanced *SQL* recursion into *GraphX* primitives when possible, and the latter does not use any *GraphX* primitives. The difference between the two workflows demonstrates the performance gain by using *GraphX* primitives in our *SQL* approach.

We report the testing of the 4 algorithms, WCC, SSSP, PR and LP on *SQL-G* under the *SQL-G*/*SQL* and *SQL-G*/Graph, respectively. For both, *SQL-G* takes the same input data/query and generates the same optimized recursive *SQL* logical plan. The difference is that in *SQL-G*/*SQL* a query is evaluated by the *Spark SQL* physical operators under the data-flow processing, whereas in *SQL-G*/Graph the optimization rules with graph support (Rule-1 - Rule-8) are applied in the recursive *SQL* logical plan to transform it to logical graph operators. Moreover, the logical graph operators are further transformed into *GraphX* APIs to be evaluated. For WCC, SSSP and PR, Rule-1, Rule-3, and Rule-4 are applied in order. For LP, Rule-1, Rule-7, Rule-3, and Rule-4 are applied in order.

In Table 3, we summarized the executing time and total shuffle size (read/write). Here the executing time starts from the query planning to the end of the evaluation, given that the source *RDD*s have been cached in memory. The total shuffle read is the total shuffle data one stage reads from local and remote executors, and the total shuffle write is the data written on disk which will be read by a shuffle in a future stage. We use the shuffle size to evaluate the communication cost. For the 4 algorithms, *SQL-G*/Graph reduces the shuffle size to over one order for the middle-size graph OK and the large graph AR. Under *SQL-G*/Graph, WCC and SSSP achieve 9-3.3 times speed-up compared with PR and LP achieve 4.2-1.2 times speed-up. This

TABLE 3
SQL Versus Graph Workflow

Algorithm	Dataset	16 Instances				9 Instance			
		SQL-G/SQL		*SQL-G*/Graph		*SQL-G/SQL*		*SQL-G*/Graph	
		Time (s)	Shuffle (GB)	Time (s)	Shuffle (GB)	Time (s)	Shuffle (GB)	Time (s)	Shuffle (GB)
WCC	CA	968	13.6/28.8	132	2.7/3.7	998	16.9/29.4	114	2.8/3.7
	OK	221	31.6/34.8	45	3.7/3.2	200	54.9/37.4	56	3.3/3.2
	AR	1,814	527.4/567.4	254	14.3/14.2	2,318	926.4/581.6	287	13.5/14.2
SSSP	CA	921	11.4/22.4	114	0.065/0.302	638	17.6/22.9	78	0.226/0.302
	OK	122	31.6/33.7	37	2.6/2.2	272	54.7/38	50	2.3/2.2
	AR	2,496	343.8/381.6	268	6.7/6.6	2,858	249.7/189.8	281	6.3/6.6
PR	CA	240	4.3/6.5	57	1.2/1.6	152	4.2/6.7	72	1.5/1.6
	OK	403	114.2/82.7	114	9/9.3	596	104.9/84.9	152	9/9.3
	AR	983	211.5/175.5	336	29.6/31.2	1,358	787.6/551.3	419	28.1/31.2
LP	CA	336	6.5/8.1	159	1.5/3	321	7.5/8.3	150	1.6/3
	OK	300	157.5/127.4	207	10/10.3	758	152.2/129.5	524	8.3/10.3
	AR	1200	317.3/252.6	966	43.7/45.3	1,778	454.5/294.6	2,071	41.9/45.4

is because, for WCC and SSSP, the $\oplus$ in $\mathcal{G}_{(\odot,\oplus)}$ has a strict partial order. Therefore, In the physical implementation of $\mathcal{G}_{(\odot,\oplus)}$, the vertex-centric function, sendMsg generated by *SQL-G* sends a message $vw_{src} \odot ew$ from source to the destination only if it can dominate vw_{dst} ($vw_{dst} \odot \mathbf{1}_{\odot}$).

The PR and LP, shown in Table 3, are computation intensive algorithms. Under *SQL-G*/Graph, the communication cost of every iteration is $O(m)$ where m is the number of edges. *SQL-G*/Graph benefits from vertex-centric computation as the graph is indexed with the vertex and edge partition routed.

As shown in Table 3, there are two points that need to be mentioned. First, the shuffle size is similar for the two algorithms, WCC and SSSP (AR is treated as an undirected graph for WCC) since *SQL-G/SQL* evaluates an MV-join and a union by update in each iteration. However, under *SQL-G*/Graph, SSSP has less shuffling since message passing and merge only occur locally from the source vertex. Second, even though CA and its shuffling data are much smaller than OK, the executing time of CA is much longer than OK since the large diameter leads to slow convergence, which is a drawback of the vertex-centric computation with *BSP* model. *SQL-G/SQL* will aggravate this influence since dataflow processing does not consider message reduction.

6.2 The *SQL-G* Scalability

To compare the performance of *SQL-G* scalability, we conduct the scalability test by varying the cluster size of 4, 8, 12, 16 EC2 m4.2xlarge instances. we partition a graph into 192 partitions (tasks), so each node in a cluster has the size 4, 8, 12, 16 executes 48, 24, 16, and 12 tasks, respectively. The cluster size starts from 4 since we have the out of memory exception on the clusters of 1 and 2 instances.

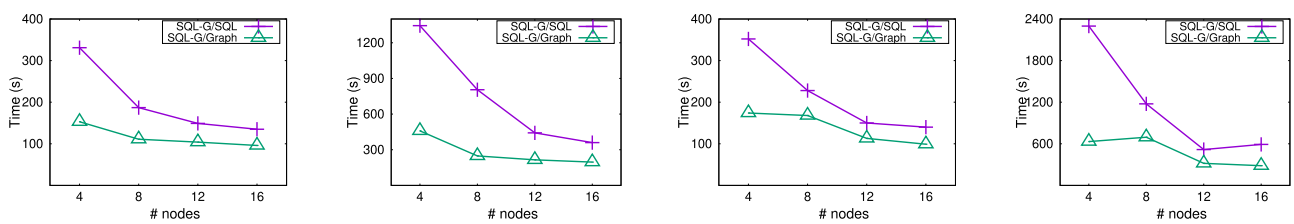
Figs. 11a, 11b, 11c, 11d show the 4 algorithms SSSP, PR, WCC and LP on dataset OK, respectively. In general, for both *SQL-G*/Graph and *SQL-G/SQL*, the running time reduces as the cluster size increases, while the improvement of performance becomes slower when the cluster is larger than 8 instances. The reason is that it is difficult to achieve linear scale-up while increasing the number of nodes, as a communication incentive framework. A larger cluster also leads to more overhead for garbage collection, job startup, etc. In our testing, this case occurs when we increase from 4 nodes to 8 as shown in Fig. 11d under *SQL-G*/Graph. For algorithm LP, communication cost is much larger than other algorithms since the message each vertex passes is an array instead of a signal value.

6.3 *SQL-G* Versus Other Systems

We compare *SQL-G*/Graph with other systems since *SQL-G*/Graph outperforms *SQL-G/SQL*. Below, *SQL-G* refers to *SQL-G*/Graph.

First, we confirm that *SQL-G* does not incur extra overhead into the execution of the recursive query, using COST [36] analysis. We compare our *SQL-G* and *GraphX* with a single-thread *PageRank* on a cluster of 8 machines, where each machine is equipped with 28 cores. For 20 iterations *PageRank*, the running time of this single-thread program takes 97.69 s over the Orkut dataset. The execution time of *SQL-G* and *GraphX* is shown in Table 4. *SQL-G* outperforms *GraphX*.

To further validate that, we compare the performance of SSSP, WCC, PR and LP over the 9 real graphs in our *SQL-G* and the 3 systems, *GraphX*, *GraphFrames*, and *BigDatalog* on an EC2 cluster of 16 instances, as shown in Fig. 12. The comparison aims at evaluating whether *SQL-G* can



(a) SSSP (b) PR (c) WCC (d) LP

Fig. 11. Scalability test.

TABLE 4
Running Time of 20 Iterations *PageRank* Over Orkut

# cores	16	32	64	128	224
SQL-G	105.90s	55.82s	35.04s	40.21s	34.63s
GraphX	117.48s	65.21s	38.94s	37.14s	37.69s

achieve the best in the *Spark* ecosystem. For all the systems, we report the total time of evaluation starting from loading the same data file from persistent storage (i.e., HDFS) to the end of the evaluation. In Fig. 12, 'Time Out' means the application cannot terminate in 3 hours and 'Out of Memory' means the application terminates with the out of memory exception.

Overall, *SQL-G* and *GraphX* reach a draw. Except for the costs of query plan transformation and graph construction, *SQL-G* does not incur additional costs during the query execution period. It is important to know that all the graph algorithms used are *GraphX* built-in algorithms, whereas *SQL-G* assumes that end users do not know anything about vertex-centric programming and the design paradigm of *GraphX*. Compared to *GraphX* built-in algorithms, *SQL-G* mainly differs from the following 4 points. 1) *SQL-G* requires collecting statistics of the graph. 2) *SQL-G* uses the code generated from *Spark SQL* expressions. 3) *SQL-G* uses the lower API aggregateMessages to conduct vertex-aggregate ($\mathcal{G}_{(\odot,\oplus)}$), which generates fewer shuffle data than that by the MapReduceTriples used in *GraphX* algorithms. 4) *Spark* 2.2 adds a graph checkpointing scheme in *GraphX* algorithms. For WCC, SSSP, and PR, *SQL-G* performs similar to *GraphX*. For LP, *SQL-G* outperforms *GraphX* on large graphs. A main reason is that the *GraphX* built-in LP algorithm performs vertex-centric computation on the Scala Map object, whereas *SQL-G* transforms the MM-join used for LP, by rules, into the optimized data structure SparseVector and corresponding optimized arithmetic operations. Therefore, the total shuffle size of *SQL-G* for LP is about 95-49 percent of that of *GraphX*.

As an alternative of using a high-level language *Datalog*, *BigDatalog* can process limited graph analytics. *BigDatalog* performs best for SSSP on large graphs, e.g., UK and TT, whereas it is time out for CA and US since we cannot control the maximum iteration number in the *Datalog* program supported by *BigDatalog*. The reason for *BigDatalog* to perform best for SSSP using UK and TT is because of its optimized monotonic aggregation and its single-job evaluation techniques, which are more effective on large graphs. However, as mentioned, *BigDatalog* cannot support PR and LP since *BigDatalog* cannot guarantee a fixpoint when non-monotonic aggregations are needed.

7 Related Works

Graph Processing Systems. Many graph systems have been extensively designed and developed in recent years. We review representative graph processing systems [54] from the perspective of the programming model. A well-known programming model is "Think Like a Vertex" (*TLAV*) [35], where users develop vertex-centric functions to specify how data is processed. The first published TLAV graph system is Google's *Pregel* [33], and *Giraph* [3], *GraphLab* [21] and *GraphX* [4] are three implementations based on the basic ideas of *Pregel*. Programs running on such TLAV systems can have one or multiple phases [35] in a distributed platform. *TLAV* systems adopt synchronous, asynchronous and hybrid models to access data using a communication scheme, which can be by message passing or shared-memory. There are natural extensions of the vertex-centric model, for example, block-centric model [55] and subgraph-centric model [50]. In such systems, graph algorithms access block/subgraph as a unit, and send messages from one block/subgraph to another, which is a trade-off between computation and communication. In addition, there are

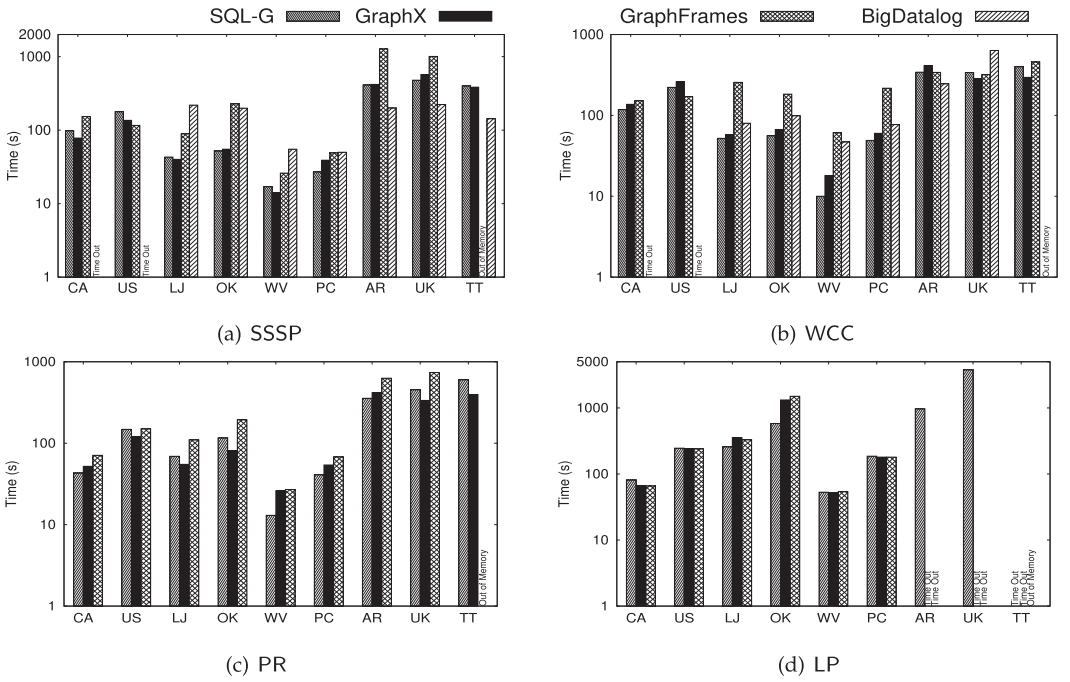


Fig. 12. Systems comparison.

matrix-based systems which adopt matrix-based programming model on single machine [64], [65] or distributed frameworks [26], [27]. The graph systems mentioned above provide collections of low-level APIs for users to implement graph algorithms. The users are required to fully understand the programming model, the APIs and the optimization techniques used in the underlying system, in order to achieve high efficiency. There are graph processing systems using high-level programming language such as *Datalog* [40]. A survey of early deductive database systems using *Datalog* is given in [40]. A new deductive application language system *DeALS* is developed to support graph queries [47], and the optimization of monotonic aggregations is further studied [46]. *SociaLite* [43] executes *Datalog* queries in parallel and distributed environments [44]. These systems support graph analytics, especially iterative graph analytics since *Datalog* has great expressive power for recursion. The efficient evaluation of *Datalog* is investigated. *Big-Datalog* is developed on the data flow processing system *Spark* [45].

Graph Query Languages. It is still an open question how to query graphs since there is no consensus on such a universal graph query language. A survey on graph query language for pattern matching queries is given in [53]. It discusses graph query languages for pattern matching including conjunctive query, regular path query and conjunctive regular path query. ,which covers conjunctive query (CQ), regular path query (RPQ), and CRPQ combining CQ and RPQ. The language *GraphQL* proposed in [23] is based 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. Transactional graph databases have their own query languages. For instance, the declarative, *SQL*-inspired query language *Cypher* [1], and functional language *Gremlin* [2] are integrated to specify graph patterns and graph traversals. And there are several new attempts focus on querying graphs on *RDBMS*. *PGQL* in [51] 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. *Portal* in [37] is a declarative query language to be used to query evolving graphs, based on temporal relational algebra. An *SQL*-based declarative query language *SciQL*, which supports graph analytics, is proposed in [61], [62] to perform array computation in *RDBMS*s. However, most of these graph query languages cannot support iterative graph analytics.

Graph Analytics by SQL. Several studies focus on supporting specific graph algorithms by *SQL*. *Aster-6* [48] takes an approach to invoke graph analytics functions from *SQL* based on [19], which handles a function as a table in *SQL* and executes the function by *MapReduce*. As shown in [48], to compute *PageRank*, *Aster-6* evaluates the query "select pagerank from PageRank()". Here, PageRank() is a graph analytics function which needs many inputs including a vertex table and an edge table. In querying *PageRank*, users need to use various graph-based methods to implement the PageRank() function. In other words, it requires users to have good knowledge of both *SQL* and vertex-centric programming. There are approaches to translate graph analytics by vertex-centric programming to *SQL* including *GRAIL* [17], *GraphiQL* [24], *Vertica* [25] and *GraphFrames* [15]. Here, *GRAIL* in [17] converts *UDF*s to *SQL* scripts by controlling the looping explicitly. *GraphiQL* in [24] compiles program manipulating graph table controlled by foreach loop to *SQL*. *Vertica* in [25] is a column-oriented relational database to be used as a platform for vertex-centric graph analysis. *GraphFrames* in [15] is a system built on *Spark* with an API mixing graph and relational queries. These approaches support graph analytics by utilizing *SQL* but do not treat graph analytics and those data analytics by *SQL* in a unified way. Users need to write programs and need to know a graph programming model well since different graph processing systems have their own ways to process graph analytics.

Srihari et al. in [49] propose an approach for mining dense subgraphs in *RDBMS*s. Passing et al. in [38] study how to process *PageRank* by extending the *SQL* recursive query by the lambda expression in main-memory *RDBMS*s. Gao et al. in [20] study how to process shortest path queries using the *SQL* window functions and the merge statement in *RDBMS*s. There are systems for evaluating path and pattern queries. *GQ-Fast* in [31] is an indexed and fragmented database that supports efficient *SQL* queries for graph analytics. *G-SQL* in [32] is designed as an *SQL* dialect for graph exploration, where multi-way joins are boosted by the underlying graph processing engine. *EmptyHead* in [6] is developed as a graph pattern engine to process graph patterns by parallel join processing.

8 Conclusion

In this paper, we integrate distributed graph processing system and *SQL* engine to support graph analytics and data analytics uniformly. We focus on how to efficiently process the enhanced recursive *SQL* queries by an underlying distributed graph processing system. Our approach only requires the knowledge of *SQL* and hides the distributed system programming paradigm and implementation details from users. We have implemented our *SQL-G* system on *Spark SQL* and *GraphX* using a strategy to push the processing inside the underlying systems to fully utilize the system resources to perform graph analytics. We enhanced *Spark SQL* by adding new logical operators to conduct graph analytics and convert them to *GraphX* operators. We have conducted extensive performance studies and confirm that *SQL-G* can achieve approaching or even higher efficiency for user-specified recursive *SQL* queries in comparison to the efficiency of built-in graph algorithms in the distributed graph processing systems.

Acknowledgments

This work was supported by the Research Grants Council of the Hong Kong SAR, China, No. 14203618 and No. 14202919.

References

[1] 2019. [Online]. Available: http://neo4j.com
[2] 2015. [Online]. Available: http://thinkaurelius.github.io/titan/
[3] 2019. [Online]. Available: http://giraph.apache.org
[4] 2018. [Online]. Available: http://spark.apache.org/graphx/
[5] DIMACS challenge 9 - shortest paths, 2006. [Online]. Available: http://www.dis.uniroma1.it/challenge9/
[6] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré, "EmptyHeaded: A relational engine for graph processing," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 431–446.

[7] C. C. Aggarwal, *Social Network Data Analytics*. Berlin, Germany: Springer, 2011.

[8] C. C. Aggarwal and H. Wang, *Managing and Mining Graph Data*. Berlin, Germany: Springer, 2010.

[9] M. Armbrust *et al.*, "Spark SQL: Relational data processing in spark," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1383–1394.

[10] F. Arni, K. Ong, S. Tsur, H. Wang, and C. Zaniolo, "The deductive database system LDL++," *Theory Practice Logic Program.*, vol. 3, no. 1, pp. 61–94, 2003.

[11] P. Barceló, "Querying graph databases," in *Proc. 32nd ACM SIGMOD-SIGACT-SIGAI Symp. Princ. Database Syst.*, 2013, pp. 175–188.

[12] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. 13th Int. Conf. World Wide Web*, 2004, pp. 595–602.

[13] U. Brandes and T. Erlebach, *Network Analysis: Methodological Foundations*. Berlin, Germany: Springer-Verlag, 2005.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3 ed. Cambridge, MA, USA: MIT Press, 2009.

[15] A. Dave, A. Jindal, L. E. Li, R. Xin, J. Gonzalez, and M. Zaharia, "GraphFrames: An integrated API for mixing graph and relational queries," in *Proc. 4th Int. Workshop Graph Data Manage. Experiences Syst.*, 2016, Art. no. 2.

[16] L. T. DeCarlo, "On the meaning and use of kurtosis," *Psychological Methods*, vol. 2, no. 3, pp. 292–307, 1997.

[17] J. Fan, A. Gerald, S. Raj, and J. M. Patel, "The case against specialized graph analytics engines," in *Proc. Biennial Conf. Innovative Data Syst. Res.*, 2015.

[18] F. Fouss, A. Pirotte, J. Renders, and M. Saerens, "Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 3, pp. 355–369, Mar. 2007.

[19] E. Friedman, P. M. Pawlowski, and J. Cieslewicz, "SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1402–1413, 2009.

[20] J. Gao, R. Jin, J. Zhou, J. X. Yu, X. Jiang, and T. Wang, "Relational approach for shortest path discovery over large graphs," *Proc. VLDB Endowment*, vol. 5, no. 4, pp. 358–369, 2011.

[21] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Conf. Operating Syst. Design Implementation*, 2012, pp. 17–30.

[22] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. 11th USENIX Conf. Operating Syst. Design Implementation*, 2014, pp. 599–613.

[23] H. He and A. K. Singh, "Graphs-at-a-time: Query language and access methods for graph databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 405–418.

[24] A. Jindal and S. Madden, "GRAPHiQL: A graph intuitive query language for relational databases," in *Proc. IEEE Int. Conf. Big Data*, 2014, pp. 441–450.

[25] A. Jindal, S. Madden, M. Castellanos, and M. Hsu, "Graph analytics using vertica relational database," in *Proc. IEEE Int. Conf. Big Data*, Santa Clara, CA, USA, Oct. 29–Nov. 1, 2015, pp. 1191–1200, doi: 10.1109/BigData.2015.7363873.

[26] U. Kang, H. Tong, J. Sun, C. Lin, and C. Faloutsos, "GBASE: A scalable and general graph management system," in *Proc. 17th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2011, pp. 1091–1099.

[27] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: Mining peta-scale graphs," *Knowl. Inf. Syst.*, vol. 27, no. 2, pp. 303–325, 2011.

[28] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. Philadelphia, PA, USA: SIAM, 2011.

[29] H. Kwak, C. Lee, H. Park, and S. B. Moon, "What is Twitter, a social network or a news media?" in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 591–600.

[30] J. Leskovec and A. Krevl, "SNAP datasets: Stanford large network dataset collection," Jun. 2014. [Online]. Available: http://snap.stanford.edu/data

[31] C. Lin, B. Mandel, Y. Papakonstantinou, and M. Springer, "Fast in-memory SQL analytics on typed graphs," *Proc. VLDB Endowment*, vol. 10, pp. 265–276, 2016.

[32] H. Ma, B. Shao, Y. Xiao, L. J. Chen, and H. Wang, "G-SQL: Fast query processing via graph exploration," *Proc. VLDB Endowment*, vol. 9, pp. 900–911, 2016.

[33] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.

[34] C. D. Manning *et al.*, *Introduction to Information Retrieval*, vol. 1. Cambridge, U.K.: Cambridge Univ. Press, 2008.

[35] 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.*, vol. 48, no. 2, 2015, Art. no. 25.

[36] F. McSherry, M. Isard, and D. G. Murray, "Scalability! But at what cost?" in *Proc. 15th USENIX Conf. Hot Topics Operating Syst.*, 2015, pp. 14–14.

[37] V. Z. Moffitt and J. Stoyanovich, "Towards a distributed infrastructure for evolving graph analytics," in *Proc. 25th Int. Conf. Companion World Wide Web*, 2016, pp. 843–848.

[38] L. Passing *et al.*, "SQL- and operator-centric data analytics in relational main-memory databases," in *Proc. 20th Int. Conf. Extending Database Technol.*, 2017, pp. 84–95.

[39] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Phys. Rev. E*, vol. 76, no. 3, 2007, Art. no. 036106.

[40] R. Ramakrishnan and J. D. Ullman, "A survey of deductive database systems," *J. Log. Program.*, vol. 23, no. 2, pp. 125–149, 1995.

[41] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. Das Sarma, "Finding connected components in Map-Reduce in logarithmic rounds," in *Proc. IEEE 29th Int. Conf. Data Eng.*, 2013, pp. 50–61.

[42] S. Salihoglu and J. Widom, "HelP: High-level primitives for large-scale graph processing," in *Proc. Workshop Graph Data Manage. Experiences Syst.*, 2014, pp. 1–6.

[43] J. Seo, S. Guo, and M. S. Lam, "SociaLite: Datalog extensions for efficient social network analysis," in *Proc. IEEE 29th Int. Conf. Data Eng.*, 2013, pp. 278–289.

[44] J. Seo, J. Park, J. Shin, and M. S. Lam, "Distributed socialite: A datalog-based language for large-scale graph analysis," *Proc. VLDB Endowment*, vol. 6, no. 14, pp. 1906–1917, 2013.

[45] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo, "Big data analytics with datalog queries on spark," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1135–1149.

[46] A. Shkapsky, M. Yang, and C. Zaniolo, "Optimizing recursive queries with monotonic aggregates in DeALS," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 867–878.

[47] A. Shkapsky, K. Zeng, and C. Zaniolo, "Graph queries in a next-generation datalog system," *Proc. VLDB Endowment*, vol. 6, no. 12, pp. 1258–1261, 2013.

[48] D. E. Simmen *et al.*, "Large-scale graph analytics in Aster 6: Bringing context to big data discovery," *Proc. VLDB Endowment*, vol. 7, no. 13, pp. 1405–1416, 2014.

[49] S. Srihari, S. Chandrashekar, and S. Parthasarathy, "A framework for SQL-based mining of large graphs on relational databases," in *Proc. 14th Pacific-Asia Conf. Advances Knowl. Discovery Data Mining*, 2010, pp. 160–167.

[50] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From "think like a vertex" to "think like a graph"," *Proc. VLDB Endowment*, vol. 7, no. 3, pp. 193–204, 2013.

[51] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi, "PGQL: A property graph query language," in *Proc. 4th Int. Workshop Graph Data Manage. Experiences Syst.*, 2016, Art. no. 7.

[52] S. Verma, L. M. Leslie, Y. Shin, and I. Gupta, "An experimental comparison of partitioning strategies in distributed graph processing," *Proc. VLDB Endowment*, vol. 10, no. 5, pp. 493–504, 2017.

[53] P. T. Wood, "Query languages for graph databases," *ACM SIGMOD Rec.*, vol. 41, no. 1, pp. 50–60, 2012.

[54] D. Yan, Y. Bu, Y. Tian, and A. Deshpande, "Big graph analytics platforms," *Found. Trends Databases*, vol. 7, no. 1/2, pp. 1–195, 2017.

[55] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Blogel: A block-centric framework for distributed computation on real-world graphs," *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 1981–1992, 2014.

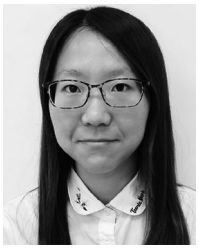
[56] D. Yan, Y. Tian, and J. Cheng, *Systems for Big Graph Analytics*. Berlin, Germany: Springer, 2017.

[57] P. S. Yu, J. Han, and C. Faloutsos, *Link Mining: Models, Algorithms, and Applications*. Berlin, Germany: Springer, 2010.

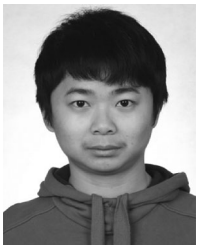
[58] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Design Implementation*, 2012, pp. 2–2.

[59] C. Zaniolo, N. Arni, and K. Ong, "Negation and aggregates in recursive rules: The LDL++ approach," in *Proc. Int. Conf. Deductive Object-Oriented Databases*, 1993, pp. 204–221.

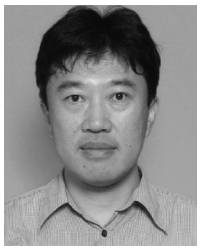
[60] C. Zaniolo *et al*., *Advanced Database Systems*. San Mateo, CA, USA: Morgan Kaufmann, 1997.
[61] Y. Zhang, M. Kersten, M. Ivanova, and N. Nes, "SciQL: Bridging the gap between science and relational DBMS," in *Proc. 15th Symp. Int. Database Eng. Appl.*, 2011, pp. 124–133.
[62] Y. Zhang, M. Kersten, and S. Manegold, "SciQL: Array data processing inside an RDBMS," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1049–1052.
[63] K. Zhao and J. X. Yu, "All-in-One: Graph processing in RDBMSs revisited," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 1165–1180.
[64] Y. Zhou, L. Liu, K. Lee, and Q. Zhang, "GraphTwist: Fast iterative graph computation with two-tier optimizations," *Proc. VLDB Endowment*, vol. 8, no. 11, pp. 1262–1273, 2015.
[65] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2015, pp. 375–386.



Kangfei Zhao received the BE degree from the University of Science and Technology of China, in 2014, and the PhD degree from the Department of Systems Engineering and Engineering Management, Chinese University of Hong Kong, under the supervision of Prof. Jeffrey Xu Yu, in 2019. Her research interests include graph data management and in-database machine learning.



Jiao Su received the BE degree from the School of Information, Renmin University of China, in 2015, and the PhD degree from the Department of Systems Engineering and Engineering Management, Chinese University of Hong Kong, Hong Kong. His research interests include graph algorithms and keyword search.



Jeffrey Xu Yu held teaching positions with the Institute of Information Sciences and Electronics, University of Tsukuba, Japan, and the Department of Computer Science, Australian National University, Australia. Currently, he is a professor with the Department of Systems Engineering and Engineering Management, Chinese University of Hong Kong, Hong Kong.

Hao Zhang received the BEng degree from the Computer School, Wuhan University, in 2017. He is working toward the PhD degree in system engineering and engineering management, Chinese University of Hong Kong, under the supervision of Prof. Jeffrey Xu Yu. His primary research interests include distributed graph analysis system, and distributed graph algorithm.