GRFusion: Graphs as First-Class Citizens in Main-Memory Relational Database Systems

Mohamed S. Hassan[1], Tatiana Kuznetsova[1], Hyun Chai Jeong[1]
Walid G. Aref[1], Mohammad Sadoghi[2]
[1]Purdue University, West Lafayette, IN [2]University of California, Davis, CA
{msaberab,tkuznets,jeong3,aref}@cs.purdue.edu,msadoghi@ucdavis.edu

ABSTRACT

The maturity of RDBMSs has motivated academia and industry to invest efforts in leveraging RDBMSs for graph processing, where efficiency is proven for vital graph queries. However, none of these efforts process graphs natively inside the RDBMS, which is particularly challenging due to the impedance mismatch between the relational and the graph models. In this demonstration, we present GRFusion, an in-memory relational database system, where graphs are managed as first-class citizens. GRFusion is realized inside VoltDB. The SQL and query engines of VoltDB are empowered to declaratively define graphs and execute cross-data-model query plans that consist of relational operators and newly-introduced graph operators. Using a social network and a real continental-sized road network covering the entire U.S., we demonstrate the functionality and the performance of GRFusion in evaluating queries that reference both relational tables and graphs seamlessly in the same query execution pipeline. GRFusion shows up to four orders-of-magnitude speedup in query-time w.r.t. state-of-the-art approaches.[1]

ACM Reference Format:
Mohamed S. Hassan[1], Tatiana Kuznetsova[1], Hyun Chai Jeong[1] Walid G. Aref[1], Mohammad Sadoghi[2]. 2018. GRFusion: Graphs as First-Class Citizens in Main-Memory Relational Database Systems . In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183713.3193541

1 INTRODUCTION

Graphs are ubiquitous in various application domains, e.g., social networks, road networks, biological networks, and communication networks. The data of these applications can be viewed as graphs, where the vertexes and the edges have relational attributes [15], or as traditional relational data with latent graph structures [17]. Applications would issue queries that reference the topology of the graphs along with the data associated with the vertexes and the edges or other data sources (e.g., relational tables). For instance, a user may be interested to find the shortest path over a road network while restricting the search to certain types of roads, e.g., avoiding toll roads. In an RDBMS, the filtering predicates can be expressed as relational predicates, and they may reference relational tables that have indirect relation with the queried graphs. We refer to these queries as graph-relational queries. Graph-relational queries have two main ingredients: 1) graph operations, e.g., shortest-path computation, and 2) relational predicates or relational sub-queries, e.g., selecting specific users from relational tables to find the nearest hospitals using shortest-path evaluation on top of a graph of a road-network. Moreover, many graphs have relational schemas that describe the data associated with their vertexes and edges. For instance, the popular STRING biological dataset [2] has a publicly available relational schema.[2] Associating a graph with a relational schema gives rise to many interesting graph-relational queries [9].

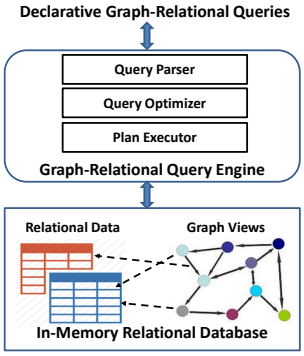

Figure 1: GRFusion's architecture allows the query engine to view/process data in the relational and the graph models.

As RDBMSs are pervasive and mature, various approaches for using an RDBMS to manage graph data have been proposed, e.g., SQLGraph [15], Grail [6]. Commercial systems, e.g., Oracle Graph and Aster [14], follow the architecture of processing graph-relational queries using different run-time systems, where the results are combined at the end. These approaches manage graphs as an application on top of an RDBMS without modifying the internals of the RDBMS. Consequently, these approaches are either limited to specific graph queries that can be translated into SQL (e.g., SQLGraph [15]), or they have performance overheads due to integrating results from two different execution engines. In this demonstration, we present GRFusion [9], a realization of a new approach where graphs are recognized as first-class citizens inside an RDBMS. GRFusion is implemented inside a centralized version of VoltDB [1, 4], the open-source implementation of the H-Store in-memory RDBMS [10].

[1]Extensive performance evaluation of GRFusion can be found in [9].

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3193541

[2]http://string-db.org/download/database.schema.v10.pdf

The main idea of GRFusion is to natively process graphs inside an RDBMS by combining the relational and the graph models under the same query engine. GRFusion separates the graph topology from the relational data associated with the vertexes and the edges, and introduces graph operators to process the graph topology inside the RDBMS, where the graph operators seamlessly co-exist with other relational operators in the same query execution pipeline (or QEP, for short). A graph topology in GRFusion is realized as a native graph structure, where each vertex or edge has pointers to the relational tuples describing their attributes. Hence, a graph topology in GRFusion can be viewed as a traversal index of the relational tuples of the vertexes and the edges. In short, GRFusion presents cross-data-model QEPs, where the inputs to the QEPs can be either relational data or native graph structures. Figure 1 illustrates the high-level architecture of GRFusion. First, the end-user provides a declarative statement to create graph views that are initialized from relational data. Second, the user is allowed to query the graph views as well as other relational tables or views in the same query. This demonstration presents how to empower the pervasive relational databases to support graph traversal queries natively and efficiently. Consequently, the relational-data owners can process vital graph queries through their RDBMS systems without the overhead of migrating their data to a different specialized graph system.

2 OVERVIEW OF GRFUSION

In GRFusion, graphs are assumed to be initially stored in relations. In the simplest case, a relational table may have a row for each vertex, and another table could have a row for each edge. Also, the vertexes/edges data can be obtained through a relational materialized view that joins or filters multiple tables. For flexibility, GRFusion provides the user with a declarative language to define and query graphs. A graph is defined in GRFusion by what we term *graph views*. A *graph view* identifies the relational sources that store the attributes of the vertexes and edges, namely, the *vertexes relational-source* and the *edges relational-source*, respectively. A *Graph view* defines a view of the relational data in the graph model, and materializes the graph topology in main-memory in native graph data structures, specifically as adjacency lists. The materialized graph topology has a *native graph representation* that holds pointers (e.g., tuple identifiers) to the relational data that describe the vertexes and the edges (see Figure 3). The main idea behind materializing the graph topology is to empower the relational database engine with the ability to realize complex graph algorithms. Thus, GRFusion helps fill the gap between the relational model and the massive body of research that assumes a graph model.

Once a graph view is defined, GRFusion allows the user to write pure graph queries, pure relational queries, or queries that mix both graph and relational operations. GRFusion's query engine views the relational data in either the relational model or the graph model according to the incoming query. In particular, the graph clauses in a query are mapped to graph operators in the query execution pipeline, where a graph operator accepts only graph representations as input. GRFusion allows the graph operators and the relational operators to co-exist in the same QEP, where the operator type determines the data model of viewing the data (i.e., graph views for the graph model, and relations for the relational model).

Users

uId	fName	lName	dob
1	Edy	Smith	09-25-1971
2	Jones	Parker	11-21-1980
3	Bill	Patrick	02-01-1976
.....	.....	.....	.....

Relationships

relId	uId1	uId2	startDate	isRelative
1	1	3	01-10-2009	true
2	2	3	12-31-2008	false
	.....	.....	.....	.....

Figure 2: A sample social-network in the relational model.

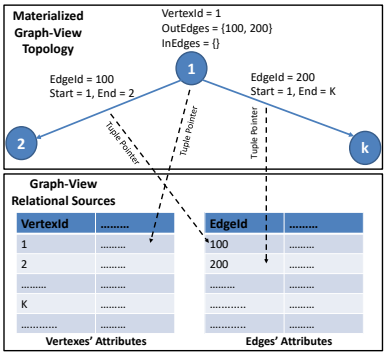

Figure 3: A graph view materializes the topology and holds pointers to the relational data of the vertexes and the edges.

3 CREATING GRAPH VIEWS

GRFusion has a declarative *Create Graph View* statement to create graph views initialized from relational data. The statement has four main objectives: (1) Identifying the name of the graph view to create, (2) Identifying and extracting the graph's set of vertexes from the underlying relational sources, (3) Identifying and extracting the graph's set of edges from the underlying relational sources, and (4) Materializing a native graph data structure in memory that reflects the graph topology based on adjacency-list structures. Notice that graph traversal operations can be performed efficiently over this native graph representation that is linked back to the corresponding relational data tuples that describe it. Notice further that the relational source can either be a table or a materialized relational-view because the graph data attributes for the edges and/or the vertexes can be constructed from multiple data sources.

Figure 2 illustrates how a graph view is created in GRFusion. Assume that the data of a social network is stored in the relational tables as in the figure. Tables *Users* and *Relationships* represent the vertexes and the edges of the social network, respectively. Each vertex or edge has an identifier in the relational tables, and this is the only required attribute when creating graph views in GRFusion. To illustrate, consider Listing 1, where the relational sources of Figure 2 are used to create a graph view, namely the *SocialNetwork* graph view. A vertex in the *SocialNetwork* graph has its Id from Users.uId and has the two attributes lName and birthdate that get their values from Users.lName and Users.dob, respectively. Similarly, Table *Relationships* defines the edges of the *SocialNetwork* graph, where the edge Id comes from Relationships.relId, the endpoints come from Relationships.uId1 and Relationships.uId2, and the two edge attributes *sDate*, *relative* refer to Attributes *startDate*, *isRelative* of Table *Relationships*, respectively. For the graph view defined by

the *Create Graph View* statement, if the set of vertexes is V, and the set of edges is E, then, the endpoints of an edge in E are constrained to be included in V.

Listing 1: A Social Network Graph View Example

```
CREATE UNDIRECTED GRAPH VIEW SocialNetwork
VERTEXES(ID = uId, lstName = lName,
    ↪ birthdate = dob) FROM Users
EDGES (ID = relId, FROM = uId1, TO = uId2,
    ↪ sDate = startDate, relative =
    ↪ isRelative) FROM Relationships
```

4 THE PATHS QUERY CONSTRUCT

As graph traversal queries form a massive body of graph queries (e.g., reachability and shortest path queries [5, 7, 13, 16]), GRFusion extends the SQL language of VoltDB to declaratively find paths in graph views. The reason for extending SQL instead of adopting a graph language (e.g., Cypher [12]) is to allow an arbitrary SQL query to reference graphs (e.g., join tables with paths as in Figure 4). GRFusion introduces the *PATHS* construct to query its graph views. For a graph view, say *GV*, GRFusion recognizes *GV.PATHS* in the From clause of a select statement. Conceptually, this allows GRFusion to traverse and retrieve paths from *GV* that satisfy a path criteria (e.g., predicates on the attributes of the edges forming the path). In addition to *GV.PATHS*, GRFusion recognizes *GV.VERTEXES*, and *GV.EDGES*, to reference the vertexes and the edges of *GV*, respectively. We focus on the *GV.PATHS* construct as the other constructs are straightforward.

GRFusion models a path as an ordered list of edges, where each edge has a start and end vertexes. The attributes of the edges and the vertexes of a path, say *PS*, can be indexed and referenced by relational predicates. Each vertex in Path *PS* has two additional integral attributes, namely *FanIn* and *FanOut*. Also, Path *PS* allows accessing to some path-specific properties, e.g., *PS.StartVertexId* and *PS.Length* refer to the identifier of the start vertex and the length of Path *PS*, respectively. To illustrate how paths can be queried in GRFusion, consider Query Q_p in Listing 2. The From clause of Q_p specifies that the paths are traversed from the SocialNetwork graph view, where the *vertexes relational-source* of the SocialNetwork graph is Relation $Users$. The query displays the last names of the friends of friends of all the users with Job = 'Lawyer'. Conceptually, Q_p is evaluated by selecting the sub-graph, say G_{sub}, containing edges with start dates after '1/1/2000'. Using Sub-graph G_{sub}, GRFusion explores paths consisting of two edges that originate from the vertexes corresponding to lawyers in the social network. Notice that Listing 2 could use *SocialNetwork.VERTEXES* instead of *Users*. However, Listing 2 uses the *Users* relation to show how relational tables can be joined with the paths of a graph view.

Listing 2: Friends-of-Friends Path Query Q_p

```
SELECT PS.EndVertex.lstName
FROM Users U, SocialNetwork.Paths PS
WHERE U.Job = 'Lawyer' AND PS.StartVertex.Id
    ↪  = U.uId AND PS.Length = 2 AND PS.
    ↪ Edges[0..*].StartDate > '1/1/2000'
```

Listing 3 presents a reachability query Q_r that queries a protein-interaction network represented by the BioNetwork graph view, and checks if *Protein X* interacts directly (i.e., by an edge) or indirectly (i.e., by a path) with *Protein Y* through either a covalent or stable interaction types. *PS.PathString* corresponds to the string representation of Path *PS*. Notice that many paths can exist between the vertexes corresponding to the specified proteins. So, Query Q_r uses the *LIMIT 1* clause because retrieving one path is sufficient to decide reachability.

Listing 3: Reachability Query Q_r

```
SELECT PS.PathString
FROM Proteins Pr1, Proteins Pr2, BioNetwork.
    ↪ Paths PS
WHERE Pr1.Name = 'Protein X' AND Pr2.Name =
    ↪ 'Protein Y' AND PS.StartVertex.Id =
    ↪ Pr1.Id AND PS.EndVertex.Id = Pr2.Id
    ↪ AND PS.Edges[0..*].Type IN ('covalent
    ↪ ', 'stable')
LIMIT 1
```

5 QUERY PROCESSING IN GRFUSION

GRFusion defines three operators to evaluate the graph constructs of graph-relational queries. In particular, GRFusion defines the *VertexScan*, *EdgeScan*, and *PathScan* operators that iterate over a graph view's vertexes, edges, and paths, respectively. The PathScan operator is a lazy operator following the iterator model [8] to avoid eager generation of paths that might not be required by parent operators. The reason for this design decision is that many queries (e.g, reachability) limit the number of retrieved paths, and consequently generating all/multiple paths may be expensive and unnecessary.

In GRFusion, the *PathScan* operator is responsible for traversing a graph view to construct simple paths identified by a graph query. *PathScan* is a logical operator that has three physical operators with three corresponding graph-traversal algorithms. All the physical operators explore a traversed vertex only once to avoid loops. In particular, a logical *PathScan* operator is mapped into *DFScan*, *BFScan*, or *SPScan*, corresponding to depth-first search, breadth-first search, or shortest-path search physical operators, respectively. The mapping is based on a query-hint.

As a logical operation, the paths-discovery process in GRFusion starts from a set of start vertexes. These start vertexes are either stated explicitly in the query (e.g., PS.StartVertex.Id = Value) or are generated by some operators during query evaluation (e.g., PS.StartVertex.Id = VS.Id as in Listing 2). In the latter scenario, the start vertexes selected by some operators (e.g., TableScan, relational sub-query) are used to probe the *PathScan* traversal operator, i.e., the PathScan operator does not materialize all the paths of the scanned graph but rather explores the graph, and generates paths on the fly. Hence, the paths in GRFusion are not eagerly materialized by a PathScan operator, rather they are lazily generated. If the start vertexes of a path selection are not defined, all the vertexes of the corresponding graph view will be used as starting vertexes. To illustrate how paths are explored in GRFusion, consider Query Q_p in Listing 2. Q_p explicitly states that the path discovery process starts from the vertexes corresponding to lawyers in the social network. Figure 4 gives the query evaluation pipeline QEP_p that

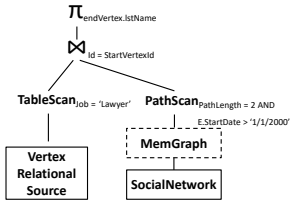

Figure 4: GRFusion joins a relational table with a graph-view PathScan operator for Query Q_p. The PathScan operator starts a path-discovery process by being probed by the start-vertex identifier from the outer table.

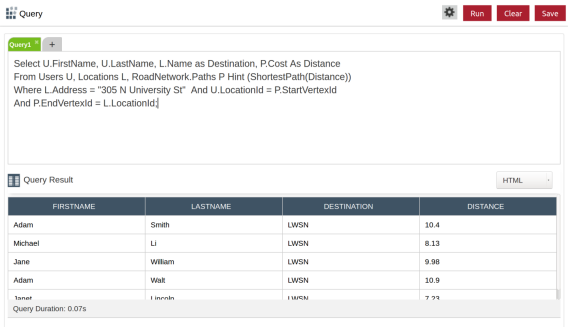

Figure 5: The Query Web-UI of GRFusion.

evaluates Query Q_p, where *MemGraph* refers to the singleton materialized graph structure of the graph view. In particular, Q_p starts the traversal process from each qualified vertex. Notice that the qualified vertexes are retrieved using a relational operator (e.g., by a TableScan or IndexScan operators) in Figure 4. The reason is that using a relational access method with filtering predicates on the *vertexes relational-source* is more efficient than using the tuple pointers in the graph view to filter all the vertexes on the fly. Because of the seamless integration of the relational and graph models in GRFusion, this optimization alternative is feasible. While traversing the graph view, only the edges with start dates after '1/1/2000' are considered. Also, QEP_p explores paths of length two only (i.e., consisting of two edges) that originate from a given start vertex. As an effective optimization, GRFusion pushes predicates, e.g., path-length predicates, inside the PathScan operator so that the predicates are considered during the traversal process. This optimization allows GRFusion to apply early pruning of paths, and to reduce the size of the intermediate results flowing through the query pipeline. Consequently, the performance of the query evaluation process is boosted w.r.t. the processing time as well as the temporary memory used for the intermediate results [9].

6 DEMO SCENARIO

We will demonstrate GRFusion by showing its functionality in defining and querying directed and undirected graphs inside VoltDB. We will also show the performance gains due to representing and processing graphs natively inside a relational engine. In the demonstration, we will show the functionality and the performance of GRFusion using a location-based social network [11], and the Tiger road-network [3]. The performance gains will be demonstrated by showing different plans and execution times for the same queries. In particular, we will compare the performance when evaluating reachability and shortest path queries using the hybrid query-plans of GRFusion consisting of relational and graph operators, and we will compare to using only relational operators.

The audience will interact with GRFusion through a Web interface (Figure 5), where they can issue graph-relational queries and observe query results and the execution times. The system will start with a database of relational tables that include tables with vertexes and edges' data. The audience will be able to create and observe directed and undirected graph views initialized from existing relational tables. Then, the audience will be introduced to the extended SQL of GRFusion that allows querying graphs (e.g., reachability, shortest paths) with relational predicates to filter out vertexes/edges on which the queries are evaluated. Moreover, the audience will be guided to write graph-relational queries, where graph views and relational tables are referenced in the same query. For example, the audience will see how a relational table, say R, can be joined with a graph operator discovering the paths of a graph, say G, where the qualified tuples from R are used to probe the paths of G. We will also demonstrate that the paths of G are never materialized beforehand, and how a join operator can have a graph view as *the inner* operand to start a path discovery process (i.e., probing). We will also show the execution plans of the queries that VoltDB provides and how graph operators and relational operators seamlessly co-exist in the same query execution pipeline.

REFERENCES

[1] [n. d.]. https://github.com/VoltDB/voltdb/.
[2] [n. d.]. http://string-db.org/.
[3] [n. d.]. https://www.census.gov/geo/maps-data/data/tiger.html.
[4] [n. d.]. https://www.voltdb.com/.
[5] E. W. Dijkstra. 1959. A note on two problems in connection with graphs. *Numerical Mathematics* 1 (1959), 269–271.
[6] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. [n. d.]. The Case Against Specialized Graph Analytics Engines. In *CIDR'15*.
[7] Jun Gao, Ruoming Jin, Jiashuai Zhou, Jeffrey Xu Yu, Xiao Jiang, and Tengjiao Wang. 2011. Relational Approach for Shortest Path Discovery over Large Graphs. *Proc. VLDB Endow.* 5, 4 (Dec. 2011), 358–369.
[8] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25, 2 (June 1993), 73–169.
[9] Mohamed S. Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G. Aref, and Mohammad Sadoghi. [n. d.]. Extending In-Memory Relational Database Engines with Native Graph Support, (Full Paper). In *EDBT'18*.
[10] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. [n. d.]. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* ([n. d.]).
[11] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).
[12] Onofrio Panzarino. 2014. *Learning Cypher.* Packt Publishing.
[13] Mohamed Sarwat, Sameh Elnikety, Yuxiong He, and Gabriel Kliot. [n. d.]. Horton: Online Query Execution Engine for Large Distributed Graphs. In *ICDE '12*.
[14] David Simmen, Karl Schnaitter, Jeff Davis, Yingjie He, Sangeet Lohariwala, Ajay Mysore, Vinayak Shenoi, Mingfeng Tan, and Yu Xiao. 2014. Large-scale Graph Analytics in Aster 6: Bringing Context to Big Data Discovery. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1405–1416.
[15] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. [n. d.]. SQLGraph: An Efficient Relational-Based Property Graph Store. In *SIGMOD'15*. 1887–1901.
[16] Jeppe Rishede Thomsen, Man Lung Yiu, and Christian S. Jensen. [n. d.]. Effective Caching of Shortest Paths for Location-based Services. In *SIGMOD '12*.
[17] Konstantinos Xirogiannopoulos, Udayan Khurana, and Amol Deshpande. 2015. GraphGen: Exploring Interesting Graphs in Relational Data. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 2032–2035.