Extending In-Memory Relational Database Engines with Native Graph Support

Mohamed S. Hassan
Purdue University
West Lafayette, IN
msaberab@cs.purdue.edu

Tatiana Kuznetsova
Purdue University
West Lafayette, IN
tkuznets@cs.purdue.edu

Hyun Chai Jeong
Purdue University
West Lafayette, IN
jeong3@cs.purdue.edu

Walid G. Aref
Purdue University
West Lafayette, IN
aref@cs.purdue.edu

Mohammad Sadoghi
University of California
Davis, CA
msadoghi@ucdavis.edu

ABSTRACT

The plethora of graphs and relational data give rise to many interesting graph-relational queries in various domains, e.g., finding related proteins retrieved by a relational subquery in a biological network. The maturity of RDBMSs 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 paper, we propose to manage graphs as first-class citizens inside the relational engine. We realize our approach inside *VoltDB* [6], an open-source in-memory relational database, and name this realization GRFusion. The SQL and query engine of GRFusion are empowered to declaratively define graphs and execute cross-data-model query plans acting on graphs and relations, resulting in up to four orders-of-magnitude in query-time speedup w.r.t. state-of-the-art approaches.

1 INTRODUCTION

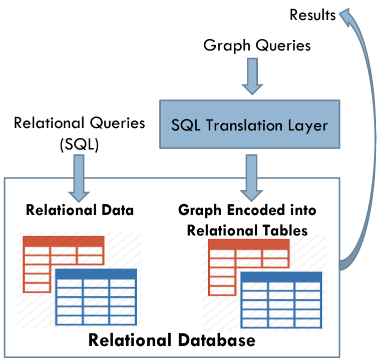
Graphs are ubiquitous in various application domains, e.g., social networks, road networks, biological networks, and communication networks [3, 8, 9, 12]. The data of these applications can be viewed as graphs, where the vertexes and the edges have relational attributes [46], or as traditional relational data with latent graph structures [51]. Applications would issue queries that consult the topology of the graphs along with the data associated with the vertexes and the edges or other data sources (e.g., relational tables in an RDBMS). 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 (or *G+R* queries, for short). *G+R* queries have two main ingredients: 1) graph operations, e.g., shortest-path computation, and 2) relational predicates or relational sub-queries. For example, selecting specific users from relational tables to find the nearest hospitals using shortest-path evaluation on top of a road-network.

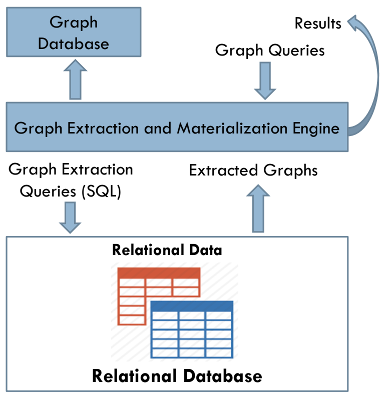
As RDBMSs are pervasive and mature, various approaches for using an RDBMS to manage graph data have been proposed, e.g., Grail [25] and Aster [45]. The literature has two main approaches that share the idea of building an application on top of an RDBMS to support graphs without modifying the internals of the RDBMS. We refer to these approaches as *Native Relational-Core* and *Native Graph-Core*. In this paper, we propose and investigate a hybrid approach that we term *Native G+R Core* that exploits the strengths of the former two approaches, and we realize our approach inside VoltDB [7, 10].

The *Native Relational-Core* approach (e.g., as in SQLGraph [46] and Grail [25]) embeds a graph inside of relational tables of specific schema. Then, an application on top of the RDBMS is built to translate specific types of graph queries into SQL statements for the RDBMS to execute. For example, Grail can translate shortest-path queries to procedural SQL [25], while SQLGraph translates Gremlin queries with some restrictions [5] into SQL queries [46]. Figure 1(a) illustrates the general architecture of the *Native Relational-Core* approach. Although many graph queries and algorithms are hard to translate into SQL statements, tools can be developed to automate the translation. However, the main issue of the *Native Relational-Core* approach is that the graph operations are evaluated by a sequence of relational operations (e.g., self-joins) that may be more expensive than traversing a native graph representation. Moreover, the *Native Relational-Core* approach does not guarantee an easy-to-comprehend relational schema of the embedded graphs in an RDBMS, e.g., the storage-optimized relational schema generated automatically by SQLGraph is hard for users to understand and write ad-hoc graph-relational queries [46].

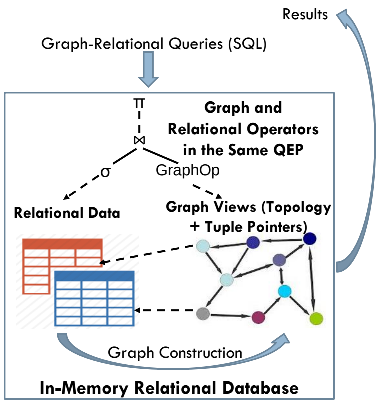
The second approach, namely *Native Graph-Core* (e.g., as in Ringo [38], GraphGen [51, 52]), assumes that graphs are already stored in an RDBMS, where an application on top of the RDBMS is built to extract these graphs to analyze them outside the realm of the RDBMS. This approach follows the same philosophy as that of specialized graph databases, where an RDBMS has nothing to do with query execution. Figure 1(b) illustrates the general architecture of the *Native Graph-Core* approach. Notice that a graph in the *Native Graph-Core* requires re-extraction if the relational tables storing the graph in the RDBMS are updated. Moreover, users cannot issue declarative graph-relational queries that reference both the extracted graphs and any other relational data in the RDBMS. One solution to allow graph-relational queries in the *Native Graph-Core* approach is to build another layer that queries both the RDBMS and the extracted graph. This solution is similar to that of Teradata Aster [45], where a data movement fabric and two different query executors (i.e., a relational executor and a graph executor) are used in processing graph-relational queries. However, integrating the results from the graph and the relational executors imposes additional overhead. In summary, the



(a) *Native Relational-Core*



(b) *Native Graph-Core*


Results
Graph-Relational Queries (SQL)
Graph and Relational Operators in the Same QEP
GraphOp
Relational Data
Graph Views (Topology + Tuple Pointers)
Graph Construction
In-Memory Relational Database

(c) *Native G+R Core*

Figure 1: Various approaches for leveraging relational databases in support of graph processing.

	Native Relational-Core	*Native Graph-Core*	*Native G+R Core*
Hybrid QEPs	✗	✗	✓
Native Graph Processing	✗	✓	✓
No Query-Translation Overhead	✗	✓	✓
No Graph Reconstruction/Re-embedding on Updates	✗	✗	✓

Table 1: Contrasting various approaches for graph support in RDBMSs.

Native Relational-Core and the *Native Graph-Core* approaches use a vanilla RDBMS, where graphs are not natively recognized by the RDBMS. However, if the necessary layers of the RDBMS are modified to manage graphs as first-class citizens, processing and managing graphs will be more efficient.

In this paper, we investigate a third approach, namely *Native G+R Core*, where graphs are recognized as first-class citizens inside an RDBMS. We address the impedance mismatch between the graph and the relational model, and we realize the *Native G+R Core* approach in a centralized version of VoltDB [7, 10], the open-source implementation of the H-Store in-memory relational DBMS [32]. In-memory data management witnessed early academic and industrial contributions, where the current affordability of large main-memory hardware motivated several and diverse research efforts [14, 15, 23, 30, 33, 35, 36, 38, 39, 44, 49–51, 53].

We refer to our realization of this approach as GRFusion. The main idea of GRFusion is to natively process graphs inside an RDBMS by combining the *Native Relational-Core* and the *Native Graph-Core* approaches under the same umbrella. GRFusion realizes this idea by separating the graph topology from the relational data associated with the vertexes and the edges, and by proposing 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(c) illustrates the general idea of the *Native G+R Core* approach. First, the end-user provides a declarative statement to create graph views that are initialized from relational data, where a graph view is materialized as a new database object. Second, the user is allowed to query the graph views as well as other relational tables or views in the same query. Table 1 contrasts the *Native Relational-Core*, *Native Graph-Core*, and *Native G+R Core* approaches. The objective of this paper is not to replace the specialized graph systems. However, the main objective is to empower the pervasive relational databases to support graph traversal queries natively and efficiently. Consequently, the relational-data owners can process important class of graph queries through their RDBMS systems without the cost and the overhead of migrating their data and manage it in a separate graph system. The contributions of this paper are as follows:

- Introducing graphs as native objects inside a relational database system, namely VoltDB (Section 3), where online graph updates are supported (Section 3.3).
- Allowing users to seamlessly query and operate on graphs and relations simultaneously and declaratively without leaving the realm of the relational database system (Section 4).
- Introducing graph operators for graph traversals (Section 5.1), and showing their ability to seamlessly co-exist with the relational operators to construct cross-data-model query execution pipelines (Section 5.2).
- Addressing the impedance mismatch between the graph model and the relational model (Section 5.3).
- Conducting an extensive performance study of GRFusion w.r.t. state-of-the-art systems, and reasoning about the benefits of processing graphs in a graph-native representation inside an RDBMS. We compare to SQLGraph, Grail, Neo4j, and Titan, where GRFusion achieves up to four orders-of-magnitude query-time speedup (Section 7).

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 may have a row for each edge. Also, the vertexes or the edges data can be obtained through a relational materialized view that joins or filters multiple relational tables. To allow flexibility, GRFusion provides the user with a declarative language to define and query graphs (see Figure 2). A graph is defined in GRFusion by what we term *graph views*. A *graph view* identifies the relational tables or the relational views that store the attributes of the vertexes and edges, namely, the *vertexes relational-source*, and the *edges relational-source*, respectively. *Graph views* define a view of the relational data in the graph model and materializes the graph topology in main-memory in native graph data structures. 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. 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. Listing 1 shows how a graph view is created in GRFusion from the relational sources of Figure 3, which is detailed in Section 3.1.

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 or the graph model according to the incoming query. In particular, the graph clauses in a query are mapped to graph operators in the QEP, 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).

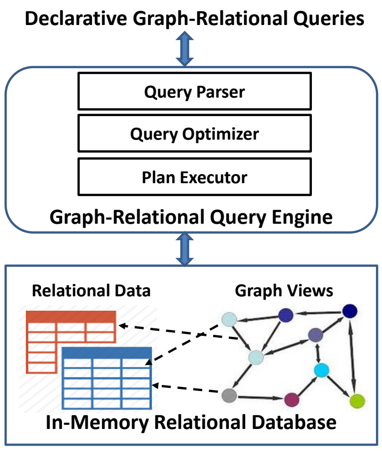

Declarative Graph-Relational Queries
Query Parser
Query Optimizer
Plan Executor
Graph-Relational Query Engine
Relational Data
Graph Views
In-Memory Relational Database

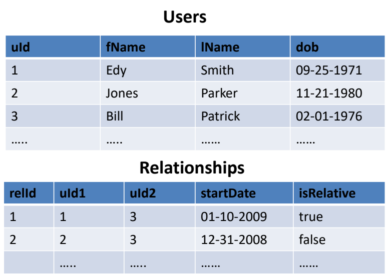
Figure 2: GRFusion's architecture allows the query engine to process data in both the relational and the graph models.

3 GRAPHS AS DATABASE OBJECTS

As users can create tables in relational databases, they can also create materialized graph views in GRFusion as database objects. A graph view is created once as a singleton object, and can be referenced by multiple users and queries. In Section 3.1, we highlight how graph views are defined declaratively in GRFusion. Section 3.2 illustrates how the topology of a graph in GRFusion is decoupled from the graph data, and how they can be inter-linked. Because dynamic graphs are essential in many applications, the support for graph updates is addressed in Section 3.3.

3.1 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


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 3: A sample social-network in the relational model.

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 and 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.

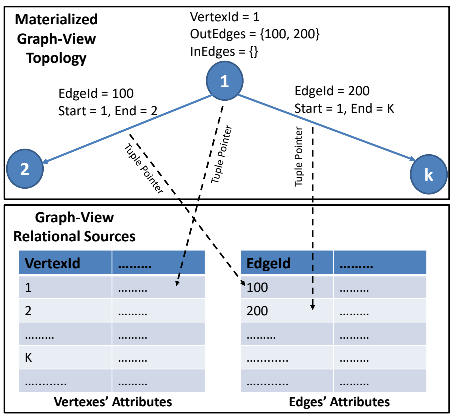
Figure 3 illustrates how a graph view is created in GRFusion. Assume that the data of a social network is stored in the relational tables as in the figure. Tables *Users* and *Relationships* represent the vertexes and the edges of the social network, respectively. Each vertex or edge has an identifier in the relational tables. To illustrate, consider Listing 1 that shows an example of creating a graph view, namely the *SocialNetwork* graph view, in GRFusion from the relational sources in Figure 3. 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
```

3.2 Decoupling the Graph Topology and the Graph Data

The *Create Graph View* statement updates the system catalog of GRFusion to store the definition of the graph view. Creating a graph view results in the materialization of the graph topology as a native graph structure in the main-memory managed by GRFusion (as a singleton object that multiple users and queries can reference). However, the attributes of the vertexes and the edges stored in the relational sources are not replicated in the native graph structure, and main-memory tuple pointers are used to link



VertexId	....
1	....
2	....
....	....
K	....
....	....

EdgeId	....
100	....
200	....
....	....
....	....
....	....

Figure 4: A graph view materializes the topology and holds pointers to the relational data of the vertexes and the edges.

the graph topology to the relational sources. To illustrate, Figure 4 demonstrates how the graph topology is separated from the graph data (i.e., the relational attributes of the vertexes and the edges). As in Figure 4, each vertex or edge has a main-memory tuple pointer that points to the corresponding relational tuple storing the attributes of this vertex or edge. Notice that the design of GRFusion allows a vertex or edge in a graph topology to store multiple tuple-pointers if the relational sources are vertically partitioned (e.g., to support semistructured RDF data, where not all the vertexes or edges share the same set of attributes). Without loss of generality, we assume a single tuple pointer per vertex or edge as the focus is to explore the benefits of empowering an RDBMS with native graph-processing.

The graph topology follows the graph model, where the topology is represented physically as a graph data-structure based on adjacency-lists. The key idea behind this native graph representation is to allow for the efficient execution of graph traversals, where relational joins can be mitigated when traversing a graph. The reason is that materializing the topology of a graph view can be thought of as a traversal index, where each vertex, say V, is associated with the identifiers of both the outgoing edges and the incoming edges of V. Given a graph view, say GV, its topology can be constructed using a single pass over the relational sources defining the vertexes and the edges of GV.

Notice that there is a bi-directional linkage between the graph topology and the graph's corresponding relational data. To illustrate, let T be a relational tuple containing the attribute values of Vertex V. Using the VertexId attribute of T, GRFusion can locate Vertex V in the graph representation in $O(1)$ time using the hash map of the native graph structure. Also, using the tuple pointer associated with Vertex V in the graph data-structure, Tuple T can be located in $O(1)$ time. The benefit of separating the graph topology from the graph data is two-fold. First, the size of the graph view is not affected by the size of the graph data that can be very large in some cases. Second, the attributes of the vertexes and the edges in the relational sources can be easily updated without affecting the native graph representation.

3.3 Graph Updates

GRFusion supports *serializable* graph updates that affect the topology or the attributes stored in the relational sources. The topology is affected only when vertexes/edges are added or deleted. GRFusion relies on the design and the implementation of VoltDB to maintain pointers to the relational tuples on memory reallocations.

3.3.1 Graph-Data Updates. Updating the attribute data of an edge or vertex is straightforward as the attributes are stored in relations outside the native graph representation. Hence, these relational attributes can be updated directly. However, updating the *VertexId* and the *EdgeId* attributes need special handling because these attributes are used for navigating from the relational store to the native graph structure (e.g., to probe path-traversal operators in a QEP as in Section 5). Although updating the identifiers are not common, GRFusion maintains the consistency of the identifiers in the graph representation when updating their corresponding attributes in the relational sources. Also, GRFusion maintains the referential integrity of the *edges relational-source* when updating a vertex identifier in the *vertexes relational-source*.

3.3.2 Graph-Topology Updates. GRFusion allows topological updates when the relational sources are either relational tables or a relational views selecting from a single table. GRFusion associates each relational source, say R, with the identifiers of the graph views that reference R. When inserting a new tuple into R, the transaction of the insertion statement updates the graph-view topology as part of the transaction (i.e., adding a new vertex or adding a new edge in the graph representation). Similarly, when deleting a vertex or edge, the deletion statement detects the graph views associated with R and updates the affected graph views accordingly as part of the deletion transaction. For example, if R is an *edges relational-source* for a graph view, say GV, the edge in GV corresponding to a deleted tuple is removed from GV.

4 THE PATHS QUERY CONSTRUCT

As graph traversal queries form a massive body of graph queries (e.g., reachability and shortest path queries [19, 24, 26, 42, 43, 47]), GRFusion extends the SQL language to declaratively find paths in graph views. 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 (as it is treated conceptually as a set of paths). Conceptually, this allows GRFusion to traverse and retrieve simple 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 edges and the vertexes of a path, say *PS*, can be indexed and referenced by relational predicates as follows:

- **PS.Edges[StartIndex..EndIndex].EdgeAttribute**: References an attribute of the edges starting from $StartIndex$ until $EndIndex$. A value of '*' for the $EndVertex$ placeholder indicates that all the edges starting from $StartIndex$ should satisfy the relational predicate.
- **PS.Vertexes[StartIndex..EndIndex].VertexAttribute**: References an attribute of the vertexes starting from $StartIndex$ until $EndIndex$. A value of '*' for the $EndVertex$ placeholder indicates that all the vertexes starting from $StartIndex$ should satisfy the relational predicate.

Observe that the aforementioned *EdgeAttribute*, and the *VerterxAttribute* placeholders can refer to any attribute of the edges or the vertexes that have been defined at the time of creating Graph-view GV. In addition, 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 being 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. Notice that the details of the extended query language of GRFusion are not the main focus of this paper. However, we provide sample code snippets that are relevant to illustrating the evaluation of the graph-relational queries supported by GRFusion.

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 on 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
```

In addition to the ability of referencing the attributes of the edges or vertexes forming a path, say *PS*, GRFusion allows aggregation functions on the attributes of the vertexes or the edges of *PS*. The aggregate functions on the attributes of paths have the same usage and constraints as those on relational attributes. For example, if the edges of *PS* have an attribute, say $Weight$, a query can compute the sum of the weight values across all the edges of *PS*, i.e., $sum(PS.Edges.Weight)$ can appear in the select-clause of a query to compute the sum of the weights associated with the edges of Path *PS*.

The PATHS construct can also retrieve sub-graphs based on specific patterns (e.g., the topology of the sub-graph, attributes of the vertexes/edges of the subgraph). For instance, finding triangular structures with specific edge properties, and counting these triangles are important primitives for Machine-Learning, e.g., [48], where a triangle structure can be viewed as a loop of three edges. Listing 4 presents Query Q_t that counts the number of triangles, where the edges have specific values for their *Label* attribute. Notice the use of the *Path.Length* property, where it is necessary to retrieve only triangles (as the sub-graph of interest has only three edges).

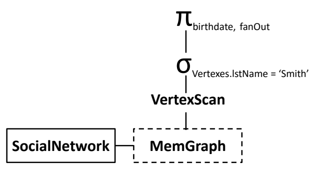


Figure 5: QEP for Query Q_v.

Listing 4: Subgraph Pattern Query to Find Triangles Q_t

```
SELECT Count(P)
FROM MLGraph.Paths P Where P.Length = 3 AND
    ↪  P.Edges[0].Label = 'A' AND P.Edges
    ↪ [1].Label = 'B' AND P.Edges[2].Label
    ↪  = 'C' AND P.Edges[2].EndVertex = P.
    ↪ Edges[0].StartVertex
```

More interestingly, paths can be joined to query more complex sub-graph patterns. Similar to relational engines that can perform self-joins for a relational table, GRFusion allows self-joins of the paths of a given graph view. This is possible as the vertexes and the edges of the paths to join can be referenced by relational join predicates.

5 GRAPH-RELATIONAL QUERY PROCESSING

In this section, we explain how GRFusion evaluates graph-relational queries. Section 5.1 introduces the primitive graph operators of GRFusion, while Section 5.2 illustrates how the graph operators integrate with typical relational operators in a cross-data-model QEP, where the graph operators appear in the leaf level of the QEP. Then, Section 5.3 discusses the conceptual query evaluation of graph-relational queries in GRFusion.

5.1 Graph Operators

GRFusion defines three primitive 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 [28] to avoid eager generation of paths that might not be required by parent operators. The reason of this design decision is that many queries (e.g, reachability) limit the number of paths to be retrieved, and consequently generating all/multiple paths may be expensive and unnecessary.

5.1.1 Vertex Scan and Edge Scan Operators. Operators *VertexScan* and *EdgeScan* allow GRFusion to iterate over the vertexes and edges of a given graph view, respectively. For example, the *VertexScan* operator provides an alternative access method for accessing the vertexes of a graph view, where the fan-in and fan-out properties of any vertex can be efficiently retrieved in constant time. To illustrate, consider Query Q_v in Listing 5. Q_v selects from the set of vertexes of the SocialNetwork graph view, and then applies some relational operators afterwards. To evaluate Q_v, GRFusion constructs the query execution pipeline, say QEP_v, as in Figure 5. Operator *VertexScan* scans the vertexes of the graph defined by the SocialNetwork graph view from the in-memory graph structure (represented as *MemGraph* in Figure 5, that references the singleton graph structure of the graph view). Vertexes with last name 'Smith' are selected and a relational projection operation selects only the birth date and the fan-out properties.

Listing 5: Vertexes Selection Query

```
SELECT VS.birthdate, VS.fanOut
FROM SocialNetwork.Vertexes VS
WHERE VS.lstName = 'Smith'
```

5.1.2 The PathScan Operator. 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, i.e., the paths in GRFusion are simple paths. In particular, the query optimizer maps a logical *PathScan* operator into *DFScan*, *BFScan*, or *SPScan*, corresponding to depth-first search, breadth-first search, or shortest-path search physical operators, respectively. In this section, we focus on the logical semantics of the path scan operator. We defer the discussion of the physical operators to Section 6.

As a logical operation, the paths-discovery process in GRFusion starts from a set of start vertexes to avoid materializing all possible paths. These start vertexes are either stated explicitly in the query (e.g., PS.StartVertex.Id = Value) or are generated by other 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. 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. Notice that the paths in GRFusion are not eagerly materialized by a PathScan operator, rather they are lazily generated.

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 6 demonstrates the query evaluation pipeline QEP_p that 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 6. 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, to be 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.

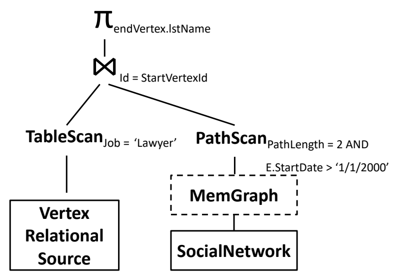


Figure 6: GRFusion joins a relational table with a graph-view traversal-operator for Query Q_p.

5.2 Cross-Model Query-Execution-Pipelines

A query in GRFusion can reference relations or relational views with graph views simultaneously. A pure relational engine has a main structure (i.e., tuple) that is passed among the relational operators in a query evaluation pipeline (QEP). GRFusion allows its query engine to view data by two different data models, namely, the relational model and the graph model. GRFusion allows a single QEP to have two main categories of operators that interact seamlessly in a QEP. The first category contains the relational operators (e.g., select, project, relational join) that can interact directly with relational tables. The second category contains graph operators that can operate on graph views. GRFusion integrates both categories of operators by allowing a relational operator to operate on the result of a graph operator. In particular, GRFusion unifies the interface of the output of both the relational and the graph operators. Specifically, the query engine of GRFusion abstracts graph processing by using three data types that extend the $Tuple$ data type, namely the $Vertex$, $Edge$, and $Path$ data types, where each has a schema that depends on the queried graph-view, as explained below.

In GRFusion, a vertex, say V, is represented in a QEP by a tuple, say T, where each attribute of V becomes an attribute in T. For example, a graph vertex in Listing 1 is represented by a tuple with attributes: (uId, $lstName$, $birthdate$). In addition, Vertex V has the following properties:

- **FanOut**: Contains the number of V's outgoing edges.
- **FanIn**: Contains the number of V's incident edges.

An edge E is represented by a tuple with attributes corresponding to E's attributes in addition to the following attributes:

- **From**: Contains the start vertex of Edge E.
- **To**: Contains the end vertex of Edge E.

GRFusion defines the $Path$ data type, where a path, say P, is a sequence of identifiers of the edges that form P. In particular, P is an extended tuple with the following attributes defining its schema:

- **Length**: Is the number of edges in P.
- **StartVertex**: Is the start vertex of P.
- **EndVertex**: Is the end vertex of P.
- **Vertexes**: Is the list of vertexes forming P.
- **Edges**: Is the list of edges forming P.

5.3 Conceptual Evaluation of Graph-Relational Queries in GRFusion

GRFusion addresses the impedance mismatch between the graph model and the relational model by unifying the type of the elements that move among the relational and the graph operators within a QEP. To illustrate, we list below the high-level steps that describe GRFusion's conceptual evaluation of declarative graph-relational queries, i.e., ones that reference relation(s) and graph-view(s):

- The relational tables and views are joined together using all the relational predicates in the WHERE clause of the query. This step yields a single resultant relation, say R.
- Each graph operator operates on a graph view, say GV, using its in-memory singleton graph-structure, say Mem_{GV}. In case of using different aliases on the same graph view, each alias is assigned an independent pointer to Mem_{GV}.
- When querying a combination of relations, relational views, vertexes, edges, or paths, all the graph operators operate only on graph views. Observe that the output of each graph operator is an extended type of the relational $Tuple$ type. Hence, the output of the graph operators can be ingested by the relational operators (e.g., the joins) in the same QEP seamlessly, where a relational join outer tuple can be used to probe a graph operator in the inner (e.g., see Figure 6).
- The predicates in the WHERE clause of the query that have not been consumed in producing R are used to join R with all the vertexes, edges, and paths referenced by the query.
- The SELECT list is used to perform projection.

6 QUERY OPTIMIZATION

GRFusion optimizes graph-traversal queries with two objectives in mind: (1) pruning undesired paths as early as possible to optimize the runtime, and (2) favoring traversal algorithms with less-memory requirements. The second goal is vital as memory should be consumed discreetly in an in-memory system. Optimization techniques for early pruning are discussed in Sections 6.1 and 6.2. In Section 6.3, we address the traversal-algorithm selection.

6.1 Path Length Inference

The query optimizer of GRFusion infers the allowed length of the paths described by the queries. The main objective is to make sure that a path returned from the PathScan operator is unlikely to be rejected by a parent operator (e.g., a join operator) due to a predicate referencing the path length. For instance, if a query has the filter "$PS.Edges[5..*].Att1 = Value$", then PathScan infers that the minimum path length to return is 6 (indexing is zero-based). Hence, PathScan will not return a path of length 5 or less. Many real-world queries specify the length of the desired paths, e.g., triangle-counting queries [48] specify a path length of three, the popular friends-of-friends queries restrict a path length to two, and many reachability queries put a cap on the maximum length of the path connecting the queried endpoints.

For each collection of paths, say PS, that is referenced in the *From*-clause, the query optimizer analyzes the predicates referencing the length of PS explicitly (e.g., PS.Length = value), or implicitly (e.g., by analyzing the logical operators as in PS.Edges[5..*].Att1 = X AND PS.Edges[7..9].Att2 = Y), to predict the range of allowed lengths of the paths to return. Then, the inferred path length is considered by PathScan while traversing the graph (e.g., an inferred maximum path length of 8 will prune any path of length ≥ 9).

6.2 Pushing Filters Ahead of Path Scans

To prune paths early, all the filters related to discovering the paths of a graph view are pushed ahead of the *PathScan* operator. For instance, for a graph view's paths, say PS, Predicate "$PS.Edges[0..*].Cost < 10$" is pushed so that *PathScan* can prune any potential path explored with an edge of cost ≥ 10. Similarly, predicates that refer to aggregates on a path's attributes will be computed and checked during the *PathScan* evaluation. For example, consider a query, say Q, with the predicate "$Sum(PS.Edges.Cost) < 100$". When *PathScan* explores Path P while evaluating Q, *PathScan* will accumulate the cost-attribute of the edges of P during the traversal. If the accumulated cost exceeds 100, P will be dropped and will not flow to the operators next in the QEP.

6.3 Logical to Physical Operator Mapping

Recall from Section 5.1.2 that the *PathScan* operator is a logical operator that is mapped into one of three physical traversal operators for execution, namely, depth-first search, breadth-first search, and shortest-path search based on Dijkstra's algorithm [24].

The shortest-path physical operator, namely *SPScan*, is very useful in top-k shortest path queries. Listing 6 illustrates how the user can instruct the optimizer to use *SPScan*. Given a non-negative numerical edge attribute, *SPScan* traverses the graph using Dijkstra's algorithm [24], and returns the next shortest-path as requested (i.e., pulled) by the parent operator in the QEP. *SPScan* is useful in many applications, e.g., recommendation systems and route discovery, to avoid the costly straightforward plan, i.e., avoid enumerating all paths, then filtering, sorting, and then returning the top ones.

For general graph-traversals where shortest paths are not defined, GRFusion can use either a depth-first search (i.e., a *DFScan* operator), or a breadth-first search (i.e., a *BFScan* operator). The user can give a query hint to decide on depth-first or breadth-first evaluations. To illustrate how GRFusion decides on the physical operator to perform a general graph traversal in the absence of an explicit query-hint, assume that a query, say Q, searches for Path P of Length L. Assume further that Query Q targets a graph view where the average fan-out is F. Following an analysis similar to that in [41], a depth-first search can contain on average $F * L$ vertexes in its stack data structure. In contrast, a breadth-first search can contain F^L vertexes in its queue data structure. Hence, GRFusion uses BFS if $F < \sqrt[L-1]{L}$ to optimize for memory. This optimization is applicable if the path length can be inferred and by maintaining the average fan-out statistic for each graph view in the system catalog. Otherwise, GRFusion uses the default scan operator that the user can set based on the expected workload (e.g., BFS can still be better if the underlying graph has a large diameter and frequent queries find the desired paths after one or two hops). GRFusion has a configuration to store the average fan-out of graph views as a statistics object. If this configuration is enabled, GRFusion runs a thread in the backend to compute the average fan-out using the compact graph-view structures.

Listing 6: Declarative Shortest-Path Query

```
SELECT TOP 2 PS
FROM RoadNetwork.Paths PS HINT(SHORTESTPATH
    ↪ (Distance)), RoadNetwork.Vertexes
    ↪ Src, RoadNetwork.Vertexes Dest
WHERE PS.StartVertex.Id = Src.Id AND PS.
    ↪ EndVertex.Id = Dest.Id AND Src.
    ↪ Address = "Address 1" AND Dest.
    ↪ Address = "Address 2"
```

7 EXPERIMENTAL EVALUATION

We experimentally evaluate the performance of GRFusion, a realization of the proposed *Native G+R Core* approach inside a centralized version of VoltDB. We compare GRFusion to the state of the art of the *Native Relational-Core* approach, namely SQLGraph [46], and we compare to Grail [25]. Although Grail uses a different computational model than GRFusion, they both have the common ground of executing queries through an RDBMS. We also compare GRFusion to two popular specialized graph systems, Neo4j [4] and Titan [11]. The reason for comparing with specialized graph systems, which follow the *Native Graph-Core* approach, is to show that graph-traversal queries can be efficiently handled by GRFusion.

Mitigating the disk IO cost from the baselines: As GRFusion is an in-memory system, the experiments are designed to mitigate the disk cost of all the baselines we compare to. We implemented SQLGraph and Grail as described in [46], and [25], respectively, on top of the in-memory VoltDB system. We configured Titan to use the in-memory storage configuration, and we set Neo4j to run and execute over a RAM disk on Linux.

We consider two important categories of graph queries, namely, traversal-based queries and pattern-matching queries, where the queries can take additional filtering predicates. For traversal-based queries, we evaluate reachability queries (e.g., Listing 3). We also evaluate shortest-path queries to compare with Grail [25]. For pattern-matching queries, we evaluate the triangle-counting query using filtering predicates on the edges while varying selectivity. The triangle-counting query is a primitive operator in many machine-learning and knowledge-discovery techniques, e.g., [48]. Experiments are conducted on a machine running Linux kernel 3.17.7 on 32 cores of Intel Xeon 2.90 GHz with 384 GB of main-memory.

7.1 Datasets

We use real graph datasets that represent four different application domains, namely, road networks, biological networks, authorship networks, and social networks. For the road networks, we use the continental-sized Tiger dataset [9] that covers the entire U.S., where the edges represent road segments, and the vertexes represent road intersections. For the biological networks, we use the String protein-interaction dataset [8], where the vertexes represent proteins, and the edges represent interactions among the proteins. We use the DBLP [1] dataset for the authorship networks, where the vertexes represent authors, and the edges represent co-authorship relations. We use the Twitter dataset [3] for the social-network application, where the dataset represents the follower graph of Twitter. The vertexes in Twitter represent users, where an edge from User A to User B denotes that User A follows User B. Table 2 summarizes the properties of these datasets.

Controlling sub-graph selectivity: We study the effect of selecting a subgraph from an underlying graph before performing a graph operation (e.g., selecting a sub-graph containing 10% of the edges of the entire graph before executing a shortest-path query or a topological pattern-matching query on the selected sub-graph). For each dataset, we vary the selectivity of the queries from 5% to 50%.

Evaluating the effect of graph-views in the *Native G+R Core* approach: To accurately study the performance gains due to the graph-views of the *Native G+R Core* w.r.t. the *Native Relational-Core* approach, we use breadth-first search instead of depth-first search, and we do not push the predicates ahead of the path scan operator in GRFusion for all the reachability-queries experiments.

7.2 Unconstrained Reachability Queries

We contrast the performance of GRFusion with that of SQLGraph, Neo4j, and Titan, when processing reachability queries without filtering predicates on the graph edges. Given two nodes, say A and B, a reachability query returns true if a path exists from Node A to Node B. The query-processing time of a reachability query is affected by the path length of the query result. The reason is that the increase in the number of edges traversed directly corresponds to the number of relational joins in the *Native Relational-Core* approach (e.g., SQLGraph).

For each dataset in Table 2, we generate random reachability queries with different path lengths that make the query endpoints connected. We vary the path length from 2 to 20. For each path length, say l, we generate $10,000$ random queries, say Q_l. We run Q_l and measure the average query-processing time using GRFusion, SQLGraph, Neo4j, and Titan.

Figure 7 shows the average query-processing time of running the queries using all four systems, where the x-axis and the y-axis give the path-length of the query answers and the query-processing time in milliseconds, respectively. GRFusion achieves up to four orders-of-magnitude speedup in query-processing time compared to SQLGraph, where the speedup increases as the graph size increases. For instance, the speedup reaches 599x for the DBLP graph, and 2483x for the larger String graph. The reason is that GRFusion uses the compact graph view that captures the graph topology, where the graph views act as navigational indexes. Hence, GRFusion does not perform any relational join on the relational sources to traverse the graphs. In contrast, SQLGraph performs a relational join for each edge traversal during the path discovery process. Consequently, the query-processing time in SQLGraph increases as the path length of the query result increases. Moreover, the SQLGraph approach may not scale in main-memory RDBMSs when the graph size is very big due to the size of the intermediate results of the relational joins. To illustrate, in Figure 7(d), in the Twitter dataset, the *Native Relational-Core* represented by SQLGraph does not execute if the query evaluation requires more than four relational joins. The reason is that the intermediate temporary-memory of the join operators exceeds 6 GB, which is 60 times the 100-MB recommended limit in VoltDB. To allow room for query-evaluation pipelining to reduce the intermediate results, and to mitigate the limits of the main-memory, we execute the Twitter queries on a popular disk-based commercial RDBMS. The queries on the Twitter graph time-out after 5 hours of execution when the traversal depth of the queries exceeds four. In contrast, the systems following the *Native Graph-Core* represented by Neo4j and Titan scale for deep graph-traversal queries on large graphs as the overhead of the relational joins does not exist, where a deep graph-traversal query is a query that explores paths of long lengths, i.e., many edges, which corresponds to many joins in the *Native Relational-Core*. However, GRFusion that realizes the proposed *Native G+R Core* approach is able to scale for deep graph-traversal queries with better performance than those of the native graph systems.

Comparing GRFusion to the specialized graph databases Neo4j and Titan, GRFusion has a query-time speedup that exceeds three orders-of-magnitude for the String graph (see Figure 7(c)). We attribute these performance gains of GRFusion over the specialized graph databases to implementation factors and not to a fundamental change in the computational model. The reason is that GRFusion is based on VoltDB that has a low-overhead concurrency model (e.g., no lock overhead as in the specialized graph

Dataset	Number of Vertexes	Number of Edges	Construction Time	Memory Size (GB)
Tiger Road Network	24,412,259	58,698,439	2.08 Min	0.88
DBLP Co-Author Network	1,007,047	6,592,656	1.59 Sec	0.09
String Protein Network	1,520,673	348,473,440	3.81 Min	4.17
Twitter Follower Network	41,652,230	1,468,365,182	10.87 Min	17.81

Table 2: The graph views in GRFusion are fast to construct with low memory overhead for the datasets of the evaluation.

databases). Moreover, VoltDB has an optimized memory manager written in C++ that is significantly more efficient than the JAVA memory managers of both Neo4j and Titan. Theoretically, if we remove all the implementation-specific factors, the performance of GRFusion should be comparable to that of the specialized graph systems as both are processing native graph representations. In Section 7.3, we present the performance of GRFusion when evaluating queries that do not only consult the graph topology, but also the edges' attributes stored in the relational sources.

7.3 Reachability Queries with Filtering Predicates

We evaluate the performance of reachability queries in GRFusion and compare it to the baselines when the queries are associated with a filtering predicate. To study the effect of sub-graph selectivity (i.e., selecting the sub-graph to perform the query on), we generate reachability queries similar to the ones described in Section 7.2 with varying selectivities. We vary the selectivity parameter from 5% to 50% using synthesized edge attributes to control the selectivity. We limit the path length of the results of the generated queries to 20 to emphasize the effect of the selectivity of the sub-graph to operate on.

Figure 8 shows the average query-processing time for executing the reachability queries with filtering predicates using all 4 systems and datasets, where the x-axis and the y-axis are the edge-selectivity of the queries, and the query-processing time in milliseconds, respectively. Observe that, for the relatively-small DBLP graph in Figure 8(a), SQLGraph outperforms Neo4j and Titan as the relational engine can execute joins and apply filtering predicates efficiently on relations of small cardinalities. GRFusion outperforms both SQLGraph and the specialized graph engines. There are two main reasons behind GRFusion's performance gains. First, GRFusion uses a compact graph data structure to perform the traversal and avoids relational joins completely to explore the underlying graph. Second, GRFusion relies on the relational engine to evaluate the filtering predicates on the edges. Recall that GRFusion has a direct pointer to an edge's tuple that is accessed in $O(1)$ time to evaluate the query filtering-predicate using the efficient logic of the relational engine. Hence, GRFusion combines the strengths of both the graph systems and the relational systems to achieve the best-of-both-worlds in terms of performance. However, the efficient evaluation of the filtering predicates and the cost of the relational joins in SQLGraph do not pay off when the size of the relations increase. To illustrate, refer to Figure 8(b), where the performance of SQLGraph degrades as more edges are selected. For the String dataset in Figure 8(c), SQLGraph exceeds the temporary memory limits of VoltDB after selecting a subgraph of size larger than 25% of the queried graph for the reasons illustrated in Section 7.2. For the largest Twitter dataset, SQLGraph is not able to perform even on a subgraph of a 5% selectivity. The reason is that the cost of 20 relational joins on the large Twitter table exceeds the temporary-memory limits of VoltDB, and time-out the queries on a commercial disk-based RDBMS after 5 hours of execution. Also, as the number of self-joins increases in the *Native Relational-Core* approach, the relational optimizer may not be able to select the best join algorithm due to inaccurate cardinality estimations of the intermediate results (see [27] for details).

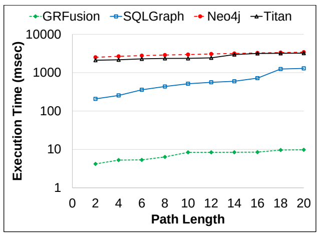
The relational engine is efficient in performing filtering predicates. This set of experiments demonstrates the power of extending the relational engine with a native graph-core processor that is optimized for graph traversals and that uses efficient memory representation. Figure 8 demonstrates the scalability and the efficiency of GRFusion in contrast to the baselines in handling graph queries with filtering predicates. Notice that increasing the edge-selectivity factor of the queries has less impact on Neo4j, Titan, and GRFusion than on SQLGraph w.r.t. query-processing time. The reason is that these queries are evaluated on a graph structure by performing the filtering predicates on the fly as the graph is being traversed. The selectivity affects the query performance of all the approaches. However, it is more impactful in the case of pure-relational evaluation. For example, in Figure 8(b), the processing time of SQLGraph increases by 138x when changing the selectivity from 5% to 50%, in contrast to an increase of 1.72x in GRFusion on the same setup.

7.4 Sub-Graph Pattern Matching

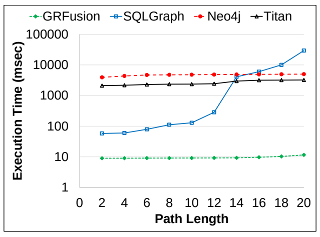
We evaluate the performance of the triangle-counting query. Given a graph, say G, a triangle-counting query, say Q_{TC}, counts all the sub-graphs of a triangle pattern (e.g, see Listing 4). Notice that the *Native Relational-Core* approach, e.g., SQLGraph, can scale for this specific pattern query as only two relational joins are needed for query evaluation. This is the reason for choosing this pattern query besides its importance as a primitive in many applications [48]. Figure 9 gives the performance of evaluating triangles queries on the DBLP, Tiger, and String graph datasets, where the x-axis and the y-axis are the edge-selectivity of the queries and the query-processing time in milliseconds, respectively.

Notice that in Figure 9, the SQLGraph approach outperforms both Neo4j and Titan when the selected sub-graph size is small, e.g., up to a selectivity of 10% for the DBLP dataset as in Figure 9(a). Also, notice that SQLGraph is more sensitive to the selectivity parameter than all the other approaches including GRFusion. Although only two joins are required by SQLGraph in this type of queries, increasing the number of tuples to join increases the query processing time, which results in better performance by Neo4j and Titan when increasing the selectivity parameter. For instance, Neo4j and Titan are more efficient than SQLGraph for the String dataset in Figure 9(c) for a selectivity parameter greater than 20%.

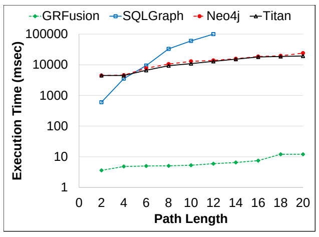
Figure 9 illustrates that GRFusion outperforms SQLGraph, Neo4j, and Titan by up to one order of magnitude in query performance. We attribute this performance advantage by GRFusion to the same reasons reported in Section 7.2.



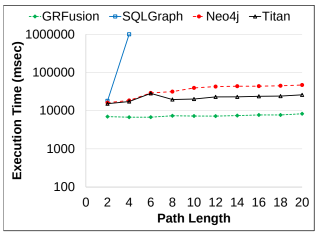
(a) DBLP Dataset



(b) Tiger Dataset

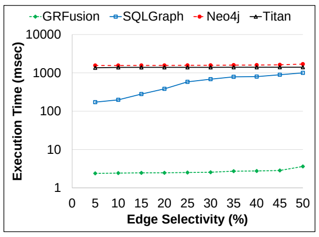


(c) String Dataset

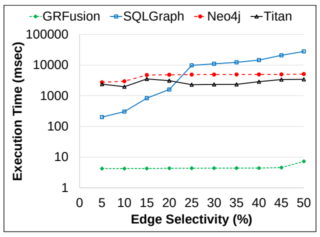


(d) Twitter Dataset

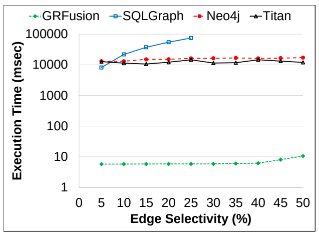
Figure 7: GRFusion achieves up to 4 orders-of-magnitude query-time speedup for *unconstrained reachability queries*.



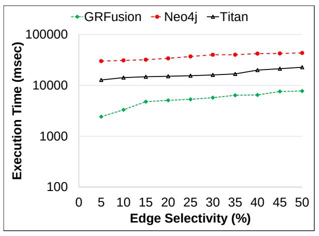
(a) DBLP Dataset



(b) Tiger Dataset

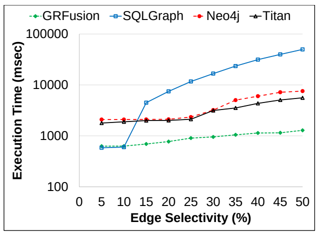


(c) String Dataset

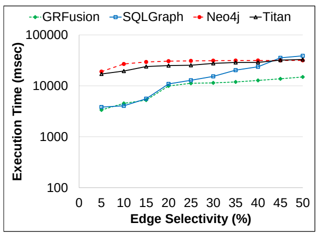


(d) Twitter Dataset

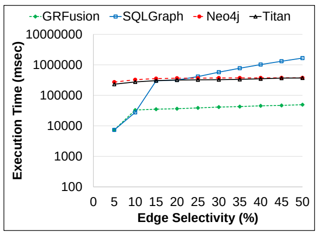
Figure 8: GRFusion achieves up to 4 orders-of-magnitude query-time speedup for *reachability queries with filtering predicates*.



(a) DBLP Dataset



(b) Tiger Dataset



(c) String Dataset

Figure 9: GRFusion finds all the triangles with filtering predicates with a query-time speedup of one order-of-magnitude.

7.5 Shortest-Path Queries with Filtering Predicates

We conduct an experiment using the Tiger road network to assess the performance of GRFusion in evaluating the single-source shortest-path query (or SSSP, for short) in contrast to Grail [25]. The purpose of this experiment is to show that a simple algorithm, e.g., Dijkstra's algorithm [24], executing inside a relational database system can achieve significant performance gains over a pure-relational approach, e.g., as in Grail [25], when evaluating SSSP queries, or more generally, intensive traversal queries. Notice that the computational model of Grail is based on the vertex-centric computational approach that is different from the graph-traversal model of GRFusion. However, both approaches have a common ground due to using an RDBMS in the evaluation. We implement the SSSP query of Grail as reported in Listing 3 in Grail's paper [25]. Our Grail implementation is an in-memory implementation on top of VoltDB to mitigate the disk IO cost, and we allow Grail to filter the edges while processing to report the effect of sub-graph selections on the query-execution performance.

We generate 1000 random sources from which we execute an SSSP query to all the other vertexes, and we report the average query execution time for various sub-graph selectivity factors.

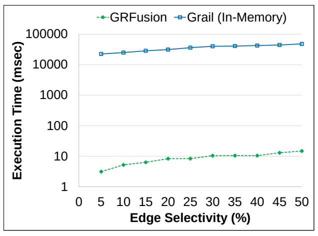


Figure 10: GRFusion executes SSSP queries natively inside an RDBMS few-thousand times faster than Grail.

Figure 10 gives the performance of evaluating SSSP queries on the Tiger road network, where the x-axis and the y-axis are the edge-selectivity of the queries and the query-processing time in milliseconds, respectively. GRFusion achieves more than three orders-of-magnitude query-time speedup w.r.t. Grail. Notice that we do not use an advanced SSSP evaluation method. Instead, we use a straightforward Dijkstra's algorithm that utilizes efficient filtering-predicates of the relational database engine. This emphasizes the point that having a native and an efficient graph representation inside an RDBMS can fill the gap between the RDBMSs and the graph algorithms that are designed for native graph structures, where these graph algorithms can achieve significant performance gains when compared to equivalent pure-relational query evaluation approaches.

7.6 The Overhead of Graph Views

As graph views are materialized in GRFusion, we report the construction time as well as the consumed memory space for each dataset. Table 2 illustrates that the construction time ranges from two seconds to 10 minutes according to the size of the graph. The reason is that the construction process passes only once by the vertexes relational-source as well as the edges relational-source. Similarly, Table 2 shows the memory size due to the materialization of the topology of every graph. The consumed memory is of acceptable overhead because only the graph topology is materialized, where each vertex and edge holds pointers to the relational data instead of replicating the relational data inside the graph views. For example, only 0.88 GB is needed to construct a graph view for the continental-sized US road network. Moreover, the overhead of updating the graph views is low. On average, it takes 0.04 milliseconds to add a new edge into an existing graph view, i.e., the total time to insert a tuple in the relational source as well as updating the topology of the corresponding graph view. For both the deletions and insertions of vertexes and edges, GRFusion incurs 5%-11% additional overhead to the time of manipulating the relational sources. The reason for this low overhead is that the logic of manipulating the graph views is linear in time w.r.t. the number of affected vertexes or edges as illustrated in Section 3.3.

8 RELATED WORK

Graphs Integration with Relational Databases: There is a plethora of database systems that adopt the graph data model (e.g., Neo4j [4] and Titan [11]). These systems have powerful graph querying features. However, it has been shown that for many graph queries, the performance of these systems can be achieved or exceeded by a vanilla relational database [25, 46]. For graph-relational queries, a graph database is useful if it is feasible to: a) import all the relational data into the graph database, or b) develop a custom layer where results from the graph database and the relational database are integrated to form the final results. In contrast, GRFusion allows efficient execution of graph-relational queries with neither the overhead of importing data nor the overhead of integrating query results from different systems. Commercial systems, e.g., Oracle Graph and Aster [45], follow the architecture of processing graph-relational queries using different run-time systems, where the results are combined at the end. For example, Aster allows defining graph functions that can be referenced in the FROM-Clause of a SQL statement. During query execution, the graph function is extracted and evaluated using a graph runtime system. Eventually, the result from the **external** graph-runtime is transformed into a relational table that can be integrated with the parent SQL query. Similarly, G-SPARQL [40] is a SPARQL-like language for querying attributed graphs, where a graph is represented and processed using a hybrid Memory/Disk model, and the query-execution is split between the RDBMS and a memory-based layer outside the RDBMS. In contrast, GRFusion executes the graph operations as well as the relational operations of a query through a cross-data-model QEP without leaving the realm of the RDBMS.

Several graph libraries and systems target graph analytics, e.g., CRAY Graph Engine [13], Pregel [34] and its open source version Giraph [2], GraphLab, GraphFrames [22]. For graph analytics, it may be acceptable to import data from relational databases for analytical purposes. In contrast, GRFusion also serves OLTP scenarios. This is possible as the relational data in GRFusion is not deeply copied into the graph views. Moreover, the updates to the relational data that affect the topology of the defined graph views incur little overhead to update the in-memory graph structures in GRFusion.

Relational Databases with Modified Layers for Graph Processing: In this category, the internals of an RDBMS are modified to some extent, but not to a level that executes a graph-relational query through the same QEP as in GRFusion. For example, SAP HANA Graph and GRAPHITE [37] allow graph operations to directly execute on the relational data in a column-store without replication. However, two different runtime components execute the graph-relational queries. In contrast, GRFusion uses a single runtime leading to better performance. In [18], an access method is proposed to process graphs stored on disk under certain locality assumptions. In contrast, GRFusion is a main-memory system that traverses a graph by realizing a light-weight structure describing the graph topology.

Extracting Graphs from Relational Databases: In this category, graphs stored in relational tables are extracted from the database system to be under the control of an independent application. This independent application allows for querying the extracted graphs using graph APIs. Ringo [38] and GraphGen[51, 52] are representatives of this approach. In contrast, GRFusion processes graphs inside the relational database and does not extract the graphs outside the realm of the database engine. Additionally, GRFusion supports dynamic graphs, where online updates are possible. Notice that to support graph-relational queries, e.g., in Ringo or GraphGen, the relational part of the query should be processed by the relational database, and the graph operations should be processed by Ringo or GraphGen, where another external layer will be responsible for integrating the graph results and the relational results into the final query result.

Encoding Graphs in Relational Databases: In this line of work (e.g., SQLGraph [46], Grail [25]), graphs are stored in relational tables with schema optimized for specific graph queries. After encoding graphs in a vanilla relational database, a translation layer is designed to translate the supported graph queries into complex SQL statements for the relational database to execute. Although the query performance of this approach is comparable to specialized graph databases for specific queries, these systems make it difficult for users to write declarative graph-relational queries. In particular, the schema of the relations storing the graph data may not be suitable for users to query directly and join with other relational data. The reason is that the schema is usually auto-generated based on the input graph for optimization purposes.

Tailored Operators for Specific Graph Operations: In this category, several research efforts (e.g., [17, 20, 21]) have been conducted since the 1980s and until recently (e.g., [16, 26]). However, most of these efforts target specific query types (e.g., transitive closure, shortest paths). Unlike GRFusion, these approaches also do not support a unified/cross-model declarative language to query both graph and relational objects simultaneously. In [17, 20], Relational Algebra is extended with operators to allow for recursive queries. Although the proposed recursive algebra helps execute some graph traversal queries, query execution is not efficient because the graph operators execute over relational tables and not over native graph representations. For instance, several iterations with insertions into temporary tables are needed to keep track of the traversal state. Similarly, Vertica [31] presents optimizations for graph-relational queries. However, the graph operations execute over pure relational structures and not on graph representations. Thus, costly relational joins are mandatory in many cases to traverse graphs. In contrast, GRFusion's graph operators process native graph structures in main-memory without performing costly joins and without manipulating temporary tables to traverse a graph topology. Dar et al. [21] use relational operators repetitively to compute the transitive closure of a graph represented in a predefined relational schema. Gao et al. [26] present specific optimizations to process shortest-path queries over graphs stored in a relational database. GRFusion is more general and can join graph views with relational tables in the same query. Moreover, GRFusion addresses the impedance mismatch between the graph model and the relational model. In EmptyHeaded [16], graphs in a relational storage are queried using a datalog-like language [29]. The core idea of EmptyHeaded is to leverage join algorithms with strong theoretical guarantees in addition to using advanced query-compilation techniques. In contrast, GRFusion avoids relational joins completely when traversing the topology of a graph view.

9 CONCLUSION

We introduce the notions of in-memory materialized graph views, graph operators that seamlessly integrate with relational operators in query evaluation pipelines, memory management, and query optimization techniques for optimizing graph-relational queries. GRFusion is a realization of the proposed *Native G+R Core* approach inside VoltDB. The key idea behind GRFusion is to show the effect of extending an RDBMS to handle natively and seamlessly graph and relational data through cross-data-model QEPs. We introduce the PATH construct, and the extended SQL language of GRFusion to declaratively express graph-relational queries. GRFusion constructs in-memory graph structures to capture the graph topology and exploits the relational engine's power in evaluating the relational constructs of the queries. Consequently, GRFusion efficiently handles deep graph-traversal queries without any relational joins to explore the connectives of the vertexes of a graph. We evaluate GRFusion using various graph queries w.r.t specialized graph engines and systems following the *Native Relational-Core* approach, where GRFusion achieves up to four orders-of-magnitude query-time speedup.

REFERENCES

[1] http://dblp.uni-trier.de/xml/.
[2] http://giraph.apache.org/.
[3] http://konect.uni-koblenz.de/networks/twitter.
[4] http://neo4j.com/.
[5] https://github.com/tinkerpop/gremlin/wiki.
[6] https://github.com/voltdb/voltdb/.
[7] https://github.com/voltdb/voltdb/.
[8] http://string-db.org/.
[9] https://www.census.gov/geo/maps-data/data/tiger.html.
[10] https://www.voltdb.com/.
[11] http://thinkaurelius.github.io/titan/.
[12] http://www.caida.org/data/passive/.
[13] http://www.cray.com/products/analytics/cray-graph-engine.
[14] Oracle timesten: http://www.oracle.com/us/products/database/timesten.
[15] soliddb: https://teamblue.unicomsi.com/products/soliddb.
[16] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *SIGMOD '16*.
[17] R. Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Trans. Softw. Eng.*, July 1988.
[18] R. Chen. Managing massive graphs in relational dbms. In *BIG DATA '13*.
[19] T. Chondrogiannis, P. Bouros, J. Gamper, and U. Leser. Alternative routing: K-shortest paths with limited overlap. In *SIGSPATIAL '15*.
[20] L. S. Colby. A recursive algebra and query optimization for nested relations. In *SIGMOD '89*.
[21] S. Dar, R. Agrawal, and H. V. Jagadish. Optimization of generalized transitive closure queries. In *ICDE '91*.
[22] 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. of the 4th Int. Workshop on Graph Data Management Experiences and Systems*, GRADES '16.
[23] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *SIGMOD '13*.
[24] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1, 1959.
[25] J. Fan, A. G. S. Raj, and J. M. Patel. The case against specialized graph analytics engines. In *CIDR '15*.
[26] 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 Endow.*, 5(4), Dec. 2011.
[27] A. Ghazal, D. Seid, A. Crolotte, and M. Al-Kateb. Adaptive optimizations of recursive queries in teradata. In *SIGMOD '12*.
[28] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2), June 1993.
[29] T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou. Datalog and recursive query processing. *Found. Trends databases*, 5(2):105–195, Nov. 2013.
[30] M. S. Hassan, W. G. Aref, and A. M. Aly. Graph indexing for shortest-path finding over dynamic sub-graphs. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1183–1197, 2016.
[31] A. Jindal, S. Madden, M. Castellanos, and M. Hsu. Graph analytics using vertica relational database. In *BIG DATA '15*.
[32] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, Aug. 2008.
[33] D. Kernert, F. Köhler, and W. Lehner. Slacid - sparse linear algebra in a column-oriented in-memory database system. In *SSDBM '14*.
[34] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD '10*.
[35] K. Molka and G. Casale. Contention-aware workload placement for in-memory databases in cloud environments. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 2(1), Nov. 2016.
[36] H. Montaner, F. Silla, H. Fröning, and J. Duato. Memscale: In-cluster-memory databases. In *CIKM '11*.
[37] M. Paradies, W. Lehner, and C. Bornhövd. Graphite: An extensible graph traversal framework for relational database management systems. In *SSDBM '15*.
[38] Y. Perez, R. Sosič, A. Banerjee, R. Puttagunta, M. Raison, P. Shah, and J. Leskovec. Ringo: Interactive graph analytics on big-memory machines. In *SIGMOD '15*.
[39] M. Sadoghi, S. Bhattacherjee, B. Bhattacharjee, and M. Canim. L-Store: A real-time OLTP and OLAP system. In *EDBT '18*.
[40] S. Sakr, S. Elnikety, and Y. He. Hybrid query execution engine for large attributed graphs. *Inf. Syst.*, 41:45–73, May 2014.
[41] A. D. Sarma, H. Lee, H. Gonzalez, J. Madhavan, and A. Halevy. Consistent thinning of large geographical data for map visualization. *ACM Trans. Database Syst.*, 38(4), Dec. 2013.
[42] M. Sarwat, S. Elnikety, Y. He, and G. Kliot. Horton: Online query execution engine for large distributed graphs. In *ICDE '12*.
[43] M. Sarwat, S. Elnikety, Y. He, and M. F. Mokbel. Horton+: A distributed system for processing declarative reachability queries over partitioned graphs. *Proc. VLDB Endow.*, 6(14), Sept. 2013.
[44] A. Shahvarani and H.-A. Jacobsen. A hybrid b+-tree as solution for in-memory indexing on cpu-gpu heterogeneous computing platforms. In *SIGMOD '16*.
[45] D. Simmen, K. Schnaitter, J. Davis, Y. He, S. Lohariwala, A. Mysore, V. Shenoi, M. Tan, and Y. Xiao. Large-scale graph analytics in aster 6: Bringing context to big data discovery. *Proc. VLDB Endow.*, 7(13), Aug. 2014.
[46] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie. Sqlgraph: An efficient relational-based property graph store. In *SIGMOD '15*.
[47] J. R. Thomsen, M. L. Yiu, and C. S. Jensen. Effective caching of shortest paths for location-based services. In *SIGMOD '12*.
[48] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: Counting triangles in massive graphs with a coin. In *KDD '09*.
[49] L. Wang, M. Zhou, Z. Zhang, Y. Yang, A. Zhou, and D. Bitton. Elastic pipelining in an in-memory database cluster. In *SIGMOD '16*.
[50] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li. Scaling multicore databases via constrained parallel execution. In *SIGMOD '16*.
[51] K. Xirogiannopoulos and A. Deshpande. Extracting and analyzing hidden graphs from relational databases. In *SIGMOD '17*.
[52] K. Xirogiannopoulos, U. Khurana, and A. Deshpande. Graphgen: Exploring interesting graphs in relational data. *Proc. VLDB Endow.*, 8(12), Aug. 2015.
[53] C. Yao, D. Agrawal, G. Chen, B. C. Ooi, and S. Wu. Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases. In *SIGMOD '16*.