G-SPARQL: A Hybrid Engine for Querying Large Attributed Graphs

Sherif Sakr*
National ICT Australia
UNSW, Sydney, Australia
ssakr@cse.unsw.edu.eu

Sameh Elnikety
Microsoft Research
Redmond, WA, USA
samehe@microsoft.com

Yuxiong He
Microsoft Research
Redmond, WA, USA
yuxhe@microsoft.com

ABSTRACT

We propose a SPARQL-like language, *G-SPARQL*, for querying attributed graphs. The language expresses types of queries which of large interest for applications which model their data as large graphs such as: pattern matching, reachability and shortest path queries. Each query can combine both of structural predicates and value-based predicates (on the attributes of the graph nodes and edges). We describe an algebraic compilation mechanism for our proposed query language which is extended from the relational algebra and based on the basic construct of building SPARQL queries, *the Triple Pattern*. We describe a hybrid Memory/Disk representation of large attributed graphs where only the topology of the graph is maintained in memory while the data of the graph is stored in a relational database. The execution engine of our proposed query language splits parts of the query plan to be pushed inside the relational database while the execution of other parts of the query plan are processed using memory-based algorithms, as necessary. Experimental results on real datasets demonstrate the efficiency and the scalability of our approach and show that our approach outperforms native graph databases by several factors.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Graphs and networks

Keywords

Graphs, Graph Queries, SPARQL

1. INTRODUCTION

Recently, graph query processing has attracted a lot of attention from the database research community due to the increasing popularity of modeling the data as large graphs in various application domains such as social networks, bibliographical networks and knowledge bases. In many real applications of these domains, both the graph topological structure in addition to the properties of the vertices and edges are important. For example, in a social network, a vertex which represents a *person* object can be described with a property that represents the *age* of a person while the topological structure could represent different types of relationships (directed edges) with a group of people. Each of these relations can be described by a *start date* property. Each vertex is associated with a basic descriptive attribute that represents its *label* while each edge has a *label* that describes the type of relationship between the connected vertices. Figure 1 shows a snippet of an example large graph where a vertex represents an entity instance (e.g., *author*, *paper*, *conferences*) and an edge represents a structural relationship (e.g., *co-author*, *affiliated*, *published*). In addition, there are attributes (e.g., *age*, *keyword*, *location*) that describe the different graph vertices while other attributes (e.g., *order*, *title*, *month*) describe the graph edges.

In general, there are three common types of queries which of large interest for applications with large graphs: 1) *Pattern match query* that tries to find the existence(s) of a pattern graph (e.g., path, star, subgraph) in the large graph [21, 6]. 2) *Reachability query* that verifies if there exists a path between any two vertices in the large graph [4, 8]. 3) *Shortest path query* which returns the shortest distance (in terms of number of edges) between any two vertices in the large graph [1, 18]. In practice, a user may need to pose a query on the large graph that can involve more than one of the common graph query types. The problem studied in this paper is to query a graph associated with attributes for its vertices and edges (called as *attributed graph*) based on both structural and attribute conditions. Unfortunately, this problem did not receive much attention in the literature and there is no solid foundation for building query engines that can support a combination of different types of queries over large graphs. In particular, examples of motivating queries to our work include:

1) Find the names of two authors, `X` and `Y`, where `X` and `Y` are connected by a path of any length (number of edges), the author `X` is affiliated at `UNSW`, the author `Y` is affiliated at `Microsoft` and each of the authors has published a paper in `VLDB'12`. *This query involves pattern matching and reachability expressions.*

2) Find the names of two authors, `X` and `Y`, where `X` and `Y` are connected by a path of any length, the author `X` is affiliated at `UNSW`, the author `Y` is affiliated at `Microsoft`, each of the authors has published a paper in `VLDB'12` as a `first` author and each of the authors has an `age` which is more than 35. *This query involves pattern matching expression with conditions on the attributes of graph nodes and edges in addition to reachability expression.*

3) Find the names of two authors, `X` and `Y`, and the connecting path(s) between them where `X` and `Y` are connected by a path with a length which is less than 3 edges, the author `X` is affiliated

*This work has been done while the author was visiting Microsoft Research Laboratories, Redmond, USA

CIKM'12, October 29–November 2, 2012, Maui, HI, USA.

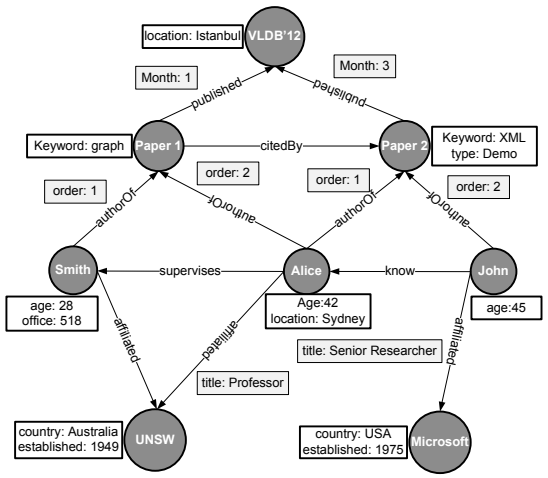


Figure 1: An example attributed graph.

at `UNSW`, the author `Y` is affiliated at `Microsoft` and each of the authors has published a paper in `VLDB'12` as a `first` author. *This query involves pattern matching expression in addition to a reachability expression which is constrained by a path filtering condition. The query returns the information of the connecting paths(s) as part of the results.*

In this paper, we present an approach for querying large attributed graphs which relies on a hybrid main memory/disk-based relational representation of the graph database and devising efficient algebraic-based query processing mechanisms for different types of graph queries. In our approach, the incoming queries are compiled into algebraic plans where parts of the query plans are pushed down and executed inside the relational database layer while the rest of the query plan is processed using memory-based algorithms. In principle, the main reason behind our decision for relying on a relational database at the physical storage layer is to leverage the decades' worth of research in the database systems community. Some optimizations developed during this period include the careful layout of data on disk, indexing, sorting, buffer management and query optimization. By combining the memory representation of the graph topology and memory-based graph algorithms with the storage layer of the RDBMS, we are able to gain the best features of both worlds. Our goal is to optimize the performance of query processing while minimizing the memory consumption and achieving the scalability goals. In particular, our main contributions can be summarized as follows:

1) We propose a SPARQL-like language, called *G-SPARQL*, for querying attributed graphs. The language enables combining the expression of different types of graph queries into one request. We show that the language is sufficiently expressive to describe different types of interesting queries (Section 3).

2) We present an efficient hybrid Memory/Disk representation of large attributed graphs where only the *topology* of the graph is maintained in memory while the *data* of the graph is stored and processed using relational database (Section 4).

3) We describe an execution engine for our proposed query language which applies a split query evaluation mechanism where the execution of parts of the query plan is pushed inside the relational database while the execution of other parts is processed using memory-based algorithms, as necessary, for optimizing the query performance (Section 5).

4) We conduct extensive experiments with real datasets that demonstrate the efficiency of our approach (Section 6).

2. RELATED WORK AND LIMITATIONS OF EXISTING APPROACHES

Several techniques have been proposed in the literature for querying large graphs. However, in practice, the existing techniques turn to be inadequate, in many cases, for querying large attributed graphs due to the following limitations:

a) Most of the existing graph querying techniques follow the approach of building an index for storing information about the main features of the graph database. The structure and content of this index is usually optimized for accelerating the evaluation of *only one* of the common types of the graph queries but usually they cannot be used for accelerating the evaluation of other types of queries. For example, different subgraph query processing techniques exploit different types of graph features for building their indices (e.g., path [21], tree [20], subgraph [19]) while proposed techniques for handling reachability queries use different indexing mechanisms such as the 2-hop cover [4] and 3-hop cover [8]. In practice, answering user requests that can involve more than one of the common graph query types would require maintaining different type of indices which would be very expensive in terms of memory consumption. In addition, given the increasing sizes of the graph database, the efficiency of such indexing techniques will break down after a certain limit is reached.

b) The existing graph querying methods that have been presented in the literature mainly focus on querying the topological structure of the graphs [15, 19, 21] and very few of them have considered the use of attributed graphs [17, 5]. In practice, it is more common that the querying requirements for the applications of large graph databases (e.g., social networks or bibliographical networks) would involve querying the graph data (attributes of nodes/edges) in addition to the graph topology. Answering queries that involve predicates on the attributes of the graphs (vertices or edges) in addition to the topological structure is more challenging as it requires extra memory consumption for building indices over the graph attributes in addition to the structural indices in order to accelerate the query evaluation process. Furthermore, it makes the query evaluation and optimization process more complex (e.g., evaluation and join orders).

c) Several techniques have been proposed for querying RDF graphs [12, 7]. However, these approaches cannot be reused for querying general attributed graphs due to the differences in the data model and the specifications of the query requirements. For example, in the RDF data model, graph edges *cannot* be described by attributes. The initial specifications of SPARQL [14] did not provide any facility for expressing *path queries* or *reachability* expressions. In addition, some query types that are of common interest in the general domain of large graph such as *shortest path* queries might not be of direct interest in the scope of RDF/SPARQL domain and is thus, so far, not been considered.

d) Recently, some native graph database systems have been introduced (e.g., Neo4j[1], HypergraphDB[2]). These sys-

[1]http://neo4j.org/

[2]http://www.kobrix.com/hgdb.jsp

tems are mainly designed to provide efficient graph traversal functions[3]. However, these systems lack the support of *declarative* query interfaces and do not apply any query optimization strategies. In particular, they are language-specific and have their own APIs and low-level interfaces. Therefore, the efficiency of any graph query evaluation is *programmer-dependent*. Thus, it can turn to be quite inefficient in many cases especially when the programmer has no or little knowledge about the characteristics of the underlying graph.

In the following sections, we present our approach for querying large attributed graphs which aims to address the above mentioned challenges.

3. THE G-SPARQL QUERY LANGUAGE

The SPARQL query language is the official W3C standard for querying and extracting information from RDF graphs [14]. It is based on a powerful *graph matching* facility that allows the binding of variables to components in the input RDF graph. In principle, the RDF data model represents a special kind of the general model of attributed graph which represents our main focus in this paper. In particular, the main differences between the two kind of models (RDF and attributed graph) can be specified as follows:

a) In the RDF data model, graph edges are used for representing the structural relationships (graph topology) between the graph entities (connecting two vertices) in addition to representing the *graph data* by connecting the graph entities to the information of their attribute values (connecting a vertex with a literal value). Such uniform treatment leads to a significant increase in the size of the graph topology as the graph data is considered as a part of the topology and not as a separate part. The situation is different in the attributed graph model where the graph data (attributes of graph nodes/edges) are represented differently from the structural information of the graph.

b) In attributed graphs, edges are treated as first class citizens where any edge (similar to any vertex) can be described by an arbitrary set of attributes. That is not the case of the RDF model where there is no support for edges to be described by any attribute information (only vertices).

We introduce *G-SPARQL* as a SPARQL-like query language that employs the basic graph matching facilities of the SPARQL language. However, the language introduces new constructs that handle the above mentioned differences in the data model in addition to compensating the lack of some querying requirements that are not supported by the standard specification of the SPARQL language. In particular, our language aims to fulfill the following set of large graph querying requirements:

1) The language supports querying structural graph patterns where filtering conditions can be specified on the attributes of the graph vertices and/or edges which are participating in the defined patterns as well.

2) The language supports various forms for querying graph paths (sequence of edges) of possibly unknown lengths that connect the graph vertices. In particular, the language enables the expression of reachability queries and shortest path queries between the graph vertices where filtering conditions can be applied on the queried path patterns (e.g., constraints on the path length).

[3]A traversal refers to visiting the graph vertices sequentially by following the graph edges in some algorithmic fashion (e.g., depth-first or breadth-first)

```
        <Query> =   SELECT <VarList>
                    WHERE{ <Triple>+
                    [FILTER (<Predicate>)]*
                    [FILTERPATH (<PathPredicate>)]* }
      <VarList> =   {?var | <PathVar>}
      <PathVar> =   ??var | ?*var
       <Triple> =   <Term> (<Term> | <Edge> | <Path> ) <Term>
         <Term> =   literal | ?var
         <Edge> =   literal | literal+ | @literal | ?var(literal)
         <Path> =   <PathVar> | <PathVar>(literal)
    <Predicate> =   BooleanFunction
<PathPredicate> =   Length(<PathVar>, Predicate) |
                    AtLeastNode(<PathVar>, number, Predicate) |
                    AtMostNode(<PathVar>, number, Predicate) |
                    ALLNodes(<PathVar>, Predicate) |
                    AtLeastEdge(<PathVar>, number, Predicate) |
                    AtMostEdge(<PathVar>, number, Predicate) |
                    AllEdges(<PathVar>, Predicate)
```

Figure 2: *G-SPARQL* grammar.

Figure 2 shows the grammar of the *G-SPARQL* language whereas non-terminal `Query`, defining a *G-SPARQL* query, is the start symbol of this grammar. More details about the syntax and semantics of the *G-SPARQL* language are discussed in the following subsections.

3.1 Querying Attributes of Nodes and Edges

According to the standard specifications of the language, each SPARQL query defines a graph pattern P that is matched against an RDF graph G where each variable in the query graph pattern P is replaced by matching elements of G such that the resulting graphs are contained in G (pattern matching). The basic construct of building these graph patterns is the so-called a *Triple Pattern* [13]. A Triple Pattern represents an RDF triple `(subject, predicate, object)` where *subject* represents an entity (vertex) in the graph and *predicate* represents a relationship (edge) to an *object* in the graph. This object in the triple pattern can represent another entity (vertex) in the graph or a *literal* value. Each part of this triple pattern can represent either a constant value or a variable `(?var)`. Hence, a set of triple patterns concatenated by `AND (.)` represents the query graph pattern. The following example shows a simple SPARQL query that finds all persons who are affiliated at UNSW and are at least of 42 years old.

```
SELECT   ?X
 WHERE   {?X affiliatedAt UNSW. ?X age ?age.
          FILTER (?age >= 42).}
```

In our context, we need to differentiate between the representation of two types of query predicates. a) *Structural predicates*: specify conditions on the structural *relationship* between graph vertices. The *subject*, and *object* of the triple pattern refer to vertices and the *predicate* to an edge. b) *Value-based predicates*: specifies a condition on the value of an *attribute* of a graph element. The *subject* is either a vertex (or an edge as we explain below), the *predicate* is the attribute name, and the *object* is the attribute value.

Therefore, the *G-SPARQL* syntax uses the symbol `(@)` at the *predicate* part of the query triple patterns that represent value-based predicates and differentiate them from the standard structural predicates. To illustrate, let us consider the following example of two query triple patterns:

```
T1 --> ?Person affiliatedBy UNSW
T2 --> ?Person @age 42.
```

where $T1$ represents a structural predicate specifying that the graph vertices represented by the variable `?Person` are connected by an `affiliatedBy` edge to a vertex with the la-

bel UNSW. In contrast, $T2$ represents a value-based predicate that specifies the condition of having the vertices represented by the variable ?Person described by an age attribute storing the value 42.

Unlike the RDF data model, the model of attributed graphs enables describing each graph edge with an arbitrary set of attributes. Therefore, our query language enables representing two types of *value-based* predicates: 1) *Vertex predicates* which enables specifying conditions on the attributes of the graph vertices. 2) *Edge Predicates* which enables specifying conditions on the attributes of graph edges. In particular, we rely on the standard query triple pattern to represent both types of predicates. However, we use the round brackets () for the *subject* part of the query triple pattern to differentiate edge predicates. In these predicates, the subject parts refers to graph edges and not for graph vertices. Let us consider the following example of query triple patterns:

```
T3 --> ?Person ?E(affiliatedBy) UNSW
T4 --> ?E @Role "Professor"
T5 --> ?Person @officeNumber 518
```

where $T3$ represents a structural predicate that specifies the condition that the vertices represented by the variable ?Person is connected to a vertex with the label UNSW with an affiliatedBy relationship. $T4$ represents an *edge predicate* that determines that the *Role* attribute of the affiliatedBy relationship (where the edge representing the relationship is bound to the variable E) should store the value $Professor$. $T5$ represents a *vertex predicate* that specifies the condition that $Person$ is described by an officeNumber attribute that stores the value 518.

3.2 Querying Path Patterns

G-SPARQL supports expressing paths of arbitrary length and querying path patterns in two main ways. First, using explicit relationships, in compatible with the recent recommendation of the SPARQL 1.1 language[4], as described in the following triple patterns.

```
T5 --> ?Person knows+ John.
T6 --> ?Person knows+ ?X.
```

where $T5$ represents a structural predicate that describes a reachability test verifying that the vertices assigned to ?Person are connected to a vertex with label John by any path of knows edges. On the other hand, $T6$ assigns to the variable X all vertices that can be reached from the vertices that are represented by the variable ?Person through any path of knows edges. The symbol (+) indicates that the path can be of any length where each edge in the path needs to represent the relationship knows.

Second, *G-SPARQL* allows path variables in the *predicate* position of a triple pattern. In particular, it supports the following options for binding path variables in the path patterns.

```
T7 --> subject ??P object.
T8 --> subject ?*P object.
T9 --> subject ??P(predicate) object.
T10 --> subject ?*P(predicate) object.
```

where $T7$ binds the path variable P to the connecting paths between the two vertices of the subject and object. The symbol (??) indicates that the matching paths between the subject and object can be of any arbitrary length. In $T8$, the symbol (?*) indicates that the variable P will be matched with the *shortest* path between the two vertices of subject and object. $T9$ ensures that each edge in the matching paths represents the specified relationship predicate. Similarly, $T10$ ensures that each edge in the matched shortest path represents the relationship predicate.

In general, any two vertices can be connected with multiple paths. Therefore, *G-SPARQL* enables expressing filtering conditions that can specify *boolean* predicates on the nodes and the edges of the matching paths which are bound to the path variable. In particular, *G-SPARQL* supports the following filtering conditions over the matched paths.

a) Length(PV, P): verifies that the length (number of edges) of each matching path which is bound to the variable PV satisfies the predicate P and filters out those paths which do not satisfy the predicate P. For example, the following path filtering condition FilterPath (Length(??X, < 4)) ensures that the length of each path which is assigned to the path variable (X) is less than 4 edges.

b) At[Least|Most][Node|Edge](PV, N, P): verifies if at least (most) N number of nodes (edges) on each path which is bound to the variable PV satisfies the predicate P and filters out those paths which do not satisfy the predicate P.

c) All[Nodes|Edges](PV, P): ensures that every node (edge) of each path which is bound to the variable PV satisfies the predicate P.

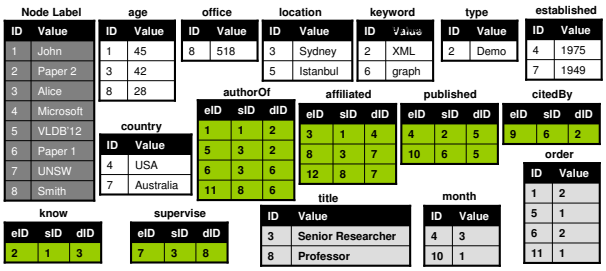


Figure 3: Relational Representation of Attributed Graph.

4. HYBRID GRAPH REPRESENTATION

Relational database systems are efficient in executing queries that benefit from indexing (e.g., B-tree) and query optimization techniques (e.g., selectivity estimation and join ordering). Relational systems are, however, inefficient in evaluating queries that require looping or recursive access to a large number of records by executing multiple expensive join operations which may yield a large intermediate result. Therefore, we rely on algorithms that execute on main memory data structures to answer graph queries that require extensive traversal operations on the graph topology. In particular, we use a hybrid representation for attributed graphs where the entire graph information (topology + data) is stored in a relational database, and only the graph topology information needs to be loaded to the main memory.

There are multiple approaches to store an attributed graph in a relational database. We adopt the fully decomposed storage model (DSM) [3] which is agnostic to the graph schema and therefore can be applied to any attributed graph. In addition, it permits efficient attribute retrieval during query processing. Figure 3 illustrates an example of our relational representation for the attributed graph in Figure 1. In particular, we start by assigning identifiers (IDs) to each vertex and edge in the graph. Vertex attributes are stored in M two-column tables, where M is the number of *unique* attributes of the graph vertices. Similarly, edge attributes

[4]http://www.w3.org/TR/sparql11-query/

Operator	Description	Relational
$\texttt{NgetAttVal}_{id,(attName):id,value}$	returns the values of an attribute for a set of nodes.	Yes
$\texttt{EgetAttVal}_{id,(attName):id,value}$	returns the values of an attribute for a set of edges.	Yes
$\texttt{getEdgeNodes}_{sID,[(eLabel)]:sID,eID,dID}$	returns adjacent nodes, optionally through a specific relation, for a set of graph nodes.	Yes
$\texttt{strucPred}_{sID,(eLabel),(dNLabel):sID,[eID]}$	returns a set of vertices that are adjacent to other vertices with a specific relationship and optionally returns the connecting edges.	Yes
$\texttt{edgeJoin}_{sID,dID,[(eLabel)]:sID,dID,[eID]}$	returns pairs of vertices that are connected with an edge, optionally of a specified relationship, and optionally returns the connecting edges.	Yes
$\texttt{pathJoin}_{sID,dID,[(eLabel)]:sID,dID,[pID,pRel]}$	returns pairs of vertices which are connected by a sequence of edges of any length, optionally with a specified relationship, and optionally returns connecting paths.	No
$\texttt{sPathJoin}_{sID,dID,[(eLabel)]:sID,dID,pID,pRel}$	returns pairs of vertices which are connected by a sequence of edges of any length, optionally with a specified relationship, and returns the *shortest* connecting path.	No
$\texttt{filterPath}_{pID,pRel,(cond):pID,pRel}$	returns paths that satisfy a condition.	No

Table 1: *G-SPARQL* algebraic operators.

are stored in N two-column tables where N is the number of *unique* attributes of the graph edges. The first column (ID) of a two-column attribute table stores the identifiers of those vertices/edges that are described by the associated attribute, and the second column ($Value$) stores the literal values for those attributes. For example, the vertex with label *Alice* is assigned ($ID = 3$) in table *nodeLabel* and the age attribute is stored in table *age* in row (3, 42). Each table is sorted as clustered index on the ID column in order to enable a fast merge join when multiple attributes of the same vertex/edge are retrieved. In addition, a secondary index on the $Value$ column is created for each table to reduce access costs for value-based predicates on the attributes.

The graph edges are stored in P three-column tables, where P is the number of *unique* relationships that exist between the graph vertices. These P three-column tables capture the graph *topology*. Each of these tables groups the information of all graph edges that represent a particular relationship. Each edge is described by (1) the edge identifier (eID), (2) the identifier of the source vertex (sID), and (3) the identifier of the destination vertex (dID).

In our hybrid graph representation, we rely on a native pointer-based data structure for representing the graph topology information in main memory. In particular, this memory representation of the graph topology encodes the information of the P relationship tables that store the *structural* information of the graph edges. In principle, this information is needed for executing the index-free and memory-based algorithms that involve heavy traversal operations on the graph topology and for recursive algorithms. Example algorithms include Dijkstra's algorithm to obtain the shortest path between two vertices or performing a breath-first search (BFS) to answer reachability queries [2]. Therefore, our hybrid representation achieves a clear reduction in main memory data structures as we show in Table 2 in Section 6.

5. QUERY EXECUTION ENGINE

This section presents the query compilation and optimization process that effectively translates a *G-SPARQL* query into a physical execution plan on our hybrid graph representation. In general, one of the key effective query optimization techniques for any query language is the availability of a powerful algebraic compilation and rewriting framework of logical query plans. In our approach, we rely on a dialect of *tuple algebra* for compiling *G-SPARQL* queries. In particular, our algebra considers *tuples* as the basic unit of information where each algebraic operator manipulates collections of tuples [11, 10]. Hence, we can leverage the well-established relational query planning and optimization techniques in several venues before further translating our query plans (or parts of them) into SQL queries. However, our logical algebra extends the set of traditional relational algebraic operators (e.g., selection, projection, join) with a set of logical operators that are capable of expressing complex *G-SPARQL* operations that can not be matched with the semantics of the traditional relational operators.

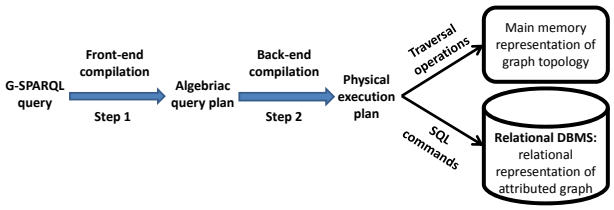


Figure 4: Execution Steps of a *G-SPARQL* query.

As shown in Figure 4, the compilation process consists of two main steps : (1) *Front-end compilation* which translates the input *G-SPARQL* query into an algebraic query plan expressed using the intermediate language. (2) *Back-end compilation* which translates the algebraic query plan to the physical execution plan on the hybrid engine. We first present the algebraic operators which are used in describing our algebraic query plans, and then we describe the steps of the query compilation process.

5.1 Algebraic Operators

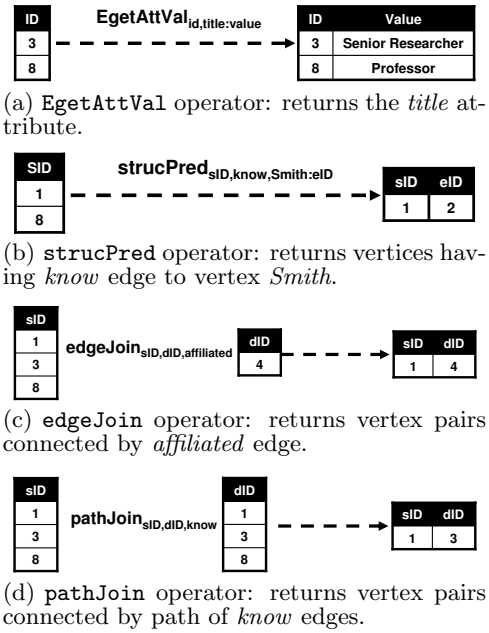
In general, the design of our logical operators are independent of any specific disk or memory representation of the attributed graph. In addition, they are independent of the underlying query evaluation engine. The descriptions of our algebraic operators are listed in Table 1. For example, the `NgetAttVal` is a unary operator which is used for retrieving the values of a specific attribute for a set of graph nodes. The operator receives a set of tuples where the column (id) of the input relation identifies the graph nodes and the name of the attribute to be accessed ($attName$). The schema of the output tuples extends the schema of the input tuples with the ($value$) column that represent the values of the accessed attribute. Similarly, the `EgetAttVal` operator retrieves the values of a specific attribute for a set of graph edges. The traditional relational `Selection` operator (σ_p) is used for representing value-based predicates over the values of the attributes of graph nodes or edges. It selects only those tuples of an input relation for which a value-

based predicate (p) over a specific column holds. Hence, it represents the right match for reflecting the expressivity of the SPARQL `FILTER` expressions. Based on the attributed graph of Figure 1 and its relational representation in Figure 3, Figure 5(a) illustrates an example for the behavior of the `EgetAttVal` operator where it retrieves the values of the `title` attribute for an input relation with the (id) of two graph edges. The schema of the output relation extends the schema of the input relation with an attribute that stores the value of the accessed attribute.

The `getEdgeNodes` is a unary operator which is used for retrieving a set of adjacent nodes. The operator receives a set of tuples where the column (id) of the input relation identifies the graph nodes and *optionally* a specified relation for accessing the adjacent nodes ($eLabel$). The schema of the output tuples extends the schema of the input tuples with the two columns that represent the identifiers of the connecting edges (eID) and the adjacent nodes (dID). If the operator receives the ($eLabel$) parameter then it filters out the nodes that do not have adjacent nodes connected with the specified relationship. The `strucPred` is another unary operator which is used for filtering a set of nodes based on a specified structural predicate. It receives a set of tuples where the column (sID) of the input relation identifies the graph nodes and a structural predicate which is described by the label of the connecting relation ($eLabel$) and the label for the adjacent node that should be accessed through this relation ($dNLabel$). Figure 5(b) illustrates an example of the `strucPred` operator where it applies a structural predicate which filters out the graph vertices that are not connected to an adjacent vertex with the label `Smith` through the `know` relationship and projects the information of the connecting edges that represent the structural predicate.

The `edgeJoin` is a binary join operator which receives two relations (S and D) where the two columns (sID) and (dID) identify the graph nodes of S and D, respectively. The operator checks for each pair of nodes whether it is connected with any graph edge, filters out the not connected pairs and returns the tuples of the connected pairs as a result. The output of the operator is a single relation where the schema of the output tuples concatenates the columns of (S and D). The operator can receive an optional parameter which imposes a condition on the connecting edge between each pair of nodes to be representing a specified relationship label ($eLabel$). Figure 5(c) illustrates another example of the `edgeJoin` operator where it receives two sets of graph vertices - (`"John"`,`"Alice"`,`"Smith"`) and (`"Microsoft"`) - and returns pairs of graph vertices that are connected through an `affiliated` relationship. Moreover, the `edgeJoin` operator can optionally project the information of the connecting edge(s) where it extends the schema of the output relation by an additional column (eID) that represents the identifiers of the connecting edges between each pair of nodes.

The `pathJoin` operator is another binary join operator which receives two relations (S and D) where the two columns (sID) and (dID) identify the graph nodes of S and D, respectively. The operator checks for each pair of nodes whether it is connected by a sequence of edges (of any length), filters out the not connected pairs and returns the tuples of the connected pairs as a result. The operator can receive an optional parameter which imposes a condition on the edges of each connecting path between each pair of nodes to be representing a specified relationship ($eLabel$). Moreover, the



(a) `EgetAttVal` operator: returns the *title* attribute.

(b) `strucPred` operator: returns vertices having *know* edge to vertex *Smith*.

(c) `edgeJoin` operator: returns vertex pairs connected by *affiliated* edge.

(d) `pathJoin` operator: returns vertex pairs connected by path of *know* edges.

Figure 5: Examples of the *G-SPARQL* Operators.

`pathJoin` operator can optionally project the information of the connecting path(s) as follows:

1) It extends the schema of the input relation by an additional column (pID) that represents an assigned identifier for each connecting edge between each pair of nodes. It should be noted that each pair of nodes can be connected with multiple paths. Therefore, each input pair of nodes can have multiple representing tuples that describes the information of the bound paths.

2) It returns another output relation ($pRel$) which describes the information of the resulting paths where each path is described by a sequence of tuples that represent the nodes and edges constituting the path in an orderly manner. The value of a path variable in the query output is represented by a serialization of the ($Label$) information of its associated tuples in this relation according to their ascending ($order$).

Figure 5(d) illustrates an example of the `pathJoin` operator where it receives two sets of graph vertices, (`"John"`, `"Alice"`, `"Smith"`) and (`"John"`, `"Alice"`, `"Smith"`), returns pairs of graph vertices that are connected through a sequence of any length of `know` relationships and projects the information of the resulting connecting paths. The `sPathJoin` operator works in the same way as the `pathJoin` operator with only one difference being that it returns a *single* path that represents the *shortest* connection between each pair of nodes (if exist a connection). The `filterPath` is a binary operator which receives two relations (R and $pRel$) where the column (pID) of the relation (R) represents the path identifiers that have their associated description information represented by the relation ($pRel$). The operator returns the relation (R) where the tuples which have paths (pID) with information ($pRel$) that do not fulfill the condition ($cond$) are filtered out. The ($cond$) parameter represents one of the path filtering conditions which was previously described in Section 3.2. For more examples of the behavior of our algebraic operators, we refer to our technical report [16].

$\texttt{NgetAttVal}_{nodeID,(attName):value} \Rightarrow \pi_{R.*,attName.value}(R \bowtie_{R.nodeID=attName.ID} attName)$
$\texttt{EgetAttVal}_{edgeID,(attName):value} \Rightarrow \pi_{R.*,attName.value}(R \bowtie_{R.edgeID=attName.ID} attName)$
$\texttt{getEdgeNodes}_{sID,(eLabel):eID,dID} \Rightarrow \pi_{R.*,eLabel.eID,eLabel.dID}(R \bowtie_{R.sID=eLabel.sID} eLabel)$
$\texttt{strucPred}_{sID,(eL),(dNL):eID} \Rightarrow \pi_{R.*,eL.eID}(\sigma_{nodeLabel.Value=dNL}((R \bowtie_{R.sID=eL.SID} eL) \bowtie_{eL.dID=nodeLabel.ID} nodeLabel))$
$\texttt{edgeJoin}_{R.sID,S.dID:eID} \Rightarrow \pi_{R.*,S.*,allEdges.eID}((R \bowtie_{R.sID=allEdges.SID} allEdges) \bowtie_{allEdges.dID=S.dID} S)$
$\texttt{edgeJoin}_{R.sID,S.dID,(eLabel):eID} \Rightarrow \pi_{R.*,S.*,eLabel.eID}((R \bowtie_{R.sID=eLabel.SID} eLabel) \bowtie_{eLabel.dID=S.dID} S)$

Figure 6: Relational Representation of *G-SPARQL* Algebraic Operators.

As shown in Table 1, not all of our algebraic operators can be represented by the standard relational operators. Based on our relational representation of the attributed graphs, Figure 6 depicts the mappings for those operators that can be translated into a pattern of standard relational operators. Since the semantics of the operators `getEdgeNodes` and `edgeJoin` can be not restricted by a specified relationship (*eLabel*), compiling these operators using the standard relational operators requires joining the input relation(s) with each of the *relation* tables, separately, and then union all the results. To simplify, we have created a materialized view (`allEdges`) that represents such union of all *relation* tables. For the SQL translation templates of our algebraic operators, we refer to our technical report [16].

5.2 Query Compilation

Front-end Compilation. In this step of our compilation process, we start by assigning for each query triple pattern, a mapping onto algebraic operators. Figures 7 illustrates examples of the inference rules for mapping the *G-SPARQL* query triple patterns into our algebraic operators. (the full list of the inference rules are available in our technical report [16]). A sample interpretation of the inference rule OpMap-1 is that it maps a query triple pattern `(?var, @attName, ?var2)` of query q into the algebraic operator ($\texttt{NgetAttVal}_{ID,(attName):value}$) where the variable `?var2` is bound to the column `value` (`Col(?var2)` $\equiv$ `value`) of the output relation from applying the `NgetAttVal` operator *given that* the mapping of the variable `?var` (`Map(?var)`) is bound to the column `ID` as a part of the input relation `R`. Figure 8 illustrates an example algebraic compilation for the following *G-SPARQL* query:

```
SELECT   ?L1 ?L2
 WHERE   {?X @label ?L1.  ?Y @label ?L2.
          ?X @age ?age1.  ?Y @age ?age2.
          ?X affiliated UNSW. ?X livesIn Sydney.
          ?Y ?E(affiliated) Microsoft.  ?E @title "Researcher".
          ?X ??P ?Y.
          FILTER(?age1 >= 40).  FILTER(?age2 >= 40)}
```

During this compilation step, a set of query rewriting rules is applied in order to optimize the execution time of the query evaluation. In addition, this compilation step uses three main heuristics to reorder the query triple patterns according to their *restrictiveness* (the more restrictive pattern has higher precedence) in order to optimize the join order for the generated SQL statments based on the following rules. Let `t1`, `t2` $\in$ `Triple(q)` be two triple patterns of query `q`.

1) `t1` is defined as less restrictive than `t2` (`t1` $\gg$ `t2`) if `t1` contains more number of path variables (`??` or `?*`) than `t2`.

2) `t1` is defined as more restrictive than `t2` (`t1` $\ll$ `t2`) if `t1` contains less number of variables than `t2`.

3) `t1` is defined as more restrictive than `t2` (`t1` $\ll$ `t2`) if `t1` has the same number of variables than `t2` and the number of filter expressions over the variables of `t1` is more than the number of filter expressions over the variables of `t2`.

Back-end Compilation. The second compilation step is specific to our hybrid memory/disk representation of attributed graphs where we start by mapping the operators of the plan to their relational representation, when applicable (Figure 6), then we start optimizing the algebraic plans using a set of rules. These rules includes the traditional rules for relational algebraic optimization (e.g., pushing the selection operators down the plan) in addition to some rules that are specific to the context of our algebraic plans. In particular, the main strategy of our rules is to push the non-standard algebraic operators (with memory-based processing) above all the standard relational operators (that can be pushed inside the relational engine) in order to delay their execution (which is the most expensive due to its recursive nature) to be performed after executing all data access and *filtering* operations that are represented by the standard relational operators. At the execution level, the basic strategy of our query processing mechanism is to push those parts of query processing that can be performed independently into the underlying RDBMS by issuing SQL statements [9]. In particular, our execution split mechanism makes use of the following two main heuristics:

$$\frac{\texttt{Map(?var)} \in \texttt{R} \quad \texttt{Col(?var)} \equiv \texttt{ID}}{\texttt{(?var, @attName, ?var2)} \Rightarrow \quad \texttt{NgetAttVal}_{ID,(attName):value}\,\texttt{(R)} \quad \texttt{Col(?var2)} \equiv \texttt{value}} \text{ (OpMap-1)}$$

$$\frac{\texttt{Map(?var)} \in \texttt{R} \quad \texttt{Col(?var)} \equiv \texttt{sID}}{\texttt{(?var, eLabel, dNLabel)} \Rightarrow \texttt{strucPred}_{sID,(eLabel),(dNLabel)}\,\texttt{(R)}} \text{ (OpMap-2)}$$

$$\frac{\begin{array}{c}\texttt{(?E, predicate, object)} \in \texttt{Triple(q)} \\ \texttt{Map(?var)} \in \texttt{R} \quad \texttt{Col(?var)} \equiv \texttt{sID} \\ \texttt{Map(?var2)} \in \texttt{S} \quad \texttt{Col(?var2)} \equiv \texttt{dID}\end{array}}{\texttt{(?var, ?E(eLabel), ?var2)} \Rightarrow \quad \texttt{(R) edgeJoin}_{sID,dID,(eLabel):eID}\,\texttt{(S)} \quad \texttt{Col(?E)} \equiv \texttt{eID}} \text{ (OpMap-3)}$$

$$\frac{\begin{array}{c}\texttt{Map(?var)} \in \texttt{R} \quad \texttt{Col(?var)} \equiv \texttt{sID} \\ \texttt{Map(?var2)} \in \texttt{S} \quad \texttt{Col(?var2)} \equiv \texttt{dID}\end{array}}{\texttt{(?var, ??P, ?var2)} \Rightarrow \quad \texttt{(R) pathJoin}_{sID,dID}\,\texttt{(S)}} \text{ (OpMap-4)}$$

Figure 7: Example of *G-SPARQL* Operator Mapping Rules.

1) Relational databases are very efficient for executing queries that represent structural predicates or value-based predicates on the graph attributes (vertices or edges) due to its powerful indexing mechanisms and its sophisticated query optimizers. In addition, relational databases are very efficient on finding the most efficient physical execution plan including considering different possible variants such as different join implementations and different join orderings.

2) Relational databases are inefficient for executing queries with operators of a recursive nature (e.g., path patterns). Main memory algorithms are much faster for evaluating such types of operators which require heavy traversal operations over the graph topology.

As shown in Figure 8, our algebraic plans come in a DAG shape. Therefore, we perform the translation of these plans into SQL queries by traversing the algebraic plan in a bottom-up fashion (starting from the leaves and then climb the different paths back to the root) using a set of defined pattern-based translation rules [9]. This climbing process for each path stops if it hits one of the operators that does not have

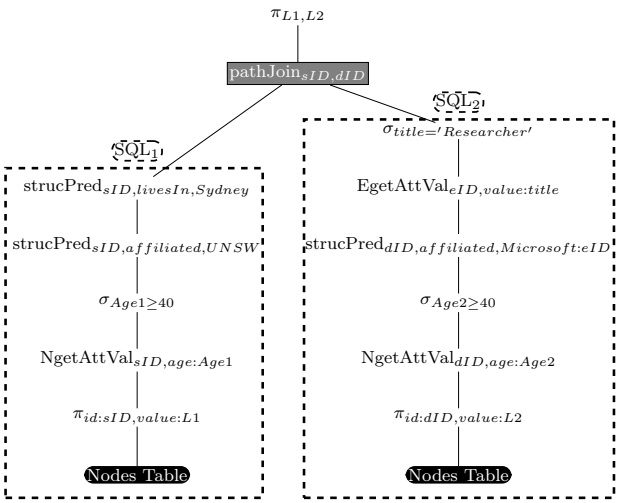


Figure 8: Front-end compilation of a G-SPARQL query into an algebraic plan.

a standard relational representation for its semantics or if it reaches the root. Each generated SQL query is tempting to simply then rely on the underlying relational backends for the physical optimization and processing. For example, in Figure 8, all operators can be translated into standard relational operators except of the `pathJoin` operator (filled with gray color). In this example, as indicated by dashed rectangles in the figure, two SQL queries are generated (SQL_1 and SQL_2) where the results of these queries are then passed for further memory-based processing using the `pathJoin` operator and the following operators in the plan.

The main implementation of our query evaluation engine relies on index-free main memory algorithms for evaluating reachability and shortest path operators [2]. However, our algebraic compilation approach remains agnostic to the physical execution of its logical operator and can make use of any available indexing information for accelerating the query evaluation process of the different types of queries taking into consideration the trade-off of building and maintaining their indices in addition to their main memory consumption.

6. EXPERIMENTAL EVALUATION

Implementation: We implemented a native pointer-based memory representation of the graph topology in addition to the Dijkstra and BFS algorithms using C++. We used IBM DB2 RDBMS for storage, indexing and performing all SQL queries. In order to measure the relative effectiveness of our query split execution mechanism, we compared the performance results of our approach with the performance of the native graph database system, Neo4j (version 1.5 GA). Neo4j is an open source project which is recognized as one of the foremost graph database systems. According to the Neo4j website, "*Neo4j is a disk-based, native storage manager completely optimized for storing graph structures for maximum performance and scalability*". It has an API that is easy to use and provides powerful traversal framework that can implement all queries which can be expressed by *G-SPARQL*. Neo4j uses Apache Lucene for indexing the graph attributes. We conducted our experiments on a PC with 3.2 GHz Intel Xeon processors, 8 GB of main memory storage and 500 GB of SCSI secondary storage. It should be noted

	Vertices	Edges	Attribute Values	Topology Information
Small	126,137	297,960	610,302	9%
Medium	242,074	761,558	1,687,465	11%
Large	825,433	3,680,156	7,336,899	12%

Table 2: Characteristics of real datasets.

that because of the expressiveness of our language, we were unable to consider either the traditional RDF query processors [12, 7] (they do not support path patterns, reachability or shortest path queries) or traditional structural graph indexing and querying techniques [4, 8, 20, 19, 21] (each of them supports only one specific type of graph queries and do not consider querying the attributes of the graph vertices or edges) as options for our performance comparison.

Dataset: We used the ACM digital library dataset (which includes the information of all ACM publications till September 2011) to construct the attributed graph. The graph vertices represent 8 different types of entities (e.g., *author*, *article*, *conference*), 12 different types of relationships between the graph entities (e.g., *authorOf*, *citedBy*, *partOfIsuue*), and a total of 76 unique attributes, of which 62 attributes are describing the graph vertices and 14 attributes are describing the graph edges. In our experiments, we used 3 scaling sizes of graph subsets (small, medium and large) in order to test the scalability of our approach. Table 2 lists the characteristics of the three sets. For more experiments on synthetic graphs, we refer to our technical report [16].

Query Workload: Our query workload consists of 12 query templates (Figure 9) where we used random literal values to generate different query instances. The queries are designed to cover the different types of the triple patterns that are supported by *G-SPARQL*. We refer to our technical report [16] for detailed descriptions and the algebraic plans of our query templates. As we have previously described, the efficiency of execution for any graph query using Neo4j is programmer-dependent and each query template can have different ways of implementations using Neo4j APIs. In our experiments, for each query template, we created two associated Neo4j implementations. The first implementation is an optimized version that considers a pre-known knowledge about the result size of each query step (triple pattern) while the second version is a non-optimized one that does not consider this knowledge. Each query template is instantiated 20 times where the data values are generated randomly.

Query Evaluation Times: The average query evaluation times for the 20 instances of each of the 12 query templates are shown in Figure 10 for the small (Figure 10(a)), medium (Figure 10(b)) and large (Figure 10(c)) graphs. As has been well recognized in conventional query processing, a good query plan is a crucial factor in improving the query performance by orders of magnitude. The results of the experiments show that our approach is on average 3 times faster than the Neo4j non-optimized implementations of the query workload on the small subset, 4 times faster on the medium subset and 5 times faster on the large subset of the experimental graph. In particular, our approach outperforms the Neo4j non-optimized implementations in each of the defined query templates. The results of the experiments also show that the average query evaluation times of our approach is 17% faster than the Neo4j optimized implementations on the small subset, 22% faster on the medium subset and 28% faster on the large experimental graph. The

Q1	SELECT ?Name1 ?Name2 WHERE {?X @name ?Name1. ?Y @name ?Name2. ?X @prolific "High". ?Y @prolific "High". ?X @affilation "%affiliation1%". ?Y @affilation "%affiliation2%". ?X ?E(co-author) ?Y. ?E @noPapers ?NO. FILTER (?NO >= 2).}
Q2	SELECT ?Name1 ?Name2 WHERE {?X @name ?Name1. ?Y @name ?Name2. ?X @prolific "High". ?Y @prolific "High". ?X @affilation "%affiliation1%". ?Y @affilation "%affiliation2%". ?X co-author+ ?Y.}
Q3	SELECT ?Name1 ?Name2 ??P WHERE {?X @name ?Name1. ?Y @name ?Name2. ?X @prolific "High". ?Y @prolific "High". ?X @affilation "%affiliation1%". ?Y @affilation "%affiliation2%". ?X ??P(co-author) ?Y.}
Q4	SELECT ?Name1 ?Name2 ?*P WHERE {X @name ?Name1. Y @name ?Name2. ?X @prolific "High". ?Y @prolific "High". X @affilation "%affiliation1%". Y @affilation "%affiliation2%". ?X ?*P(co-author) ?Y.}
Q5	SELECT ?Name1 ?Name2 ??P WHERE {X @name ?Name1. Y @name ?Name2. ?X @prolific "High". ?Y @prolific "High". X @affilation "%affiliation1%". Y @affilation "%affiliation2%". ?X ??P ?Y.}
Q6	SELECT ?Name1 ?Name2 ??P WHERE {?X @name ?Name1. ?Y @name ?Name2. ?X @prolific "High". ?Y @prolific "High". ?X @affilation "%affiliation1%". ?Y @affilation "%affiliation2%". ?X ??P(co-author) ?Y. FilterPath(Length(??P, <= 3)).}
Q7	SELECT ?Name1 ?Name2 ??P WHERE {?X @name ?Name1. ?Y @name ?Name2. ?X @prolific "High". ?Y @prolific "High". ?X @affilation "%affiliation1%". ?Y @affilation "%affiliation2%". ?X ??P(co-author) ?Y. FilterPath(AllNodes(??P, @prolific "High")).}
Q8	SELECT ?Name ?Title ?Year WHERE {?X @name ?Name. ?X @prolific "High". ?X ?E(authorOf) ?Paper. ?E @seqNo 1. ?Paper @title ?Title. ?Paper @keyword "%keyword%". ?Paper partOf ?Issue. ?Issue @year ?Year. ?Issue issueOf ?Journal. ?Journal @code "%code%".}
Q9	SELECT ?Title ?Name1 ?Name2 ?Name3 WHERE {?P @title Title. ?P @keyword "%keyword%". ?P citedBy ?P1. ?P citedBy ?P2. ?P citedBy ?P3. Au1 authorOf ?P1. ?Au2 authorOf ?P2. ?Au3 authorOf ?P3. ?AU1 @prolific "High". ?AU2 @prolific "High". ?AU3 @prolific "High". ?Au1 @name ?Name1. ?Au2 @name ?Name2. ?Au3 @name ?Name3.}
Q10	SELECT ?T1 ?T2 WHERE {?X @name ="%name%". ?X authorOf ?Paper1. ?Paper1 @Title ?T1. ?Paper2 @Title ?T2. ?Paper1 @Keyword "%keyword%". ?Paper2 @Keyword "%keyword%". ?Paper1 ??P(citedBy) ?Paper2. FilterPath(Length(??P, <= 2)).}
Q11	SELECT ?T1 ?T2 WHERE {?X @name ="%name%". ?X authorOf ?Paper1. ?Paper1 @Title ?T1. ?Paper2 @Title ?T2. ?Paper1 @Keyword "%keyword%". ?Paper2 @Keyword "%keyword%". ?Paper1 ??P(citedBy) ?Paper2. FilterPath(Length(??P, <= 4)). FilterPath(AllEdges(??P, @source "Ext")).}
Q12	SELECT ?Name WHERE {?X @name ?Name. ?Y @name "%name1%". ?Z @name "%name2%". ?X ??E1(co-author) ?Y. ?X ??E2(co-author) ?Z. FilterPath(Length(??E1, <= 2)). FilterPath(Length(??E2, <= 2)). FilterPath(AllEdges(??E1, @noPapers >2)).}

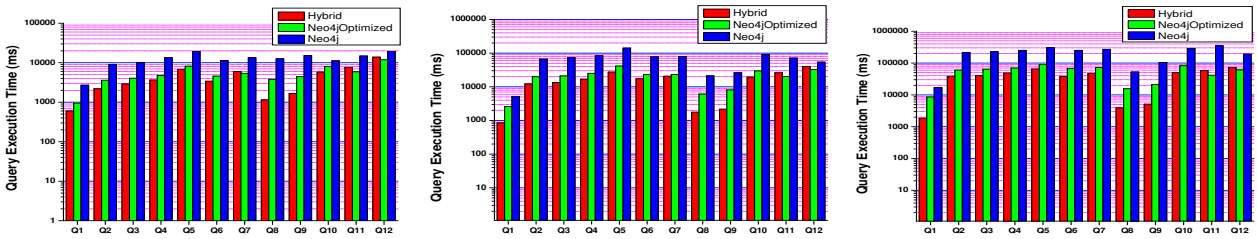
Figure 9: Query templates of our experimental workload.

Neo4j optimized implementations outperforms our approach in 2 of the 12 query templates (Q11 and Q12) while our approach performs better in the rest of the queries.

In general, the well-known maturity of the indexing and built-in optimization techniques of physical query evaluation (e.g., join algorithms) provided by the underlying RDBMs play the main role of outperforming Neo4j optimized implementations. For example, the performance of relational partitioned B-tree indices outperforms the performance of the indexing service of Neo4j that uses Lucene which is more optimized for full-text indexing rather than traditional data retrieval queries. To better understand the reasons for the differences in performance between our approach and optimized Neo4j implementations, we look at the performance differences for each query. Q1, Q8 and Q9 are pattern matching queries that involve structural predicates in addition to value-based predicates on the attributes of the graph nodes and edges. In our approach, the whole executions of these queries can be pushed inside the underlying RDBMS. Relational engine has shown to be more efficient than Neo4j as a native graph engine in performing such pattern matching queries that purely relies on the efficiency of the underlying physical execution properties of the engine, mainly physical indices and join algorithms, and do not involve any recursive operations. The scalability feature of the relational database engine is also shown by the increasing percentage of improvement for these queries over Neo4j with the increasing size of the underlying graph size. For example, for Q8, the relational execution is 3.3 times faster than the optimized Neo4j execution for the small subset of the experimental graph graph while it is 4.2 times faster on the large subset of the experimental graph.

Q2, Q3, Q4, Q5, Q6 and Q7 are queries that involve recursive join operations between two filtered set of vertices. In particular, all of these queries are seeking for two sets of authors where each set is filtered based on the `prolific` and `affiliation` attributes. Q2 verifies for each pair of vertices (one from each filtered set) whether they are connected by a sequence of edges of any length where all edges of the connecting path represent the `co-author` relationship. Assuming the numbers of vertices in the first and second sets are equal to M and N respectively, then the number of verification operations (represented by the `pathJoin` operator) equals $M * N$. Q3 is similar to Q2 but it returns additional information about the connecting paths between each pair of vertices ($\texttt{pathJoin}_{sID,dID,co-author:eID}$). That is why Q3 is slightly more expensive than Q2. Q4 is again similar to Q3. However, it only returns the information of the *shortest path* between each pair of vertices ($\texttt{sPathJoin}_{sID,dID,co-author:eID}$). Evaluating the *shortest path* over the graph topology is also slightly more expensive than the general reachability verification (Q2 and Q3). Q5 represents a more expensive variant of Q2 that generalizes the reachability verification test for each pair of vertices so that they can be connected by a sequence of edges of any length where each edge in the connecting path can represent any relationship ($\texttt{pathJoin}_{sID,dID:eID}$). Q6 and Q7 extend Q3 by adding filtering conditions on the connecting paths between each pair of vertices (the `filterPath` operator). In particular, Q6 filters out the paths with more than 3 edges. Q7 verifies that the `author` vertices for each of the resulting paths between each pair of vertices are highly `prolific`. Our hybrid approach outperforms the optimized Neo4j implementations for all of these queries by splitting the execution of the query plans between the underlying relational engine and the available topology information in the main memory. In particular, it leverages the efficiency of the relational engine for retrieving each set of vertices, utilizes memory topology information for fast execution of the required traversal operations and avoids loading unnecessary information that are not involved in evaluated queries.

In our approach, the execution of the path filtering condi-



(a) Small Graph Size (b) Medium Graph Size (c) Large Graph Size

Figure 10: Average query evaluation times of real datasets.

tion of `Q7` is an expensive operation as it can only be applied in a post-processing step after determining all the connecting paths between each pair of vertices. This post-processing step needs to issue an SQL statement that retrieves the values of the `prolific` attributes for all nodes of the connecting paths as they are not available in the topology information which are loaded onto the main memory and then filters out all paths that contains any node that does not satisfy the filtering condition over the retrieved attribute values. On the contrary, the post-processing execution of the path filtering condition of `Q6` (based on path length) is rather cheap as it does not need to retrieve any data from the underlying database during the memory-based processing.

`Q10` and `Q11` are another two queries that involve recursive join operations between two filtered set of vertices. In particular, `Q10` is seeking for two sets of papers where both sets are filtered based on the same value-based predicate on the `keyword` attribute and one of the sets is further filtered to only those papers that are authored by a specific author. `Q10` verifies for each pair of vertices whether they are connected by a path with a length that is less than or equal to 4 edges where all edges represent the `citedBy` relationship. `Q11` extends `Q10` by further filtering the connecting paths to only those paths where all of their edges are described as `external` on their `source` attribute. While our approach is faster on evaluating `Q10`, Neo4j is faster for evaluating `Q11`. The main reason behind this is the expensive cost of retrieving the `external` attribute of the edges of the connecting paths for further filtering them. Optimized Neo4j implementation outperforms our approach in `Q12` as well where it needs to ensure that all edges of the connecting paths are described by `noPapers` greater than 2. One approach to overcome this limitation in our approach is to load onto the main memory, the frequently used attributes in path filtering conditions in addition to the graph topology. For example, for our query workload by loading the edge attributes `source` and `noPapers`, the performance of our approach for queries `Q11` and `Q12` is improved by an average of 31% and thus we can outperform the Neo4j optimized implementation. Obviously, there is a trade-off between the memory consumption and the performance that can be gained on evaluating the path filtering conditions by loading more attributes of the graph nodes/edges. However, determining which attributes should be loaded onto the memory would require pre-known knowledge about the characteristics of the query workloads.

7. CONCLUSIONS

We presented *G-SPARQL*, a novel language for querying large attributed graphs. The language supports querying structural graph patterns where filtering conditions can be specified on the attributes of the graph vertices/edges. In addition, it supports various forms for querying and conditionally filtering path patterns. We presented an efficient hybrid Memory/Disk graph representation where only the topology of the graph is maintained in memory while the data of the graph are stored in a relational database. We developed an algebraic compilation technique for our execution engine with a split of execution mechanism for the generated query plans. Experimental studies on real graphs validated the efficiency and scalability of our approach.

8. REFERENCES

[1] J. Cheng and J. Yu. On-line exact shortest distance query processing. In *EDBT*, 2009.
[2] T. Cormen, R. Rivest, C. Leiserson, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.
[3] D. Abadi et al. Scalable Semantic Web Data Management Using Vertical Partitioning. In *VLDB*, 2007.
[4] E. Cohen et al. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.*, 32(5), 2003.
[5] H. Tong et al. Fast best-effort pattern matching in large attributed graphs. In *KDD*, 2007.
[6] L. Zou et al. Answering pattern match queries in large graph databases via graph embedding. *VLDB J.*, 32(5), 2011.
[7] L. Zou et al. gStore: Answering SPARQL Queries via Subgraph Matching. *PVLDB*, 4(8), 2011.
[8] R. Jin et al. 3-HOP: a high-compression indexing scheme for reachability query. In *SIGMOD*, 2009.
[9] T. Grust, M. Mayr, J. Rittinger, S. Sakr, and J. Teubner. A SQL: 1999 code generator for the pathfinder XQuery compiler. In *SIGMOD*, 2007.
[10] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. FERRY: database-supported program execution. In *SIGMOD*, 2009.
[11] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *VLDB*, 2004.
[12] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1), 2008.
[13] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *TODS*, 34(3), 2009.
[14] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF, W3C Recommendation.
[15] S. Sakr. GraphREL: A Decomposition-Based and Selectivity-Aware Relational Framework for Processing Sub-graph Queries. In *DASFAA*, 2009.
[16] S. Sakr, S. Elnikety, and Y. He. G-SPARQL: A Hybrid Query Engine for Querying Large Attributed Graphs. Technical Report MSR-TR-2011-138, Microsoft Research. http://research.microsoft.com/apps/pubs/?id=157417.
[17] M. Sarwat, S. Elnikety, Y. He, and G. Kliot. Horton: Online Query Execution Engine For Large Distributed Graphs . In *ICDE*, 2011.
[18] F. Wei. TEDI: efficient shortest path query answering on graphs. In *SIGMOD*, 2010.
[19] X. Yan, P. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *SIGMOD*, 2004.
[20] S. Zhang, M. Hu, and J. Yang. TreePi: A Novel Graph Indexing Method. In *ICDE*, 2007.
[21] P. Zhao and J. Han. On Graph Query Optimization in Large Networks. *PVLDB*, 3(1), 2010.