

Hybrid query execution engine for large attributed graphs



Sherif Sakr ^{a,*}, Sameh Elnickety ^b, Yuxiong He ^b

^a National ICT Australia, UNSW, Sydney, Australia

^b Microsoft Research, Redmond, WA, United States

ARTICLE INFO

Article history:

Received 19 December 2012

Received in revised form

26 October 2013

Accepted 29 October 2013

Recommended by: K.A. Ross

Available online 12 November 2013

Keywords:

Data models

Query

Access methods

Graphs

Graph queries

SPARQL

ABSTRACT

Graphs are widely used for modeling complicated data such as social networks, bibliographical networks and knowledge bases. The growing sizes of graph databases motivate the crucial need for developing powerful and scalable graph-based query engines. We propose a SPARQL-like language, G-SPARQL, for querying attributed graphs. The language enables the expression of different types of graph queries that are of large interest in the databases that are modeled as large graph such as pattern matching, reachability and shortest path queries. Each query can combine both structural predicates and value-based predicates (on the attributes of the graph nodes/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 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 are 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 (using SQL) while the execution of other parts of the query plan is processed using memory-based algorithms, as necessary. Experimental results on real and synthetic datasets demonstrate the efficiency and the scalability of our approach and show that our approach outperforms native graph databases by several factors.

© 2013 Published by Elsevier Ltd.

1. Introduction

Graphs are popular data structures which are used to model structural relationship between objects. Recently, graph query processing has attracted a lot of attention from the database research community due to the increasing popularity of graph databases in various application domains. In general, existing research on graph databases and graph query processing can be classified into two main categories. The first category represents graph databases which consist of a large number of small graphs (usually called *Transactional Graph Database*)

such as bioinformatic applications [28], cheminformatics applications [31] and repositories of business process models [46]. In this category, there are two types of queries that are commonly studied in the literature:

- Subgraph query* which aims to find all the graphs in the database such that a given query graph is a subgraph of them [58,59].
- Supergraph query* that aims to find all the graphs in the database which are subgraphs of the given query graph [11,60].

The second category of graph databases is usually represented as one (or a very small number) of the large graphs such as social networks [8], bibliographical networks [56]

* Corresponding author.

E-mail addresses: ssakr@cse.unsw.edu.au (S. Sakr), samehe@microsoft.com (S. Elnickety), yuxhe@microsoft.com (Y. He).

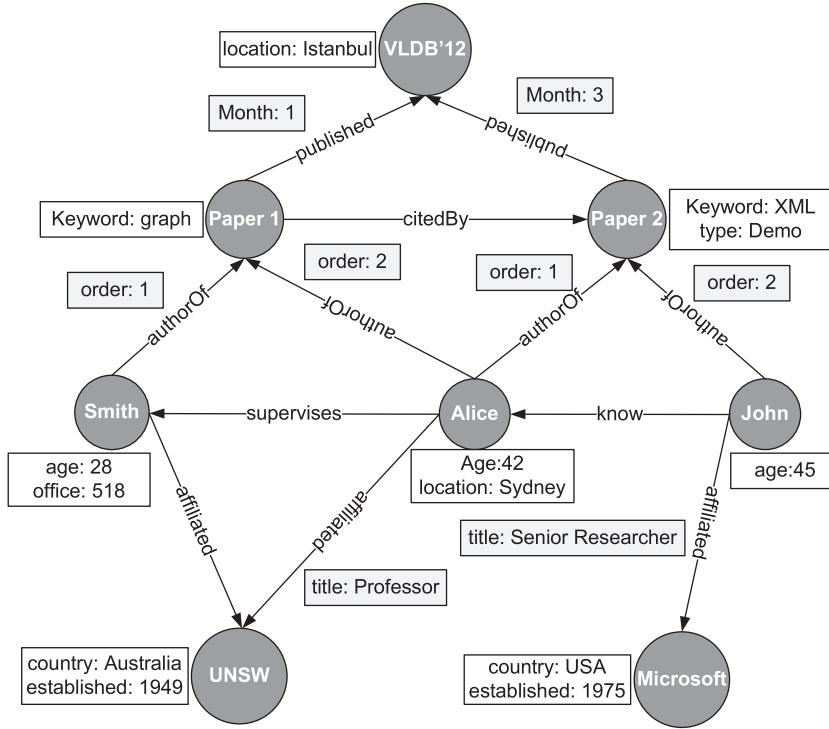


Fig. 1. An example attributed graph.

and knowledge bases [53]. In this category, there are three common types of queries:

- Pattern match query** that tries to find the existence(s) of a pattern graph (e.g. path, star, subgraph) in the large graph [62,63].
- Reachability query** that verifies if there exists a path between any two vertices in the large graph [13,30].
- Shortest path query** which represents a variant version of the reachability query as it returns the shortest path distance (in terms of number of edges) between any two vertices in the large graph (if the two vertices are connected) [12,57].

In this paper, we focus on query processing in the second category of graph databases. In many real applications of this category, both the graph topological structures in addition to the properties of the vertices and edges are important. For example, in a social network, a vertex 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. The problem studied in this paper is to query a graph associated with attributes (called as *attributed graph*) based on both structural and attribute conditions. Unfortunately, this problem did not catch 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. Formally, an *attributed graph* is denoted as $(V, E, L_v, L_e, F_v, F_e, \Lambda_v, \Lambda_e)$ where V is the set of vertices; $E \subseteq V \times V$ is the set of edges joining two distinct vertices; L_v is the set of vertex labels; L_e is the set of edge labels; F_v is a function $V \rightarrow L_v$ that assigns labels to vertices and F_e is a function $E \rightarrow L_e$ that assigns labels to edges; $\Lambda_v = \{a_1, a_2, \dots, a_m\}$ is a set of m attributes that can be associated with any vertex in V . Each vertex $v \in V$ can be described with an attribute vector $[a_1(v), \dots, a_m(v)]$, where $a_j(v)$ is the attribute value of vertex v on attribute a_j . $\Lambda_e = \{b_1, b_2, \dots, b_n\}$ is a set of n attributes that can be associated with any edge in E . Each edge $e \in E$ can be described with an attribute vector $[b_1(e), \dots, b_n(e)]$ where $b_k(e)$ is the attribute value of edge e on attribute b_k .

Fig. 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 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. Examples of these queries are:

- Find the names of two authors, x and y , where x and y are connected by a path (sequence of edges) 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 or equal 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 or equal to 3 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 as a first author. *This query involves pattern matching and reachability expressions where the reachability expression is constrained by a path filtering condition and returns the information of the connecting paths(s) in the query output.*
 - (4) Find the names and the ages of two authors, x and y , and the shortest path between them where the author x is affiliated at UNSW, the author y is affiliated at Microsoft and each of the authors has published a paper at VLDB'12. *This query involves pattern matching and shortest path expressions.*
 - (5) Find the titles of two papers, P_1 and P_2 , and the path between them where each edge in the path represents the cited by relationship and the maximum path length is three edges where the affiliation of the first author of P_1 is affiliated at UNSW and the affiliation of the first author of P_2 is Microsoft. *This query involves a pattern matching expression with condition on the attributes of graph edges in addition to a constrained reachability expression.*

1.1. Limitations of existing approaches

The emerging wave of new graph-based applications in different domains has triggered the calls for new systems for managing large graphs. *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 due to the following deficiencies.

- The existing 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 are usually optimized for accelerating the evaluation of 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 [62], tree [59], subgraph [58]) while proposed techniques for handling reachability queries

use different indexing mechanisms such as the 2-hop cover [13] and the 3-hop cover [30]. 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, it imposes significant overheads for both constructing these indices (in offline pre-processing phase) and maintaining them in the case of supporting updates on the underlying graph database. Moreover, given the increasing sizes of the graph database, the efficiency of such indexing techniques will break down after a certain limit is reached.

- The existing graph querying methods (of both categories of graph databases) that have been presented in the literature mainly focus on querying the topological structure of the graphs [58,59,62] and very few of them have considered the use of attributed graphs [48,54]. While methods for querying the topological structure are more required for applications of transactional graph database, 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. In practice, 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 the optimization process more complex (e.g. evaluation order and join order).
- The existing techniques assume that the whole graph (or at least their constructed indices) can always live entirely in the main memory. This assumption hinders the scalability of the proposed techniques due to their huge memory consumption. Very few approaches have assumed the use of persistent disk to store the graph databases (or portion of it) during the query evaluation process [38,44,55].
- The Resource Description Framework (RDF) represents a special kind of the considered attributed graphs. SPARQL is the official W3C standard query language for RDF graphs [42]. Several graph query processing techniques were proposed for large RDF graphs [7,37,64]. However, these approaches cannot be directly reused for 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 directly described by attributes. Instead, this can be indirectly achieved using a *reification* mechanism which supports the description of an RDF triple using the RDF built-in vocabulary (statement about statement) [35]. In particular, reification can be considered as a mechanism that supports the representation of nested RDF triples where an auxiliary blank node is used as a handle which refers to the whole outer RDF triple and the access to the structure of the inner triple is achieved via the auxiliary blank node. Practically, the reification mechanism is described as “*The*

RDF Big Ugly" [41] as it significantly bloats storage, inflates query times since four new triples are stored for each reification and is less intuitive for the user when he needs to express queries over the attributes of the graph edges. In addition, in the RDF model, graph edges are used to represent both the attribute/literal value pairs of the nodes and the structural relationship with other nodes in the graph with no differentiation. Such uniform treatment for the graph data and graph structure information dramatically increases the size of graph topology. Moreover, the initial specifications of SPARQL [42] did not provide any facility for expressing *path queries* or *reachability* expressions. The recent specifications of the SPARQL 1.1 query language¹ have addressed this limitation in a limited manner. However, very limited number of systems, yet, is currently supporting these features (e.g. Oracle Spatial and Graph²). 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.

- Recently, some native graph database systems have been introduced (e.g. Neo4j³ HypergraphDB⁴). These systems are mainly designed to provide efficient graph traversal functions.⁵ 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 execution for any graph query 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.
- Pregel system [34] has been introduced by Google as a large scale graph processing platform. However, Pregel is designed to work as a batch system for running graph analytical operations and it does not support online query processing.

1.2. Our approach and contributions

We present an approach for *interactive querying* of large attributed graphs which aim to address the above mentioned deficiencies. In particular, we rely 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:

- 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 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 data of the graph are stored and processed using relational database (Section 3).
- We describe an execution mechanism for our proposed query language where the incoming query is first compiled into an intermediate algebraic plan. Then, a split query evaluation mechanism is applied where the execution of parts of the query plan is pushed inside the relational database (using SQL) while the execution of other parts of the query plan is processed using memory-based algorithms, as necessary, for optimizing the query performance (Section 4).
- We conduct extensive experiments with real and synthetic data to evaluate our approach (Section 5).

We review the related work on querying graph databases in Section 6 before we conclude the paper in Section 7.

2. The G-SPARQL query language

To represent queries, a language is needed. Therefore, a number of graph query languages, with the corresponding graph models, have been proposed in the literature [5]. For example, PQL [33] is a special-purpose language which is designed for querying pathways in biological networks. GraphQL [27] was developed as a general language for querying both large sets of small graphs as well as small sets of large graphs. It is designed for querying graph patterns based on graph structure in addition to the node and edge attributes. The language considers graphs as the basic unit of abstraction where the output of each expression is restricted to take the form of a graph structure. However, GraphQL lacks the ability to express queries about arbitrary path structures in the graph. Therefore, some important types of queries for large graphs such as reachability queries and shortest path queries cannot be expressed using GraphQL. Facebook has introduced the Facebook Query Language (FQL)⁶ that provides

¹ <http://www.w3.org/TR/sparql11-query/>

² <http://www.oracle.com/us/products/database/options/spatial/overview/index.html>

³ <http://neo4j.org/>

⁴ <http://www.kobrix.com/hgdb.jsp>

⁵ 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).

⁶ <http://developers.facebook.com/docs/reference/fql/>

the users with a way to query their own data that can be accessed through API functions. The language uses a SQL-style interface and offers a rather primitive and restricted set of queries in order to achieve good performance. For example, the clauses are of the form `select-from-where` with a single from table and join operations are not supported in the language.

The **SPARQL** query language is the official W3C standard for querying and extracting information from RDF graphs [42]. It represents the counterpart to `select-project-join` queries in the relational model. 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:

- 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 represent 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 are 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.
- 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. However, that is not the case in the RDF data model where there is no support for edges to be described by any attribute information (only vertices).

In general, a good query language should have powerful expressiveness so as to satisfy the users' query requirements. In addition, it should be as clear and concise as possible in syntax expressions. Therefore, we introduce **G-SPARQL** [47] 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 compensate 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:

- 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.
- 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).

Fig. 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.

2.1. Querying attributes of nodes and edges

The standard specifications of the SPARQL query language supports defining query graph patterns and expressing various restrictions on the entities and the relationships which are participating in defining these patterns. In particular, each SPARQL query defines a graph pattern P that is matched against an RDF graph G where each

```

<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)

```

Fig. 2. G-SPARQL grammar.

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 *Triple Pattern* [39]. A **Triple Pattern** represents an RDF triple $(\text{subject}, \text{predicate}, \text{object})$ 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 (\cdot) 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 30 years old.

```
SELECT ?X
WHERE {?X affiliatedAt UNSW.      ?X age ?age.
       FILTER (?age > = 30)}
```

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 *object* part of the query triple pattern represents a graph vertex).

(b) **Value-based predicates** specify conditions on the values of the *attributes* in the graph (the *object* part of the query triple pattern represents a literal 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 --> ?Alice affiliatedAt UNSW
T2 --> ?Alice @affiliatedAt "UNSW"
```

where $T1$ represents a structural predicate that specifies the condition of having the graph vertices represented by the variable $?Alice$ connected by an edge that represents the *affiliatedAt* relationship to a vertex with the label *UNSW* while $T2$ represents a value-based predicate that specifies the condition of having the vertices represented by the variable $?Alice$ described by an *affiliatedAt* attribute that stores the literal value of *UNSW*.

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. *Vertex predicates* which enable specifying conditions on the attributes of the graph vertices. *Edge predicates* which enable 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 refer to graph edges and not for graph vertices. To illustrate, let us consider the following example

of query triple patterns:

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

where $T3$ represents a structural predicate that specifies the condition that the vertices represented by the variable $?Alice$ is connected to a vertex with the label *UNSW* with an *affiliatedAt* relationship. $T4$ represents an *edge predicate* that determines that the *Role* attribute of the *affiliatedAt* 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 *Alice* is described by an *officeNumber* attribute that stores the value 518.

2.2. Querying path patterns

One of the main requirements in querying large graphs is the ability to express matching queries based on path patterns. This querying feature is quite important especially when the information about the graph topology or the exact connection patterns between the queried graph vertices is unknown. For example, reachability query is an important type of queries in large graphs that aims to check if there exists a path (of any unknown length) between any two vertices in the large graph. Similarly, the shortest path query aims to determine the shortest connection (number of sequenced edges) between any two vertices in the graph. The initial specifications of the SPARQL query language [42] lack the constructs that support expressing such a type of queries. Therefore, several extensions to the SPARQL query language have been proposed to address some of these requirements such as SPARQ2L [6] and PSPARQL [4]. The most recent specifications of SPARQL 1.1 have tried to address some of these limitations, however, in a limited manner. In particular, SPARQL 1.1 supports the ability to match arbitrary length paths where the end of the path pattern can be represented as an RDF value or a variable. However, variables cannot be used as a part of the path itself, only the ends. For such type of expressions, the query answer includes all matches of that path expression and binds the subject or object variable as specified. To illustrate, let us consider the following examples of query triple patterns:

```
T6 --> ?Alice knows + ?X
T7 --> ?Alice knows + John
```

where $T6$ assigns to the variable x all vertices that can be reached from the vertices that are represented by the variable $?Alice$ through the relationship *knows*. The symbol (+) indicates that the path can be of any length where each edge in the path needs to represent the relationship *knows*. $T7$ represents a structural predicate that describes a reachability test which verifies if the vertices represented by the variable $?Alice$ are connected to a vertex with label *John* by any path (sequence of edges) where each edge in that path represents the relationship *knows*. The predicate filters out the vertices which are bound to the variable $?Alice$ and do not satisfy the condition.

G-SPARQL supports this extension for expressing path patterns and, similar to *SPARQL2L* [6] and *PSPARQL* [4], allows path variables in the predicate position of a triple pattern (subject, path variable, object). In particular, *G-SPARQL* supports the following options of binding path variables in the path patterns.

```
T8 --> subject ??P object
T9 --> subject ?*P object
T10 --> subject ??P(predicate) object
T11 --> subject ?*P(predicate) object
```

where *T8* 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 *T9*, the symbol $(?*)$ indicates that the variable *P* will be matched with the *shortest* path between the two vertices of *subject* and *object*. *T10* ensures that each edge in the matching paths represents the specified relationship predicate. Similarly, *T11* ensures that each edge in the matched shortest path represents the relationship predicate.

In general, any two graph 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.

- **Length(*PV*, *P*):** This filtering condition 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.
- **AtLeastNode(*PV*, *N*, *P*):** Verifies if at least *N* number of nodes 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*. This predicate can be a structural predicate or a value-based predicate. Let us consider the following examples of these predicates.
 - *AtLeastNode(??X, 2, livesIn Sydney)* ensures that at least 2 nodes of each path, which is bound to the variable *x*, are satisfying the structural predicate of being connected through the *livesIn* relationship to a vertex with the label *Sydney*.
 - *AtLeastNode(??X, 1, @affiliated UNSW)* ensures that at least 1 node of each path, which is bound to the variable *x*, satisfies the value-based predicate of having a value of the *affiliated* attribute equals to *UNSW*.
- **AtMostNode(*PV*, *N*, *P*):** Ensures that at most *N* number of nodes on each path which is bound to the variable *PV* satisfies the structure/value-based predicate *P*.

- **AllNodes(*PV*, *P*):** Ensures that every node of each path which is bound to the variable *PV* satisfies the structure/value-based predicate *P*.
- **AtLeastEdge(*PV*, *N*, *P*):** Ensures that at least *N* number of edges on each path which is bound to the variable *PV* satisfies the value-based predicate *P* (structural predicates cannot be represented for graph edges).
- **AtMostEdge(*PV*, *N*, *P*):** Ensures that at most *N* number of edges on each path which is bound to the variable *PV* satisfies the value-based predicate *P*.
- **AllEdges(*PV*, *P*):** Ensures that at every edge of each path which is bound to the variable *PV* satisfies the value-based predicate *P*.

3. Hybrid graph representation

Due to the lack of scalable graph indexing mechanisms and cost-effective graph query optimizers, it becomes very challenging to search and analyze any reasonably large networks. Therefore, there is a crucial need and strong motivation to take advantage of well-studied relational database indexing and query optimization techniques to address the problems of querying large graphs. In general, relational databases have been considered as the main choice for most traditional data-intensive storage and retrieval applications for decades. In practice, relational database systems are generally very efficient for queries that require extensive use of physical indexing (e.g. B-tree) and sophisticated query optimization techniques (e.g. selectivity estimation and join ordering). For example, by applying predicates, relational indices can limit the data that must be accessed to only those rows that satisfy those predicates. In addition, query evaluations can be achieved using index-only access and save the necessity to access the data pages by providing all the columns needed for the query evaluation. However, relational databases turn to be inefficient for answering the queries that would require looping or recursive access to large numbers of records by executing multiple expensive join operations which may yield to very large intermediate results. Hence, executing traversal operations over relationally stored graphs can be time-inefficient due to the large number of potential joins in addition to the expensive disk access cost to retrieve the target vertices. As such, it is much more efficient to rely on algorithms that execute on main memory data structures to answer graph queries that require heavy traversal operations on the graph topology. Therefore, in our graph representation, we rely on a hybrid mechanism where the entire graph information (topology+data) is stored in relational database while only the topology information of the graph needs to be loaded onto the main memory for accelerating the query evaluation process when necessary.

In our approach, we physically store the attributed graph in a relational database. In general, it is a virtue of the relational database model that its canonical physical representation, tables of tuples, is simple and thus efficient to implement. Therefore, several approaches have been proposed for storing XML [20] and RDF [45] databases using relational databases. However, none of these approaches can

be directly reused for storing attributed graphs due to the differences in the data model and the specifications of the query requirements. Therefore, we adopted an approach for storing attributed graphs in relational database using a fully decomposed storage model (DSM) [1,14]. In particular, we start by assigning identifiers (*IDs*) for each vertex and edge in the graph. Then, the attributed graph is mapped into $M+N$ two-column tables and P three-column tables where M is the number of *unique* attributes of the graph vertices, N is the number of *unique* attributes of the graph edges and P is the number of *unique* relationships that exists between the graph vertices. The first column (*ID*) for each of the $(M+N)$ two-column attribute tables stores the identifiers of those vertices/edges that are described by the associated attribute while the second column (*Value*) stores the literal values for those attributes. Obviously, the vertices/edges which are not described by a particular attribute will simply not have a representative record in the associated table for that attribute. Multi-valued attributes will be represented with multiple rows (with the same *ID* value) in the table for that attribute.

Each table is sorted with a clustered index on the *ID* column in order to enable a fast execution for merge joins where multiple attributes of the same vertex/edge need to be retrieved. In addition, a *partitioned B-tree index* (*Value, ID*) is created for each table in order to allow efficient execution for value-based predicates on the attributes of the graphs by reducing the access costs of the secondary storage to retrieve those nodes/edges that satisfy the condition of a predicate to a minimum [21]. The P three-column tables capture the graph *topology* information. In particular, each of these tables groups the information of all graph edges that represent a particular relationship where each edge is described by three pieces of information: the edge identifier (*eID*), the identifier of the source vertex of the edge (*sID*) in addition to the identifier of the destination vertex (*dID*). Fig. 3 illustrates an example of our relational representation for the attributed graph of Fig. 1.

A main advantage of the DSM-based mapping for the attributed graph is that it is agnostic to the graph schema. Therefore, it can be straightforwardly applied to any attributed graph with any schema. In addition, during the query processing, the disk access costs can be significantly reduced because only the records of the tables of those attributes/relationships which are involved in a query will be processed. On the other hand, some queries may need to query/access several attributes for the same entity (vertex or edge). In this case, a number of attribute tables which is equal to the number of attributes that are involved in the query have to be joined in order to re-group the fragmented information of the graph entities over multiple tables. However, this challenge can be tackled by relying on inexpensive merge joins which make use of the sorting of the tables in their *ID* columns and their clustered indices. In practice, it is not very common that the posed queries would involve a *very large* number of attributes/relationships.

In our hybrid graph representation, we rely on a native pointer-based data structure for representing the graph topology information in the main memory. In particular, this memory-based 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 represents the mandatory knowledge for executing the index-free and memory-based algorithms that involve heavy traversal operations on the graph topology or that of recursive nature such as performing Dijkstra's algorithm to obtain the shortest path between any two vertices or performing a breath-first search (BFS) to answer reachability queries [15]. Therefore, our hybrid representation achieves a clear reduction in the main memory consumption as it avoids loading the attributes of the graph vertices/edges and their data values ($M+N$ attribute tables) into the main memory. In addition, it avoids building extra main memory indices for the graph attributes which might be required for accelerating the query evaluation process of the query predicates and

Node Label		age		office		location		keyword		type		established			
ID	Value	ID	Value	ID	Value	ID	Value	ID	Value	ID	Value	ID	Value		
1	John	1	45	8	518	3	Sydney	2	XML	2	Demo	4	1975		
2	Paper 2	3	42			5	Istanbul	6	graph			7	1949		
3	Alice	8	28												
authorOf															
country			affiliated			published			citedBy			order			
ID			eID sID dID			eID sID dID			eID sID dID			eID sID dID			
4	USA		1	1	2	3	1	4	4	2	5	9	6	2	
5	Australia		5	3	2	8	3	7	10	6	5				
6			6	3	6	12	8	7							
7			11	8	6										
know															
eID sID dID			supervise			ID Value			ID Value			month			
2	1	3	7	3	8	3	Senior Researcher		4	3		1	2		
						8	Professor		10	1					

Fig. 3. Relational representation of attributed graph of Fig. 1.

pushes these tasks to the underlying relational storage. Such reduction in the main memory consumption provides better scalability opportunities for handling larger graphs for a given fixed size of the available main memory.

4. Query execution engine

A crucial strength of any graph query language is that it can be efficiently implemented. 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. For example, relational algebra has been a crucial foundation for relational database systems and has played a large role in enabling their success. In XML query processing, some approaches have relied on a form of tree algebra where trees are the basic unit and the operators work on collections of trees [29] while some other approaches have relied on a relational algebraic mechanism for compiling XML queries [24]. GraphQL [27] compiles its graph queries using a form of graph algebra where graphs are the basic unit of information and each operator takes one or more collections of graphs as input and generates a collection of graphs as output. Fig. 4 illustrates the execution steps of a G-SPARQL query where the incoming query is first compiled into an intermediate algebraic plan then a process of query optimization is applied to generate the query execution plan where parts of the query plans are pushed to be executed inside the relational database layer while the rest of the query plan is processed using memory-based algorithms. More details about the execution steps

of a G-SPARQL query are presented in the following subsections.

4.1. G-SPARQL algebraic operators

Several approaches have been proposed for compiling SPARQL queries into relational algebraic plans [10,17,18], as an abstract intermediate language for representing the queries. We follow the same approach of [17,24] where we rely on a dialect of *tuple algebra* for compiling G-SPARQL queries. 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. In particular, our algebra considers *tuples* as the basic unit of information. Each algebraic operator manipulates collections of tuples. However, our logical algebra extends the set of traditional relational algebraic operators (e.g. selection, projection, cartesian product, join) with a set of logical operators that are capable of expressing complex G-SPARQL operations that cannot be matched with the semantics of the traditional relational operators. The descriptions of these algebraic operators are listed in Table 1. In presenting our operators, we use the notation of $\text{opName}_{IC,(\text{Const})[OIC]:OC,[OOC]}$ where opName is the name of the operator, IC is a mandatory parameter of the operator that represents the name of an input column, Const represents the value of a constant parameter of the operator, OIC is an optional parameter that represents the name of an input column, OC is the name of a main output column of the operator and OOC is the name of an optional output column.

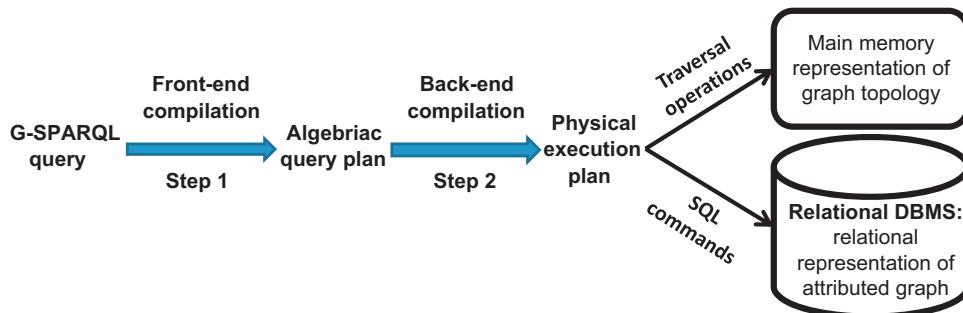


Fig. 4. Execution steps of a G-SPARQL query.

Table 1
G-SPARQL algebraic operators.

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

As we previously described, query triple patterns represent the main constructs of building *G-SPARQL* queries where each variable in the query triple pattern is replaced by matching elements from the queried graph. Evaluating a *G-SPARQL* query yields a set of variable bindings (a mapping from variable names to values). Shared variable names are represented as join equality predicates in the query plan. The semantics of each of our defined operators is identical to exactly one possible query triple pattern in *G-SPARQL*. Descriptions of the *G-SPARQL* algebraic operators ([Table 1](#)) are given as follows:

- 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. Based on the attributed graph of [Fig. 1](#) and its relational representation in [Fig. 3](#), [Fig. 5\(a\)](#) illustrates an example for the behavior of the `NgetAttVal` operator where it retrieves the values of the `location` attribute for an input relation with the (*id*) of three graph vertices ("John", "Alice", "Smith"). The output relation includes only one record (for the vertex "Alice") that has the `location` attribute with the value "Sydney". The other two vertices are filtered out because they do not have values for the indicated `location` attribute. Similarly, the `EgetAttVal` operator retrieves the values of a specific attribute for a set of graph edges. [Fig. 5\(b\)](#) 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 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 predicated (*p*) over a specific column holds. Hence, it represents the right match for reflecting the expressivity of the SPARQL FILTER expressions. [Fig. 5\(c\)](#) illustrates an extension of the example of [Fig. 5\(b\)](#) where the input edges are filtered based on a predicate (`title = "Professor"`) for the retrieved values of the `title` attribute.
- The `getEdgeNodes` is a unary operator which is used for retrieving a set of adjacent nodes that are connected through a specified relation. The operator receives a set of tuples where the column (*id*) of the input relation identifies the graph nodes and the 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 edge (*eID*) of the specified relation and the adjacent node (*dID*). The operator filters out the tuples where the nodes identified by the column (*id*) do not have adjacent nodes connected with the specified relation. [Fig. 5\(d\)](#) illustrates an example of the `getEdgeNodes` operator where it retrieves the adjacent nodes through the relationship `know` of an input relation with the (*id*) of two graph vertices ("John", "Alice"). The output relation filters out the "Alice" vertex as it is not connected to any other node through the `know` relationship. [Fig. 5\(e\)](#) illustrates an example of the `getEdgeNodes` operator where it retrieves all the adjacent nodes for an input relation with the (*id*) of two graph vertices ("John", "Alice").
- 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*). [Fig. 5\(f\)](#) 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. [Fig. 5\(g\)](#) illustrates another example of the `strucPred` operator which projects the information of the connecting edges that represent the structural predicate. In this example, the schema of the output relation is extended with an additional column that stores the *ID* of the connecting edges (*eID*).
- 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*). [Fig. 5\(h\)](#) illustrates an 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 any relationship. 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*). [Fig. 5\(j\)](#) illustrates another example of the `edgeJoin` operator where it receives two sets of graph vertices - the same as [Fig. 5\(h\)](#) - and returns pairs of graph vertices that are connected through an affiliated relationship. Moreover, the `edgeJoin` operator can optionally return 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 in the output tuples according to the specified input parameters. [Fig. 5\(i\)](#) and [\(k\)](#) illustrates another example of the behavior of the `edgeJoin` operator which are similar to the examples shown in [Fig. 5\(h\)](#) and [\(j\)](#), respectively, with only one difference that the output

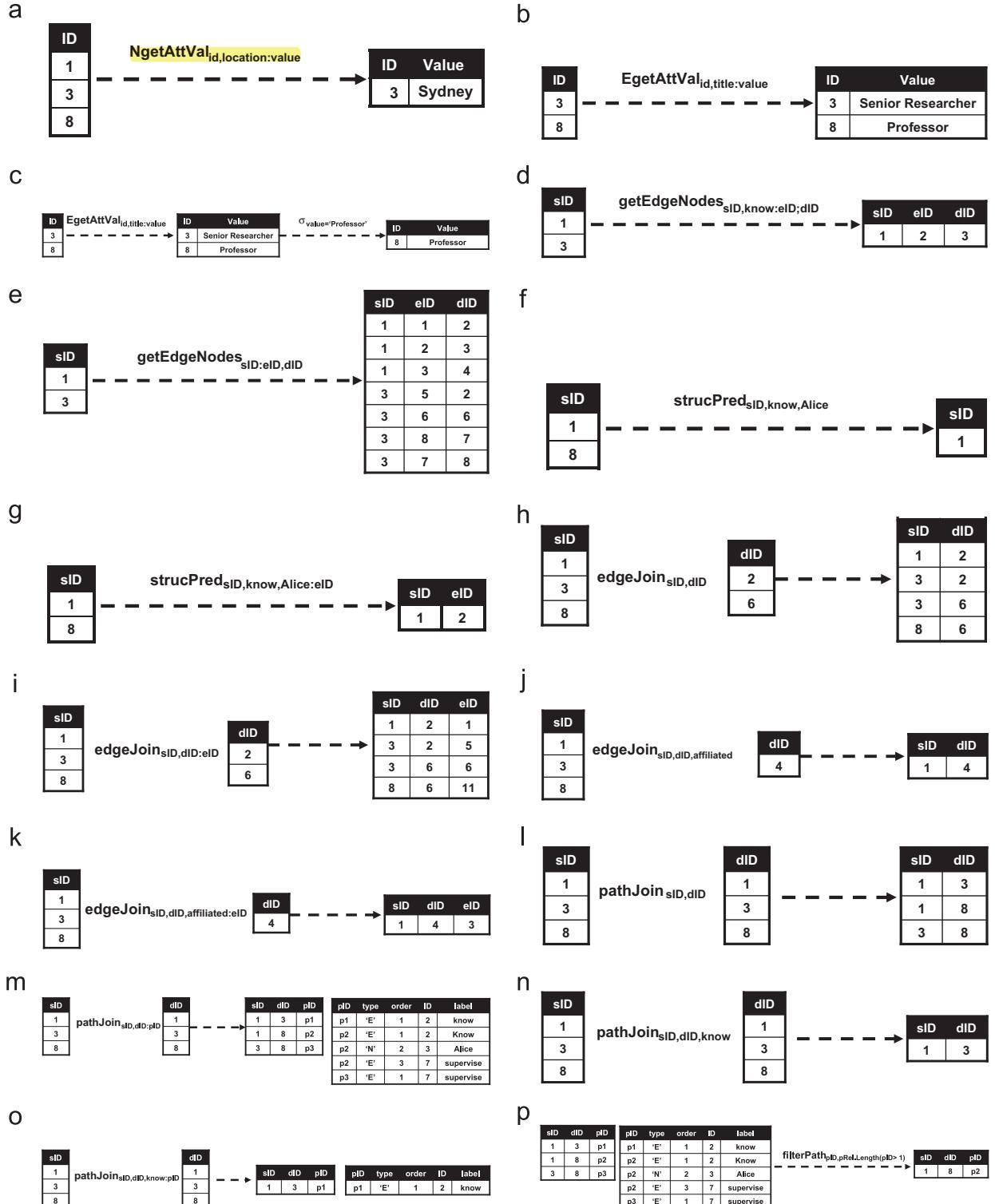


Fig. 5. Example behavior of G-SPARQL algebraic operators: (a) the `NgetAttVal` operator; (b) the `EgetAttVal` operator; (c) the `Filter` operator; (d) the `getEdgeNodes` operator (specified relationship); (e) the `getEdgeNodes` operator (all relationships); (f) the `strucPred` operator; (g) the `strucPred` operator (with projection); (h) the `edgeJoin` operator; (i) the `edgeJoin` operator (with projection); (j) the `edgeJoin` operator (specified relationship); (k) the `edgeJoin` operator (specified relationship with projection); (l) the `pathJoin` operator; (m) the `pathJoin` operator (with projection); (n) the `pathJoin` operator (specified relationship); (o) the `pathJoin` operator (specified relationship with projection); and (p) the `filterPath` operator.

relations include the (*ID*) of the connecting edges between the output pair of vertices.

- 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. Fig. 5(l) illustrates an example of the `pathJoin` operator where it receives two sets of graph vertices – ("John", "Alice", "Smith") and ("John", "Alice", "Smith") – and returns pairs of graph vertices that are connected through a sequence of relationships of any length. 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 label (*eLabel*). Fig. 5(n) illustrates an example of the `pathJoin` operator where it receives two sets of graph vertices – the same as Fig. 5(n) – and returns pairs of graph vertices that are connected through a sequence of any length of known relationships. Moreover, the `pathJoin` operator can optionally return the information of the connecting path(s) as follows:

- 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 describe the information of the bound paths.
- 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. In particular, the tuples are described by the following fields: (*pID*) represents the path identifier, (*type*) represents if the tuple describes a node (then it has a value 'N') or an edge (value 'E'), (*order*) represents the order of the node/edge that participates in the path (the order of each path always starts with an edge ordered as 1 followed by a node ordered as 2 and then it alternates till the last constituting edge of the path), (*ID*) represents the identifier of the node/edge in addition to its (*Label*) information. 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*).

Fig. 5(m) and (o) illustrates another example of the behavior of the `pathJoin` operator which is similar to the example of Fig. 5(l) and (n), respectively, with the difference that they return the information of the resulting connecting paths. The `sPathJoin` operator works in the same way of the `pathJoin` operator with only one difference is 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 we previously described in Section 2.2. Fig. 5(p) illustrates an example of the `filterPath` operator which filters a set of paths based on a *length* condition (number of edges) which returns the paths with lengths greater than 1 and filters out the rest.

4.2. Query compilation

As indicated in Table 1, not all of our algebraic operators can be represented by the standard relational algebraic operators. Based on our relational representation of the attributed graphs, Fig. 6 depicts the relational representations for those operators that can be compiled into a pattern of standard relational algebraic operators. For example, the `NgetAttVal` operator is mapped to a join operation between the column (*nodeID*) of the input relation (*R*) and the (*ID*) column of the relation that represents the specified node attribute (*attName*). The operator extends the schema of the input tuples (*R.**) with the (*value*) column for the relation of the specified attribute (*attName*). Since the semantic of the operators `getEdgeNodes` and `edgeJoin` can be not restricted by a specified relationship (*eLabel*), compiling these operators using the standard relational operators requires

<code>NgetAttVal_{nodeID,(attName):value}</code> $\Rightarrow \pi_{R.*} \text{attName.value}(R \bowtie_{R.\text{nodeID}=\text{attName.ID}} \text{attName})$
<code>EgetAttVal_{edgeID,(attName):value}</code> $\Rightarrow \pi_{R.*} \text{attName.value}(R \bowtie_{R.\text{edgeID}=\text{attName.ID}} \text{attName})$
<code>getEdgeNodes_{sID,(eLabel):eID,dID}</code> $\Rightarrow \pi_{R.*} \text{eLabel.eID,eLabel.dID}(R \bowtie_{R.\text{sID}=eLabel.sID} eLabel)$
<code>getEdgeNodes_{sID:eID,dID}</code> $\Rightarrow \pi_{R.*} \text{allEdges.eID,dID}(R \bowtie_{R.\text{sID}=allEdges.sID} allEdges)$
<code>strucPred_{sID,(eL),(dNL)}</code> $\Rightarrow \pi_{R.*} (\sigma_{nodeLabel.Value=dNL} ((R \bowtie_{R.\text{sID}=eL.sID} eL) \bowtie_{eL.dID=nodeLabel.ID} nodeLabel))$
<code>strucPred_{sID,(eL),(dNL):eID}</code> $\Rightarrow \pi_{R.*} eL.eID (\sigma_{nodeLabel.Value=dNL} ((R \bowtie_{R.\text{sID}=eL.sID} eL) \bowtie_{eL.dID=nodeLabel.ID} nodeLabel))$
<code>edgeJoin_{R.sID,S,dID}</code> $\Rightarrow \pi_{R.*} S.* ((R \bowtie_{R.\text{sID}=allEdges.sID} allEdges) \bowtie_{allEdges.dID=S.dID} S)$
<code>edgeJoin_{R.sID,S,dID:eID}</code> $\Rightarrow \pi_{R.*} S.* ((R \bowtie_{R.\text{sID}=allEdges.sID} allEdges) \bowtie_{allEdges.dID=S.dID} S)$
<code>edgeJoin_{R.sID,S,dID,(eLabel)}</code> $\Rightarrow \pi_{R.*} S.* ((R \bowtie_{R.\text{sID}=eLabel.sID} eLabel) \bowtie_{eLabel.dID=S.dID} S)$
<code>edgeJoin_{R.sID,S,dID,(eLabel):eID}</code> $\Rightarrow \pi_{R.*} S.* eLabel.eID ((R \bowtie_{R.\text{sID}=eLabel.sID} eLabel) \bowtie_{eLabel.dID=S.dID} S)$

Fig. 6. Relational representation of G-SPARQL algebraic operators.

$$\begin{array}{c}
 \frac{\text{SQL}(t) : R}{\left(\begin{array}{l} \text{SQL}(\text{NgetAttVal}_{nodeID, (attName):value}) \Rightarrow \\ \text{SELECT R.*}, attName.Value \text{ FROM R, attName WHERE R.nodeID = attName.ID;} \end{array} \right) \quad (1)} \\
 \frac{\text{SQL}(t) : R}{\left(\begin{array}{l} \text{SQL}(\text{EgetAttVal}_{edgeID, (attName):value}) \Rightarrow \\ \text{SELECT R.*}, attName.Value \text{ FROM R, attName WHERE R.edgeID = attName.ID;} \end{array} \right) \quad (2)} \\
 \frac{\text{SQL}(t) : R}{\left(\begin{array}{l} \text{SQL}(\text{getEdgeNodes}_{sID, (eLabel):eID, dID}) \Rightarrow \\ \text{SELECT R.*}, eLabel.eID, eLabel.dID \text{ FROM R, eLabel} \\ \text{WHERE R.edgeID = eLabel.sID;} \end{array} \right) \quad (3)} \\
 \frac{\text{SQL}(t) : R}{\left(\begin{array}{l} \text{SQL}(\text{getEdgeNodes}_{sID:eID, dID}) \Rightarrow \\ \text{SELECT R.*}, allEdges.eID, allEdges.dID \text{ FROM R, allEdges} \\ \text{WHERE R.edgeID = allEdges.sID;} \end{array} \right) \quad (4)} \\
 \frac{\text{SQL}(t) : R}{\left(\begin{array}{l} \text{SQL}(\text{strucPred}_{sID, (eLabel), (dNLabel)}) \Rightarrow \\ \text{SELECT R.*} \text{ FROM R, eLabel, NodeLabel} \\ \text{WHERE R.sID = eLabel.sID AND eLabel.dID = NodeLabel.ID} \\ \text{AND NodeLabel.Value = 'dNLabel'}; \end{array} \right) \quad (5)} \\
 \frac{\text{SQL}(t) : R}{\left(\begin{array}{l} \text{SQL}(\text{strucPred}_{sID, (eLabel), (dNLabel):eID}) \Rightarrow \\ \text{SELECT R.*}, eLabel.eID \text{ FROM R, eLabel, NodeLabel} \\ \text{WHERE R.sID = eLabel.sID AND eLabel.dID = NodeLabel.ID} \\ \text{AND NodeLabel.Value = 'dNLabel'}; \end{array} \right) \quad (6)} \\
 \frac{\text{SQL}(t_1) : R \quad \text{SQL}(t_2) : S}{\left(\begin{array}{l} \text{SQL}(\text{edgeJoin}_{R.sID, S, dID}) \Rightarrow \\ \text{SELECT R.*}, S.* \text{ FROM R, S, allEdges} \\ \text{WHERE R.sID = allEdges.sID AND allEdges.dID = S.dID;} \end{array} \right) \quad (7)} \\
 \frac{\text{SQL}(t_1) : R \quad \text{SQL}(t_2) : S}{\left(\begin{array}{l} \text{SQL}(\text{edgeJoin}_{R.sID, S, dID:eID}) \Rightarrow \\ \text{SELECT R.*}, S.* \text{ FROM R, S, allEdges} \\ \text{WHERE R.sID = allEdges.sID AND allEdges.dID = S.dID;} \end{array} \right) \quad (8)} \\
 \frac{\text{SQL}(t_1) : R \quad \text{SQL}(t_2) : S}{\left(\begin{array}{l} \text{SQL}(\text{edgeRJoin}_{R.sID, S, dID, (eLabel)}) \Rightarrow \\ \text{SELECT R.*}, S.* \text{ FROM R, S, eLabel} \\ \text{WHERE R.sID = eLabel.sID AND eLabel.dID = S.dID;} \end{array} \right) \quad (9)} \\
 \frac{\text{SQL}(t_1) : R \quad \text{SQL}(t_2) : S}{\left(\begin{array}{l} \text{SQL}(\text{edgeRJoin}_{R.sID, S, dID, (eLabel):eID}) \Rightarrow \\ \text{SELECT R.*}, S.* \text{ FROM R, S, eLabel} \\ \text{WHERE R.sID = eLabel.sID AND eLabel.dID = S.dID;} \end{array} \right) \quad (10)}
 \end{array}$$

Fig. 7. SQL translations of algebraic operators.

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. Fig. 7 depicts the inference rules for the SQL translation templates of the algebraic operators. A sample interpretation of the inference rule TRANS-1 that

represents the translation of the *NgetAttVal* operator is: Given the information that the relation (*R*) represents the SQL evaluation of the triple pattern *t*, the translation of the (*NgetAttVal*_{*nodeID, (attName):value*}) operator is defined using the following SQL code:

```
SELECT R.*, attName.Value FROM R, attName
WHERE R.nodeID=attName.ID;
```

$$\begin{array}{c}
 \frac{\text{Map}(\text{?var}) \in R \quad \text{Col}(\text{?var}) \equiv \text{ID}}{(\text{?var}, \text{@attName}, \text{?var2}) \Rightarrow \begin{array}{l} \text{NgetAttVal}_{ID,(\text{attName}):value}(R) \\ \text{Col}(\text{?var2}) \equiv \text{value} \end{array}} \quad (1) \\
 \frac{\text{Map}(\text{?var}) \in R \quad \text{Col}(\text{?var}) \equiv \text{ID}}{(\text{?var}, \text{@attName}, \text{'attValue'}) \Rightarrow \pi_{ID}(\sigma_{value='attValue'}(\text{NgetAttVal}_{ID,(\text{attName}):value}(R)))} \quad (2) \\
 \frac{\begin{array}{l} (\text{subject}, \text{?E}(\text{predicate}), \text{object}) \in \text{Triple}(q) \\ \text{Map}(\text{?E}) \in R \quad \text{Col}(\text{?E}) \equiv \text{ID} \end{array}}{(\text{?E}, \text{@attName}, \text{?var}) \Rightarrow \begin{array}{l} \text{EgetAttVal}_{ID,(\text{attName}):value}(R) \\ \text{Col}(\text{?var}) \equiv \text{value} \end{array}} \quad (3) \\
 \frac{\begin{array}{l} (\text{subject}, \text{?E}(\text{predicate}), \text{object}) \in \text{Triple}(q) \\ \text{Map}(\text{?E}) \in R \quad \text{Col}(\text{?E}) \equiv \text{ID} \end{array}}{(\text{?E}, \text{@attName}, \text{'attValue'}) \Rightarrow \pi_{ID}(\sigma_{value='attValue'}(\text{EgetAttVal}_{ID,(\text{attName}):value}(R)))} \quad (4) \\
 \frac{\text{Map}(\text{?var}) \in R \quad \text{Col}(\text{?var}) \equiv \text{sID}}{(\text{?var}, \text{eLabel}, \text{dNLabel}) \Rightarrow \text{strucPred}_{sID,(\text{eLabel}),(\text{dNLabel})}(R)} \quad (5) \\
 \frac{\begin{array}{l} (\text{?E}, \text{predicate}, \text{object}) \in \text{Triple}(q) \\ \text{Map}(\text{?var}) \in R \quad \text{Col}(\text{?var}) \equiv \text{sID} \end{array}}{(\text{?var}, \text{?E}(\text{eLabel}), \text{dNLabel}) \Rightarrow \begin{array}{l} \text{strucPred}_{sID,(\text{eLabel}),(\text{dNLabel}):eID}(R) \\ \text{Col}(\text{?E}) \equiv \text{eID} \end{array}} \quad (6) \\
 \frac{\text{Map}(\text{?var}) \in R \quad \text{Col}(\text{?var}) \equiv \text{sID}}{(\text{?var}, \text{eLabel}, \text{?var2}) \Rightarrow \begin{array}{l} \text{getEdgeNodes}_{sID,(\text{eLabel}):eID,dID}(R) \\ \text{Col}(\text{eLabel.eID}) \equiv \text{eID} \quad \text{Col}(\text{?var2}) \equiv \text{dID} \end{array}} \quad (7) \\
 \frac{\text{Map}(\text{?var}) \in R \quad \text{Col}(\text{?var}) \equiv \text{sID}}{(\text{?var}, \text{?E}, \text{?var2}) \Rightarrow \begin{array}{l} \text{getEdgeNodes}_{sID:eID,dID}(R) \\ \text{Col}(\text{allEdges.eID}) \equiv \text{eID} \quad \text{Col}(\text{?var2}) \equiv \text{dID} \end{array}} \quad (8) \\
 \frac{\begin{array}{l} (\text{?E}, \text{predicate}, \text{object}) \in \text{Triple}(q) \\ \text{Map}(\text{?var}) \in R \quad \text{Col}(\text{?var}) \equiv \text{sID} \\ \text{Map}(\text{?var2}) \in S \quad \text{Col}(\text{?var2}) \equiv \text{dID} \end{array}}{(\text{?var}, \text{?E}(\text{eLabel}), \text{?var2}) \Rightarrow \begin{array}{l} (\text{R}) \text{edgeJoin}_{sID,dID,(\text{eLabel}):eID}(\text{S}) \\ \text{Col}(\text{?E}) \equiv \text{eID} \end{array}} \quad (9) \\
 \frac{\begin{array}{l} \text{Map}(\text{?var}) \in R \quad \text{Col}(\text{?var}) \equiv \text{sID} \\ \text{Map}(\text{?var2}) \in S \quad \text{Col}(\text{?var2}) \equiv \text{dID} \end{array}}{(\text{?var}, \text{eLabel}, \text{?var2}) \Rightarrow \begin{array}{l} (\text{R}) \text{edgeJoin}_{sID,dID,(\text{eLabel})}(\text{S}) \end{array}} \quad (10) \\
 \frac{\begin{array}{l} (\text{?E}, \text{predicate}, \text{object}) \in \text{Triple}(q) \\ \text{Map}(\text{?var}) \in R \quad \text{Col}(\text{?var}) \equiv \text{sID} \\ \text{Map}(\text{?var2}) \in S \quad \text{Col}(\text{?var2}) \equiv \text{dID} \end{array}}{(\text{?var}, \text{?E}, \text{?var2}) \Rightarrow \begin{array}{l} (\text{R}) \text{edgeJoin}_{sID,dID:eID}(\text{S}) \\ \text{Col}(\text{?E}) \equiv \text{eID} \end{array}} \quad (11) \\
 \frac{\begin{array}{l} \text{Map}(\text{?var}) \in R \quad \text{Col}(\text{?var}) \equiv \text{sID} \\ \text{Map}(\text{?var2}) \in S \quad \text{Col}(\text{?var2}) \equiv \text{dID} \end{array}}{(\text{?var}, \text{?E}, \text{?var2}) \Rightarrow \begin{array}{l} (\text{R}) \text{edgeJoin}_{sID,dID}(\text{S}) \end{array}} \quad (12) \\
 \frac{\begin{array}{l} \text{Map}(\text{?var}) \in R \quad \text{Col}(\text{?var}) \equiv \text{sID} \\ \text{Map}(\text{?var2}) \in S \quad \text{Col}(\text{?var2}) \equiv \text{dID} \\ \text{FilterPath}(\text{??P}) \in \text{PathFilter}(q) \end{array}}{(\text{?var}, \text{??P}, \text{?var2}) \Rightarrow \begin{array}{l} (\text{R}) \text{pathJoin}_{sID,dID:pID,pRel}(\text{S}) \\ \text{Col}(\text{??P}) \equiv \text{pID} \\ \text{Schema}(\text{pRel}) = (\text{pID}, \text{type}, \text{order}, \text{ID}, \text{Label}) \end{array}} \quad (13)
 \end{array}$$

Fig. 8. G-SPARQL operator mapping rules (1).

4.2.1. Front-end compilation

In general, the design of our compilation procedure and our logical operators is independent of any specific disk or memory representation of the attributed graph.

In addition, they are independent of the underlying query evaluation engine. Therefore, in a first compilation step, we decide for each query triple pattern a mapping onto algebraic operators. Figs. 8 and 9 illustrate

$$\begin{array}{c}
 \begin{array}{l}
 \text{Map}(\text{?var}) \in R \quad \text{Col}(\text{?var}) \equiv \text{sID} \\
 \text{Map}(\text{?var2}) \in S \quad \text{Col}(\text{?var2}) \equiv \text{dID} \\
 \text{FilterPath}(\text{??P}) \in \text{PathFilter}(q)
 \end{array}
 \xrightarrow{\quad}
 \begin{array}{l}
 (\text{R}) \text{ pathJoin}_{sID,dID,(eLabel):pID,pRel} (S) \\
 \text{Col}(\text{??P}) \equiv \text{pID} \\
 \text{Schema(pRel)} = (\text{pID}, \text{type}, \text{order}, \text{ID}, \text{Label})
 \end{array}
 \quad (14)
 \end{array}$$

$$\begin{array}{c}
 \begin{array}{l}
 \text{Map}(\text{?var}) \in R \quad \text{Col}(\text{?var}) \equiv \text{sID} \\
 \text{Map}(\text{?var2}) \in S \quad \text{Col}(\text{?var2}) \equiv \text{dID}
 \end{array}
 \xrightarrow{\quad}
 \begin{array}{l}
 (\text{R}) \text{ pathJoin}_{sID,dID} (S)
 \end{array}
 \quad (15)
 \end{array}$$

$$\begin{array}{c}
 \begin{array}{l}
 \text{Map}(\text{?var}) \in R \quad \text{Col}(\text{?var}) \equiv \text{sID} \\
 \text{Map}(\text{?var2}) \in S \quad \text{Col}(\text{?var2}) \equiv \text{dID}
 \end{array}
 \xrightarrow{\quad}
 \begin{array}{l}
 (\text{R}) \text{ pathJoin}_{sID,dID,(eLabel)} (S)
 \end{array}
 \quad (16)
 \end{array}$$

$$\begin{array}{c}
 \begin{array}{l}
 \text{Map}(\text{?var}) \in R \quad \text{Col}(\text{?var}) \equiv \text{sID} \\
 \text{Map}(\text{?var2}) \in S \quad \text{Col}(\text{?var2}) \equiv \text{dID} \\
 \text{FilterPath}(\text{?*P}) \in \text{PathFilter}(q)
 \end{array}
 \xrightarrow{\quad}
 \begin{array}{l}
 (\text{R}) \text{ sPathJoin}_{sID,dID:pID,pRel} (S) \\
 \text{Col}(\text{?*P}) \equiv \text{pID} \\
 \text{Schema(pRel)} = (\text{pID}, \text{type}, \text{order}, \text{ID}, \text{Label})
 \end{array}
 \quad (17)
 \end{array}$$

$$\begin{array}{c}
 \begin{array}{l}
 \text{Map}(\text{?var}) \in R \quad \text{Col}(\text{?var}) \equiv \text{sID} \\
 \text{Map}(\text{?var2}) \in S \quad \text{Col}(\text{?var2}) \equiv \text{dID} \\
 \text{FilterPath}(\text{?*P}) \in \text{PathFilter}(q)
 \end{array}
 \xrightarrow{\quad}
 \begin{array}{l}
 (\text{R}) \text{ sPathJoin}_{sID,dID,(eLabel):pID,pRel} (S) \\
 \text{Col}(\text{?*P}) \equiv \text{pID} \\
 \text{Schema(pRel)} = (\text{pID}, \text{type}, \text{order}, \text{ID}, \text{Label})
 \end{array}
 \quad (18)
 \end{array}$$

Fig. 9. G-SPARQL operator mapping rules (2).

the inference rules for mapping the G-SPARQL query triple patterns into our algebraic operators. A sample interpretation of the inference rule OPMAP-1 is that it maps a query triple pattern $(\text{?var}, \text{@attName}, \text{?var2})$ of query q into the algebraic operator $(\text{NgetAttVal}_{ID,(\text{attName}):value})$ where the variable ?var2 is bound to the column value ($\text{Col}(\text{?var2}) \equiv \text{value}$) of the output relation from applying the NgetAttVal operator given that the mapping of the variable ?var ($\text{Map}(\text{?var})$) is bound to the column ID as a part of the input relation R . In these rules, $\text{Triple}(q)$ refers to the set of triple patterns of a query q , $\text{PathFilter}(q)$ refers to the set of path filters of a query q and allEdges (Rule OPMAP-8) refers to the materialized view that represents a union of all relations that store the graph edges. Fig. 11 illustrates an example algebraic compilation for the following 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. Examples of these rewriting rules which are specific to our introduced constructs are represented in Fig. 10. A sample interpretation of the inference

rule REWRITE-1 is that it rewrites a triple pattern $(\text{subject}, \text{?* var}, \text{object})$ of query q ($\text{Triple}(q)$) into the triple pattern $(\text{subject}, \text{??var}, \text{object})$ given that the path variable (?* var) is not part of the output variable list of the query q ($\text{outVarList}(q)$). In addition, this compilation steps reorders the query triple patterns according to their *restrictiveness* (the more restrictive pattern has higher precedence) in order to optimize the join computations based on the following rules. Let $\text{tp1}, \text{tp2} \in \text{Triple}(q)$ be two triple patterns of G-SPARQL query q .

- tp1 is defined as less restrictive than tp2 ($\text{tp1} \ll \text{tp2}$) if tp1 contains a higher number of variables (?? or ?*) than tp2 .
- tp1 is defined as more restrictive than tp2 ($\text{tp1} \ll \text{tp2}$) if tp1 contains a lower number of variables than tp2 .
- tp1 is defined as more restrictive than tp2 ($\text{tp1} \ll \text{tp2}$) if tp1 contains the same number of variables as tp2 and the number of filter expressions over the variables of tp1 is more than the number of filter expressions over the variables of tp2 .

4.2.2. 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 (Fig. 6), then we start optimizing the algebraic plans using a set of rules. These rules include the traditional rules for relational

$$\begin{array}{c}
 \frac{(\text{subject}, ?*\text{var}, \text{object}) \in \text{Triple}(q) \quad ?*\text{var} \notin \text{outVarList}(q)}{(\text{subject}, ?*\text{var}, \text{object}) \Rightarrow (\text{subject}, ??\text{var}, \text{object})} \quad (1) \\
 \\
 \frac{(\text{subject}, ?*\text{var}(\text{Predicate}), \text{object}) \in \text{Triple}(q) \quad ?*\text{var} \notin \text{outVarList}(q)}{(\text{subject}, ?*\text{var}(\text{Predicate}), \text{object}) \Rightarrow (\text{subject}, ??\text{var}(\text{Predicate}), \text{object})} \quad (2) \\
 \\
 \frac{\begin{array}{l} (\text{subject}, ?\text{var}(\text{Predicate}), \text{object}) \in \text{Triple}(q) \\ ?\text{var} \notin \text{outVarList}(q) \\ (?*\text{var}, @\text{literal}, \text{literal}) \notin \text{Triple}(q) \\ (?*\text{var}, @\text{literal}, ?\text{var2}) \notin \text{Triple}(q) \end{array}}{(\text{subject}, ?*\text{var}(\text{Predicate}), \text{object}) \Rightarrow (\text{subject}, \text{Predicate}, \text{object})} \quad (3) \\
 \\
 \frac{(\text{subject}, ?*\text{var}, \text{object}) \in \text{Triple}(q) \quad (\text{Length}(?*\text{var}, =1)) \in \text{PathPredicate}(q)}{(\text{subject}, ?*\text{var}, \text{object}) \Rightarrow (\text{subject}, ?\text{var}, \text{object})} \quad (4) \\
 \\
 \frac{(\text{subject}, ?*\text{var}(\text{Predicate}), \text{object}) \in \text{Triple}(q) \quad (\text{Length}(?*\text{var}, =1)) \in \text{PathPredicate}(q)}{(\text{subject}, ?*\text{var}(\text{Predicate}), \text{object}) \Rightarrow (\text{subject}, ?\text{var}(\text{Predicate}), \text{object})} \quad (5) \\
 \\
 \frac{(\text{subject}, ??\text{var}, \text{object}) \in \text{Triple}(q) \quad (\text{Length}(??\text{var}, =1)) \in \text{PathPredicate}(q)}{(\text{subject}, ??\text{var}, \text{object}) \Rightarrow (\text{subject}, ?\text{var}, \text{object})} \quad (6) \\
 \\
 \frac{(\text{subject}, ??\text{var}(\text{Predicate}), \text{object}) \in \text{Triple}(q) \quad (\text{Length}(??\text{var}, =1)) \in \text{PathPredicate}(q)}{(\text{subject}, ??\text{var}(\text{Predicate}), \text{object}) \Rightarrow (\text{subject}, ?\text{var}(\text{Predicate}), \text{object})} \quad (7)
 \end{array}$$

Fig. 10. G-SPARQL query rewriting rules.

algebraic optimization (e.g. pushing the selection operators down the plan) [19,40] 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 [23]. In particular, our execution split mechanism makes use of the following two main heuristics:

- 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 well-established 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.

- 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 Fig. 11, 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 [22]. This climbing process for each path stops and the intermediate results are materialized if it hits one of the operators that does not have 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 Fig. 11, all operators can be translated into standard relational operators except for the *pathJoin*

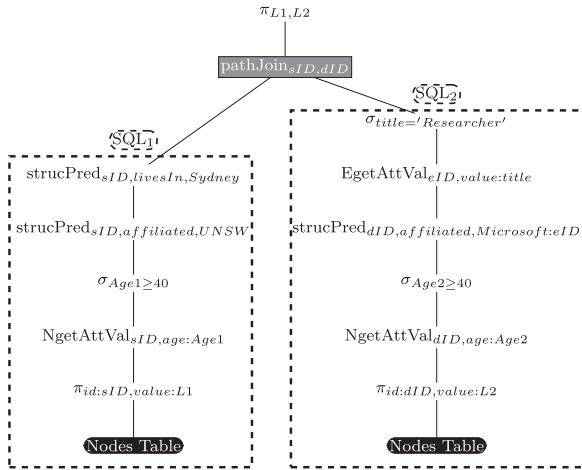


Fig. 11. An example DAG plan for G-SPARQL.

```

SQL1: SELECT N1.ID as sID, N1.Value as L1
FROM NodeLabel as N1, age, affiliated, livesIn,
NodeLabel as N2, NodeLabel as N3
WHERE N1.ID = age.ID AND N1.ID = affiliated.sID AND
affiliated.dID = N2.ID AND N2.value = "UNSW" AND N1.ID =
livesIn.sID AND livesIn.dID = N3.ID and N3.Value =
"Sydney" AND age.Value >= 40;

```

```

SQL2: SELECT N1.ID as dID, N1.Value as L2
FROM NodeLabel as N1, age, title, affiliated, NodeLabel
as N2
WHERE N1.ID = age.ID AND N1.ID = affiliated.sID AND
affiliated.dID = N2.ID AND N2.value = "Microsoft" AND
affiliated.eID= title.ID AND title.Value = "Researcher"
AND age.Value >= 40;

```

Fig. 12. SQL translations of an example DAG plan shown in Fig. 11.

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 (Fig. 12) 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 [15]. 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. For example, we rely on BFS to implement the evaluation of reachability. However, we can make use of any indexing information (such as the 2-hop cover or the 3-hop cover) to accelerate the physical execution of the logical operator. Similarly, we rely on the Dijkstra algorithm to implement the shortest path operator. However, other indexing methods can be utilized as well. It should be noted that the indexing methods that may be used for supporting the evaluation of the reachability operators or shortest path operators can be only considered in the cases where there are no restrictions or

conditions on the nodes and edges of the results of the operators as these indexing methods usually do not consider such functionalities in their design.

5. Performance evaluation

We implemented a native pointer-based memory representation of the graph topology in addition to the Dijkstra and BFS algorithms (with slightly different variants that can represent each of our non-relational algebraic operators (Table 1)) 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 that because of the expressiveness of our language, we were unable to consider either the traditional RDF query processors [37,64] (they do not support path patterns, reachability or shortest path queries) or traditional structural graph indexing and querying techniques [13,30,59,58,62] (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.

5.1. Results on real dataset

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 (*author*, *article*, *series*, *conference*, *proceedings*, *journal*, *issue* and *publisher*). The original data describes 12 different types of relationships between the graph entities (e.g. *authorOf*, *editorOf*, *publishedBy*, *citedBy*, *partOfIssue*, *partOfProceedings*). In addition, we have created the *co-author* relationships (edges) between any two *author* nodes which are sharing the authorship of at least one paper. Each created edge is described by a *noPapers*

Table 2
Characteristics of real datasets.

Dataset	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

attribute that represents the number of joint papers between the connected authors. The original dataset has a total of 76 unique attributes, of which 62 attributes are describing the graph vertices and 14 attributes are describing the graph edges. In addition, for each author, we created a *prolific* attribute where authors with more than 25 papers are labeled as highly prolific. In addition, for each *citedBy* edge, we created a *source* attribute which is labeled as an external citation if there is an empty intersection between the author lists of the two connected papers. In our experiments, we used three different sizes

of graph subsets (small, medium and large) in order to test the scalability of our approach. The *small* subset represents all journal and magazine publications, the *medium* subset adds the newsletter and transaction publications while the *large* subset adds the conference proceedings. Table 2 lists the characteristics of the three sets. On average, the graph topology information that needs to be loaded onto the main memory represents 10% of the whole graph information (topology + values of attributes of graph nodes and edges) while the remaining 90% resides in the underlying relational database.

	Pattern Matching
Q1	<i>Find the names of two authors who are highly prolific, affiliated at "affiliation 1" and "affiliation 2" respectively, and jointly co-authored more than two papers.</i> SELECT ?Name1 ?Name2 WHERE {?X @name ?Name1. ?Y @name ?Name2. ?? @prolific "High". ?X @prolific "High". ?X @affiliation "%affiliation1%". ?Y @affiliation "%affiliation2%". ?X ?E(co-author) ?Y. ?E @noPapers ?NO. FILTER (?NO >= 2.) }
Q2	Pattern Matching + Reachability <i>Find the names of two authors who are highly prolific, affiliated at "affiliation 1" and "affiliation 2" respectively, and are connected by a path of any length where all edges of the path represent the co-author relationship.</i> SELECT ?Name1 ?Name2 WHERE {?X @name ?Name1. ?Y @name ?Name2. ?X @prolific "High". ?Y @prolific "High". ?X @affiliation "%affiliation1%". ?Y @affiliation "%affiliation2%". ?X co-author+ ?Y. }
Q3	Pattern Matching + Reachability (With Projection) <i>Find the names of two authors and the connecting path between them where the two authors are highly prolific, affiliated at "affiliation 1" and "affiliation 2" respectively. The connecting path can be of any length where all edges of the path represent the co-author relationship.</i> SELECT ?Name1 ?Name2 ??P WHERE {?X @name ?Name1. ?Y @name ?Name2. ?X @prolific "High". ?Y @prolific "High". ?X @affiliation "%affiliation1%". ?Y @affiliation "%affiliation2%". ?X ??P(co-author+) ?Y. }
Q4	Pattern Matching + Shortest Path (With Projection) <i>Find the names of two authors and the shortest connecting path between them where the two authors are highly prolific, affiliated at "affiliation 1" and "affiliation 2" respectively. The connecting path can be of any length where all edges of the path represent the co-author relationship.</i> SELECT ?Name1 ?Name2 ??P WHERE {?X @name ?Name1. ?Y @name ?Name2. ?X @prolific "High". ?Y @prolific "High". ?X @affiliation "%affiliation1%". ?Y @affiliation "%affiliation2%". ?X ?P(co-author+) ?Y. }
Q5	Pattern Matching + Reachability (With Projection) <i>Find the names of two authors and the connecting path between them where the authors are highly prolific, affiliated at "affiliation 1" and "affiliation 2" respectively. The connecting path can be of any length where each edge can represent any relationship.</i> SELECT ?Name1 ?Name2 ??P WHERE {?X @name ?Name1. ?Y @name ?Name2. ?X @prolific "High". ?Y @prolific "High". ?X @affiliation "%affiliation1%". ?Y @affiliation "%affiliation2%". ?X ??P ?Y. }
Q6	Pattern Matching + Reachability (With Projection) + Path Filtering <i>Find the names of two authors and the connecting path between them where the authors are highly prolific, affiliated at "affiliation 1" and "affiliation 2" respectively. The connecting path length can not be of more than 3 edges where each edge of the path represent the co-author relationship.</i> SELECT ?Name1 ?Name2 ??P WHERE {?X @name ?Name1. ?Y @name ?Name2. ?X @prolific "High". ?Y @prolific "High". ?X @affiliation "%affiliation1%". ?Y @affiliation "%affiliation2%". ?X ??P(co-author+) ?Y. FilterPath(Length(?P, <= 3).)}
Q7	Pattern Matching + Reachability (With Projection) + Path Filtering <i>Find the names of two authors and the path between them where the authors are highly prolific, affiliated at "affiliation 1" and "affiliation 2" respectively where all nodes of the connecting path should fulfill the condition to be highly prolific and each edge of the path represent the co-author relationship.</i> SELECT ?Name1 ?Name2 ??P WHERE {?X @name ?Name1. ?Y @name ?Name2. ?X @prolific "High". ?Y @prolific "High". ?X @affiliation "%affiliation1%". ?Y @affiliation "%affiliation2%". ?X ??P(co-author+) ?Y. FilterPath(AllNodes(??P, @prolific "High").)}
Q8	Pattern Matching <i>Find the name of an author, the title of a paper, the year of an issue where the author is the first author of the paper and is highly prolific, the paper has a keyword "keyword" and is part of a journal issue which has a code equals to "code".</i> 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	Pattern Matching <i>Find the title of a paper (P) and the name of three highly prolific authors (AU1, AU2, AU3) where the paper (P) is cited by three papers (P1, P2, P3) such that: P1 is authored by AU1, P2 is authored by AU2 and P3 is authored by AU3.</i> 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	Pattern Matching + Reachability + Path Filtering <i>Find the titles of two papers, Paper1 and Paper2, where both of the papers are described by the keyword "keyword", Paper1 is authored by "name" and Paper1 is connected to Paper2 with a path of length which is less than or equal 2 edges where each edge represents the citedBy relationship.</i> 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	Pattern Matching + Reachability + Path Filtering <i>Find the titles of two papers, Paper1 and Paper2, where both of the paper are described by the keyword "keyword", Paper1 is authored by "name" and Paper1 is connected to Paper2 with a path of length which is less than or equal 4 edges where each edge represents the citedBy relationship of an external source of citation.</i> 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 "External")).}
Q12	Pattern Matching + Reachability + Path Filtering <i>Find the name of an author who is connected to two authors with names "name1" and "name2" where the connecting path with each of two authors is of length which is less than or equal to two edges that are representing the co-author relationship. In addition, all the co-author edges of the connecting path with the author "name1" are described by a "@noPapers" attribute with a value which is greater than 2.</i> 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).)}

Fig. 13. Query templates of our experimental workload.

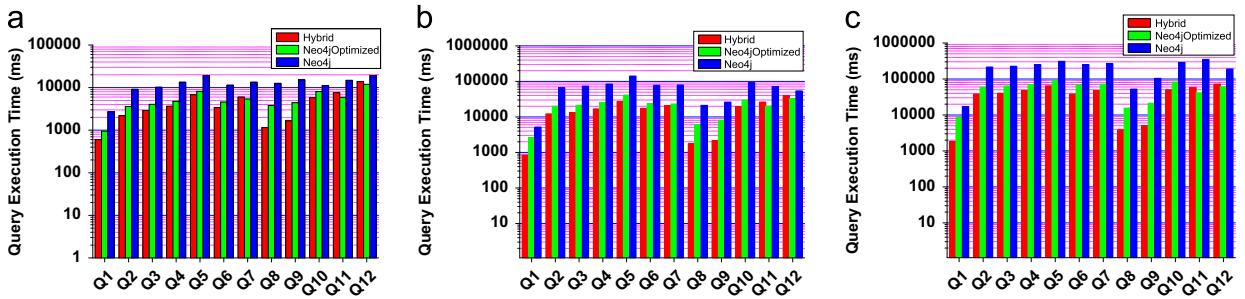


Fig. 14. Average query evaluation times of real datasets: (a) small graph size; (b) medium graph size; and (c) large graph size.

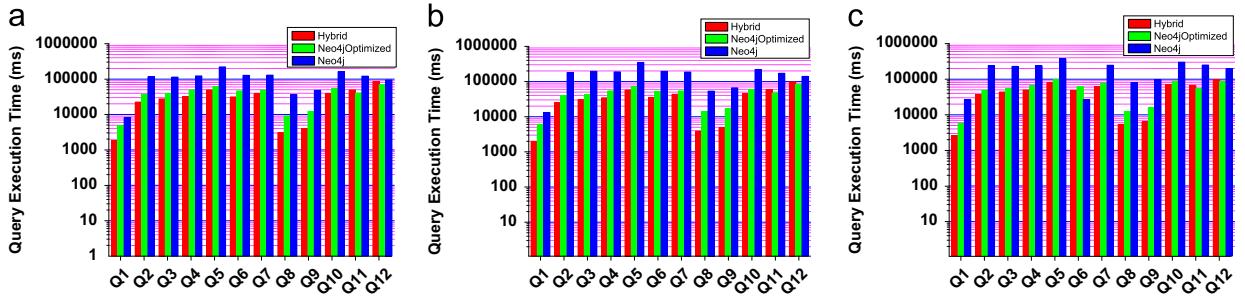


Fig. 15. Average query evaluation times of synthetic datasets: (a) Synthetic1; (b) Synthetic2; and (c) Synthetic3.

Query Workload: Our query workload consists of 12 query templates (Fig. 13) 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*. The algebraic plans of the query templates are described in the Appendix section of this paper. As we 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. The execution times of these implementations can dramatically vary. 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. Each instantiation is executed 5 times over our implementation and the two associated Neo4j implementations, and execution times were collected. All times are in milliseconds. In order to ensure that any caching or system process activity would not affect the collected results, the longest and shortest times for each instantiation were dropped and the remaining three execution times for the 20 instantiations were averaged.

Query evaluation times: The average query evaluation times for the 20 instances of each of the 12 query templates are shown in Fig. 14 for the small (Fig. 15(a)), medium (Fig. 15(b)) and large (Fig. 15(c)) sized experimental 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 of the experimental graph, 4 times faster on the medium subset of the experimental graph 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. In native graph database systems, such as Neo4j, users are required to implement their graph queries using the available system APIs. The implementation process, in fact, defines a specific query plan. Therefore, when users have sufficient knowledge about the characteristics of the underlying graph and query optimization rules they can define very efficient query plans. The results of the experiments show that the average query evaluation times of our approach is 17% faster than the Neo4j optimized implementations on the small subset of the experimental graph, 22% faster on the medium subset and 28% faster on the large experimental graph. The Neo4j optimized implementations outperform 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 Lucene⁷ which is more optimized for full-text indexing rather than traditional data retrieval queries.

⁷ <http://lucene.apache.org/>

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 where, in our approach, the whole of their executions can be pushed inside the underlying RDBMS. In particular, Q1 seeks for a pattern of two vertices (`authors`) which are connected by a `co-author` relationship. The query involves 5 value-based predicates where each of the two queried vertices are filtered based on the 2 value-based predicates on the `affiliation` and `prolific` attributes and the connecting edges are filtered based on a value-based predicate on their `noPapers` attribute. Q8 specifies a *path pattern* that consists of 4 vertices (`author`, `paper`, `issue`, `Journal`) that are connected by 3 relationships (`authorOf`, `partOf`, `issueOf`) respectively. The query involves 4 value-based predicates on the `prolific` attribute of the `author` vertex, the `keyword` attribute of the `paper` vertex, the `code` attribute of the `journal` vertex and the `seqNo` attribute of the `authorOf` relationship. Q9 represents a *star pattern* that seeks for `paper` vertices which are filtered based on a specific `keyword` and cited by 3 `papers` where each `paper` has an `author` which is `prolific`. 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 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 (`pathJoinsID,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 (`sPathJoinsID,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 (`pathJoinsID,dID:eID`). Obviously, allowing the edges of the resulting connecting paths to be representing any relationship increases the number of visited and verified paths and consequently the number of traversing operations over the graph topology. 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, except for the case of Q7 on the small graph size, 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-based topology information for fast execution of the required traversal operations and avoids loading/accessing unnecessary information that are not involved in evaluated queries.

In our approach, the execution of the path filtering condition of Q7 is an expensive operation as it can be only 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 contain any node that does not satisfy the filtering condition over the retrieved attribute values. This SQL statement can be depicted as follows:

```
SELECT Value FROM prolific WHERE id IN
([list of node identifiers that are members of the
connecting paths]);
```

on the contrary, the cost of 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 the query seeks for all authors that are connected to two authors, x and y , where the length of the connecting co-author paths between the target authors and both x and y do not exceed two edges and all edges of the connecting paths to x are described by `noPapers` greater than 2.

In general, one of the main limitations of our approach is the inefficiency of evaluating path filtering predicates over a set of paths resulting from evaluating path operators (e.g. `pathJoin`, `sPathJoin`). The main reason behind this is that we load onto the main memory only the graph topology information (the attributes of the graph nodes/edges are not loaded and only available in the underlying relational database). Therefore, the evaluation of these predicates requires late retrieval of the required values for the attributes (using SQL statement of the form (`SELECT ... IN ...`) of the set of nodes/edges over which the predicate condition needs to be applied. In practice, the efficiency of this retrieval operation is affected by the size of this intermediate set. The retrieval of these attribute values using the SQL statement (`SELECT ... IN ...`) cannot make effective use of the existing database indices. On the other side, Neo4j loads the whole traversed nodes/edges information (attributes) onto the memory during the execution of the graph traversal operations and thus the execution of the path filtering information is much more efficient by paying of a higher memory consumption. 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 main memory consumption and the performance that can be gained on evaluating the path filtering conditions by loading more attributes of the graph nodes/edges. In addition, effective determination of which attributes should be loaded onto the memory would require pre-known knowledge about the characteristics of the query workloads. On the contrary, evaluating filtering condition on the path length can be achieved very efficiently as the path length information is computed during the traversal operations for evaluating the path operators and does not require any data retrieval from the underlying database.

5.2. More results on synthetic datasets

We have also evaluated our approach and Neo4j implementations using synthetic graphs. A set of synthetic graph datasets is generated by our implemented data generator following the same idea of the *R-MAT* generator [9]. The generator allows the user to specify the number of vertices (V), the average number of outgoing edges for each vertex (E), the average number of attributes for each graph node or edge (A), the total number of distinct relationships (R) and the total number of distinct attributes (T) of graph nodes and edges. We use the notation $VvEeAaRrTt$ to represent the generation parameter of each dataset. Table 3 illustrates the characteristics of synthetic graph datasets that are used in our experiments.

The query workload of the synthetic graph datasets has been generated using the 12 query templates (Fig. 13) of the real dataset where we used random assignment of the queried graph attributes and edges in addition to random literal values to generate the different query instances. Similar to the experiments of the real datasets, each query template is instantiated 20 times and each instance is executed 5 times. The average query evaluation times for the 20 instances of each of the 12 query templates over the synthetic graph datasets are shown in Fig. 15. The results of the experiments show the scalability of our approach with regards to the increasing sizes of either the topology information or the data information of the graph database. For example, the generation parameters of the dataset *Synthetic2* are equal to the generation parameters of the dataset *Synthetic1* with only one difference that the value of the average number of outgoing edges for each vertex (E) of *Synthetic2* is double the value of the same parameter in *Synthetic1*. Such an increase in the value of (E) leads to an increase in the cost of all operators that involve traversal operations over the graph topology (e.g. `pathJoin` and `sPathJoin`) especially for the queries that do not restrict the traversal operation with a specified relationship (e.g. Q5). The average percentage of increase (between *Synthetic2* and *Synthetic1*) in the execution times for evaluating the queries in our approach (16%) is less than the average percentage of time increase for Neo4j implementations (19% for optimized implementations and 32% for non-optimized implementations) due to leveraging the advantage of our approach in holding the whole graph topology information in the main memory. The percentages of increase in the execution times for evaluating the queries between *Synthetic2* and *Synthetic1* for the different approaches are computed using

Table 3
Characteristics of synthetic graph datasets.

Dataset	Vertices	Edges	Att. values	Topology info. (%)
Synthetic1 <i>(V500kE5A3R50T150 Relational Representation of)</i>	500 K	2.5 (M)	15 (M)	21
Synthetic2 <i>(V500kE10A3R50T150)</i>	500 K	5 (M)	17.5 (M)	29
Synthetic3 <i>(V1000kE5A5R75T250)</i>	1 (M)	5 (M)	30 (M)	23

the following formula:

$$\frac{\text{Time}(\text{Synthetic2}) - \text{Time}(\text{Synthetic1})}{\text{Time}(\text{Synthetic1})}$$

On the other side, the total number of vertices, edges and attribute values of *Synthetic3* dataset is twice the total number of vertices, edges and attribute values of *Synthetic1* dataset respectively. However, the average percentage of increase (between *Synthetic3* and *Synthetic1*) in the execution times for evaluating the queries in our approach is 31%. In principle, while the evaluation costs of the parts of the query plans that can be pushed inside the relational database can make use of its well-known scalability in optimizing the performance of processing large datasets (by relying on its built-in indexing and query optimization techniques), the evaluation costs for the parts of the query plans which are executed by main memory operators increase linearly with the size of the processing datasets. For example, in Q2, Q3, Q4, Q5, Q6 and Q7, increasing the size of the graph database leads to an increase in the size of the input relations of the main memory operator *pathJoin* and consequently increases the number of join operations and the number of graph traversal operations. The average percentage of time increase (between *Synthetic3* and *Synthetic1*) in evaluating the queries using the optimized and non-optimized Neo4j implementations is 38% and 44% respectively.

6. Related Work

The field of graph data management has attracted a lot of attention from the database research community in recent years because of the emerging wave of new applications in different domains. Several graph querying techniques have been proposed in the literature for handling different types of graph queries over large graphs such as pattern matching query [62,63], reachability query [13,30] and shortest path query [12,57]. SPath [62] and GADDI [61] are presented as subgraph search techniques over large graphs. These methods rely on constructing some indices to prune the search space of each vertex to reduce the whole search space. For example, SPath [62] constructs an index by neighborhood signature which utilizes the shortest paths within the k-neighborhood subgraph of each vertex. GADDI [61] index is based on neighboring discriminating substructure (NDS) distances which needs to count the number of some small discriminating substructures. Some other techniques have been proposed for handling graph reachability queries [13,30]. These techniques are mainly designed to efficiently check whether there exist any path connections (sequence of edges) from a vertex u to another vertex v in a large directed graph. For example, Cohen et al. [13] label a graph based on the so-called 2-hop covers. In this method, each hop is a pair (h, v) where h is a path in G and v is one of the endpoints of h . Hence, if there are some paths from u to v , there must exist $(h1, u)$ and $(h2, v)$ and one of the paths between v and u would be the concatenation of $h1$ and $h2$. The 3-hop indexing scheme [30] uses chain structures in combination with hops to minimize the number of structures that must be indexed. Thus, it does not need to compute the entire transitive closure. Instead, it only needs to compute and record a number of so-called *contour* vertex pairs which can be much smaller than the size of the

transitive closure. The *TEDI* indexing structure [57] has been proposed for answering shortest path queries. TEDI is based on a tree decomposition methodology where the graph is decomposed into a tree in which each node represents a bag that contains more than one vertex from the graph. Based on this index, a bottom-up operation is executed to find the shortest path between any two vertices in the graph.

In general, these techniques are mainly focusing on querying the topological structures of the underlying databases and usually ignore the attributes of vertices and edges in the querying process. Each of these techniques assumes a specific organization of its indexing structure which is mainly designed and optimized for supporting a specific target type of graph queries without any consideration for the requirement of the other types of graph queries. In addition, these techniques are very expensive in their memory consumption so that they cannot scale to support very large graphs. Moreover, they rely on a very expensive offline pre-processing step for building their indexing structures. Usually, the maintenance of these indices is very expensive especially in the case of dynamic graph databases. However, our approach can effectively utilize the information of any of these indices (if available) to accelerate the physical evaluation of our logical operator.

Several languages have been proposed for querying different types for graph models with different purposes and querying abstractions. For example, the *Lorel* language [2] has been designed for querying semistructured data. Lorel follows an SQL/OQL style for querying a simple object model that can be viewed as an extension of the Object Data Management Group (ODMG) data model⁸ [49] that is another extension of the OQL languages which is used for querying object-oriented model that supports graphs. GOOD [26] is graph query language which is also based on an object-oriented model. The query mechanism of Good uses graph transformations such as node addition/deletion and edge addition/deletion. GOOD also provides an abstraction mechanism to group objects by means of their properties, as well as methods for defining sequences of operations. The *GraphDB* [25] language has been mainly focusing on querying spatial networks such as transportation systems. The language requires the presence of a graph schema. GraphDB includes object-oriented features such as classes for nodes and edges, as well as paths. It provides regular expressions which are defined over alternating sequences of node and edge types. It also provides built-in operators such as one for shortest path. The *G* language [16] is used for querying labelled and directed graphs. The semantics of *G* was defined in terms of matching simple paths in the graph being queried. *GraphQL* [27] is a graph query language that uses a graph pattern as the basic operational unit. It has been designed for querying and manipulating labeled directed graphs. The core of GraphQL is a graph algebra in which the selection operator is generalized to graph pattern matching and a composition operator is introduced for rewriting matched graphs. Recently, the Neo4j graph database system has introduced *Cypher*⁹ as a declarative graph query language that allows querying and updating of

⁸ <http://www.objs.com/x3h7/odmg.htm>. GOQL

⁹ <http://docs.neo4j.org/chunked/stable/cypher-introduction.html>

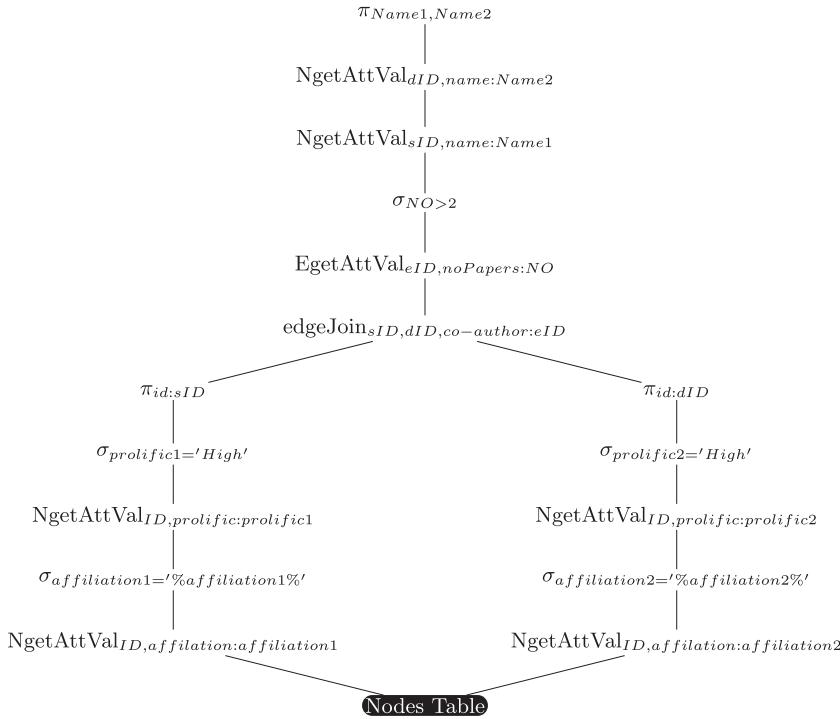


Fig. A1. Algebraic plan of Q1.

the graph store without having to write traversals through the graph structure in code. Cypher is mainly inspired by the syntax of the SQL and SPARQL query languages. However, Cypher is still growing and maturing and its syntax is still going through fundamental breaking changes. WIPE (Weakly-structured Information Processing and Exploration) [43] is a recently proposed data manipulation and query language which has been built on top of the graph functionality in the SAP HANA database [32,51]. WIPE combines support for graph traversal and manipulation with BI-like data aggregation. The language allows the declaration of multiple insert, update, delete, and query operations in one complex statement.

Several graph query processing techniques were proposed on large RDF graphs [1,7,37,64]. In general, most of these techniques have been mainly focusing on supporting the graph matching mechanism of the SPARQL query language. For example, the *RDF-3X* query engine [37] stores the whole RDF graph in a three-column table (subject, predicate, object) and tries to overcome the expensive cost of self-joins by building indices over all 6 permutations of the three dimensions that constitute an RDF triple. The query optimizer relies upon its cost model in finding the lowest-cost execution plan and mostly focuses on join order and the generation of execution plans [36]. *SW-Store* [1] is an RDF storage system which uses a fully decomposed storage model (DSM) [14] where the RDF triples table is rewritten into n two-column tables where n is the number of unique properties in the RDF dataset. The implementation of *SW-Store* relies on a column-oriented DBMS, C-store [52], to store tables as collections of columns rather than as collections of rows as usually followed in the standard row-oriented database systems (e.g. Oracle, DB2, SQLServer, Postgres, etc.) where the entire tuples are

stored consecutively. Abadi et al. [1] reported that storing RDF data in column-store database is better than that of row-store database while Sidirorgos et al. [50] have shown that the gain of performance in column-store database depends on the number of predicates in the RDF dataset. The latest version of IBM DB2 (version 10) has provided a new NoSQL Graph Store feature that supports storing the standard RDF triples inside DB2.¹⁰ ¹¹ Similarly, the latest version of Oracle database system (11 g) supports the RDF graph feature by providing a Java-based interface to store and query semantic data in the Oracle NoSQL Database.¹² In particular, the RDF graph triples are stored as key-value pairs in Oracle NoSQL Database and the Apache Jena APIs¹³ is used for interfacing with the stored graph data. It also supports using Apache Jena Joseki,¹⁴ an open source query processor for the standard SPARQL language, for creating SPARQL queries where each SPARQL query is translated into a multi-get API call against the Oracle NoSQL Database and the resulting key-value pairs are combined to form a final result set for the client. *DOGMA* [7] is a disk-based graph index for RDF databases. It represents a generalization of the well known binary-tree indexing

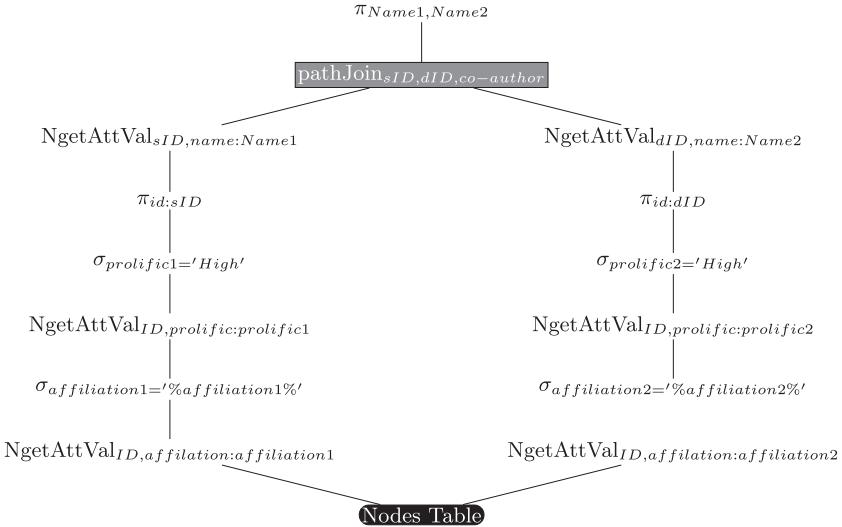
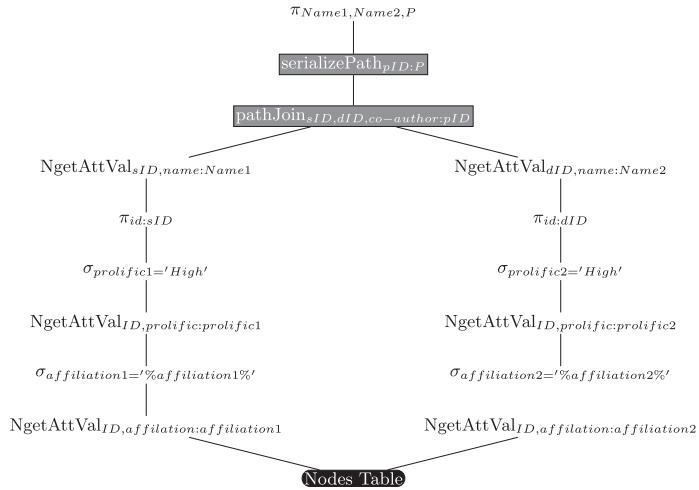
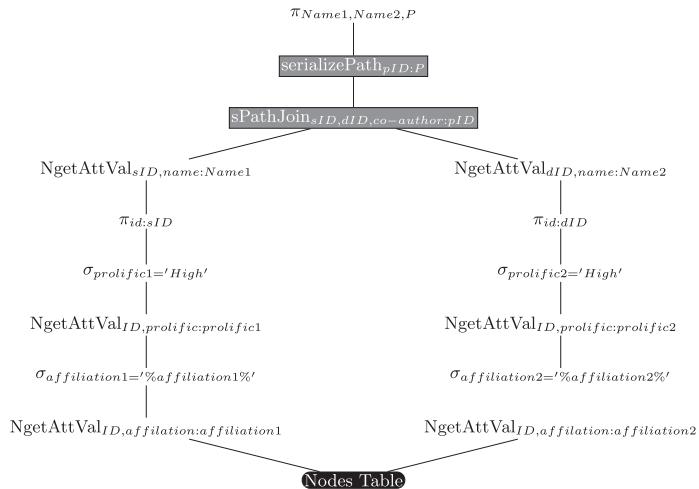
¹⁰ <http://www-01.ibm.com/software/data/db2/linux-unix-windows/nosql-support.html>. In particular, it maps the graph triples into 5 relational tables (*Direct Primary*, *Direct Secondary*, *Reverse Primary*, *Reverse Secondary*, *longStrings*). The DB2 query processor for the standard SPARQL languages is optimized for executing graph pattern matching queries which involves a lot of star patterns.

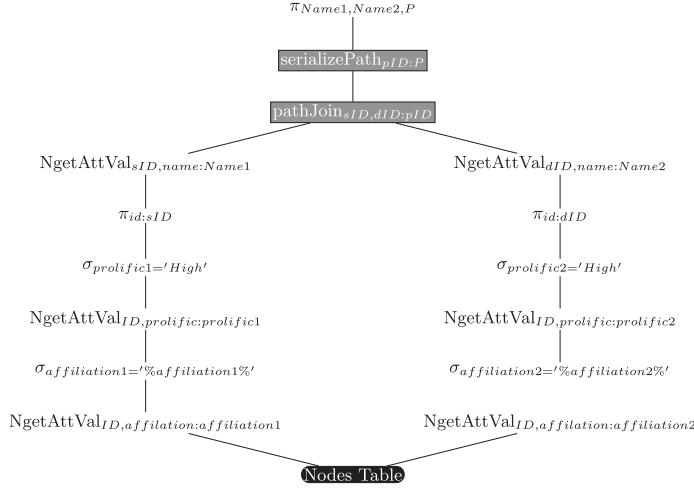
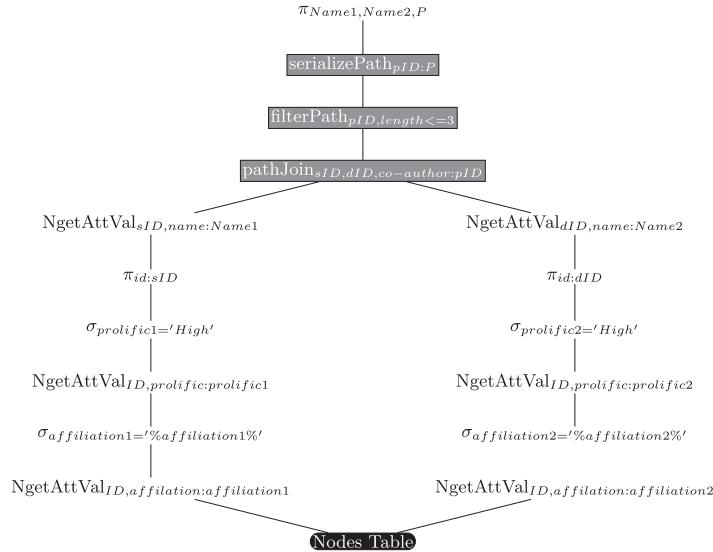
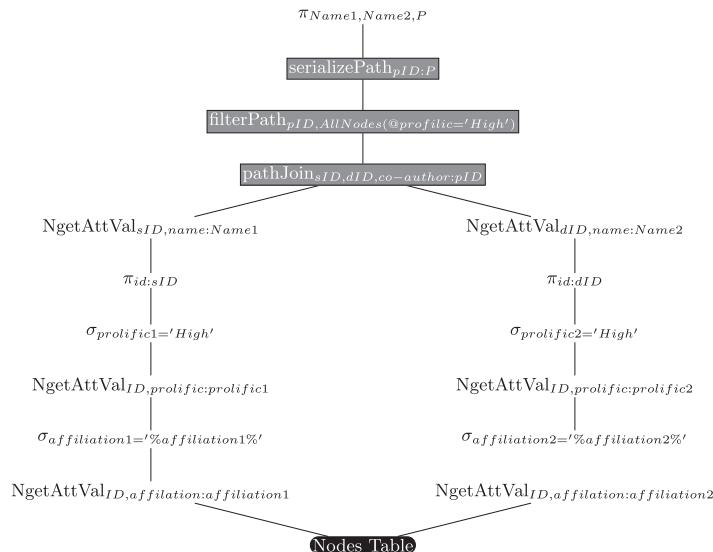
¹¹ https://www.ibm.com/developerworks/mydeveloperworks/blogs/nlp/resource/DB2_NoSQLGraphStore.pdf.

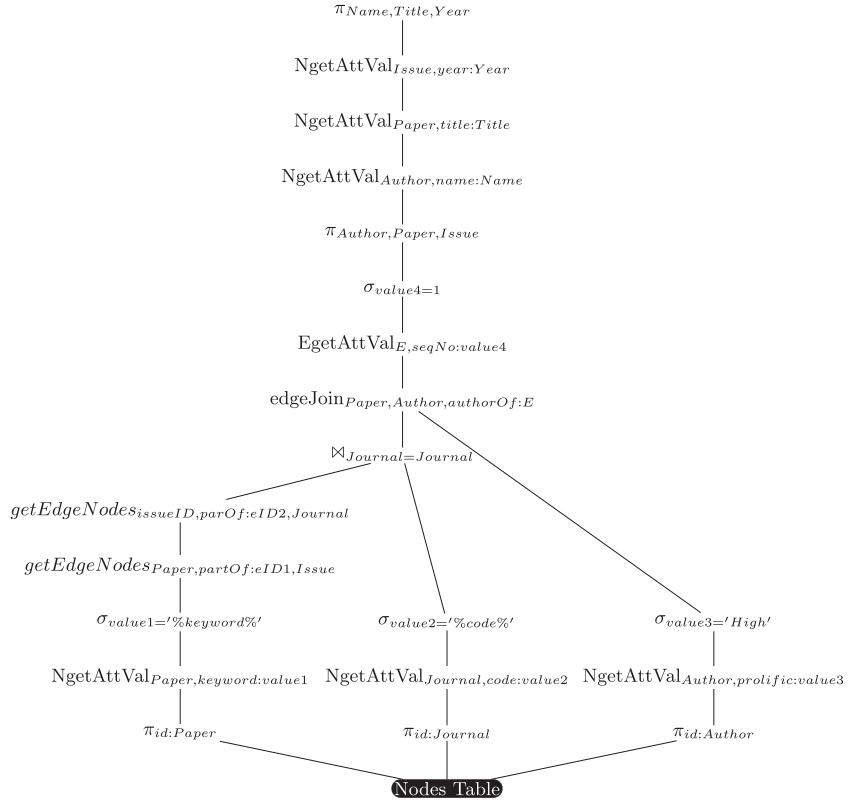
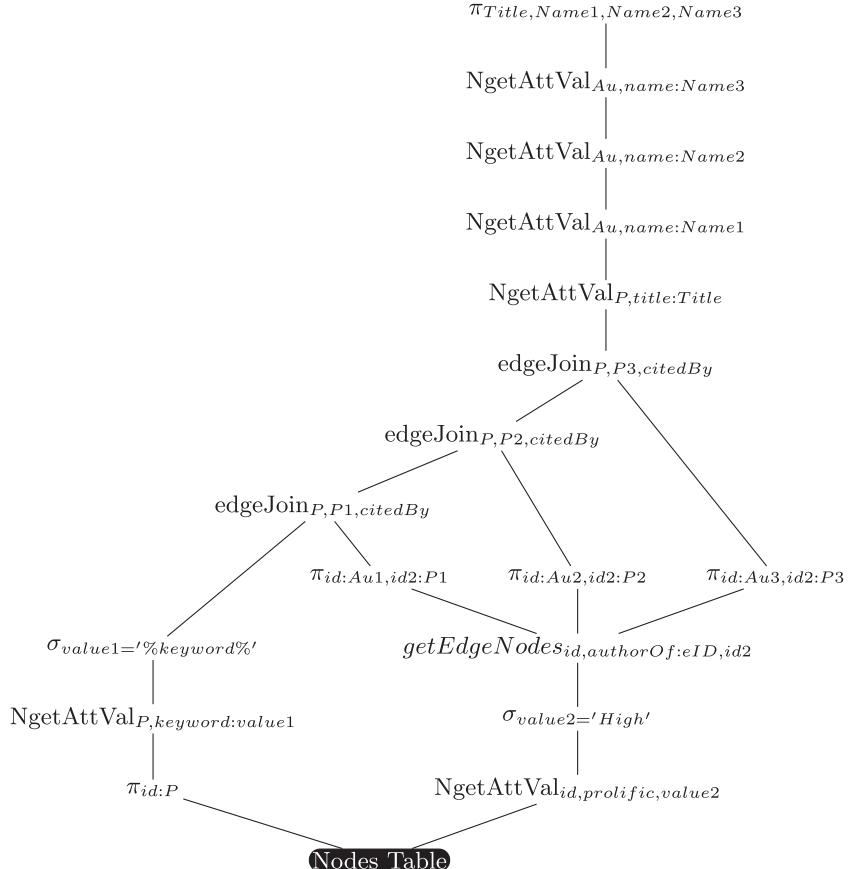
¹² <http://www.oracle.com/technetwork/database/options/semantic-tech/index.html>

¹³ <http://jena.apache.org/>

¹⁴ http://jena.apache.org/documentation/serving_data/

**Fig. A2.** Algebraic plan of Q2.**Fig. A3.** Algebraic plan of Q3.**Fig. A4.** Algebraic plan of Q4.

**Fig. A5.** Algebraic plan of Q5.**Fig. A6.** Algebraic plan of Q6.**Fig. A7.** Algebraic plan of Q7.

**Fig. A8.** Algebraic plan of Q8.**Fig. A9.** Algebraic plan of Q9.

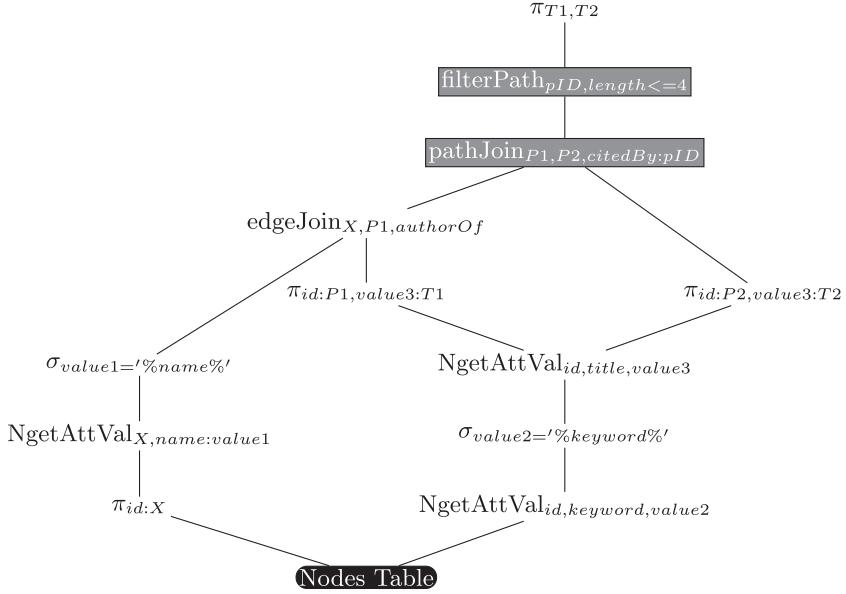


Fig. A10. Algebraic plan of Q10.

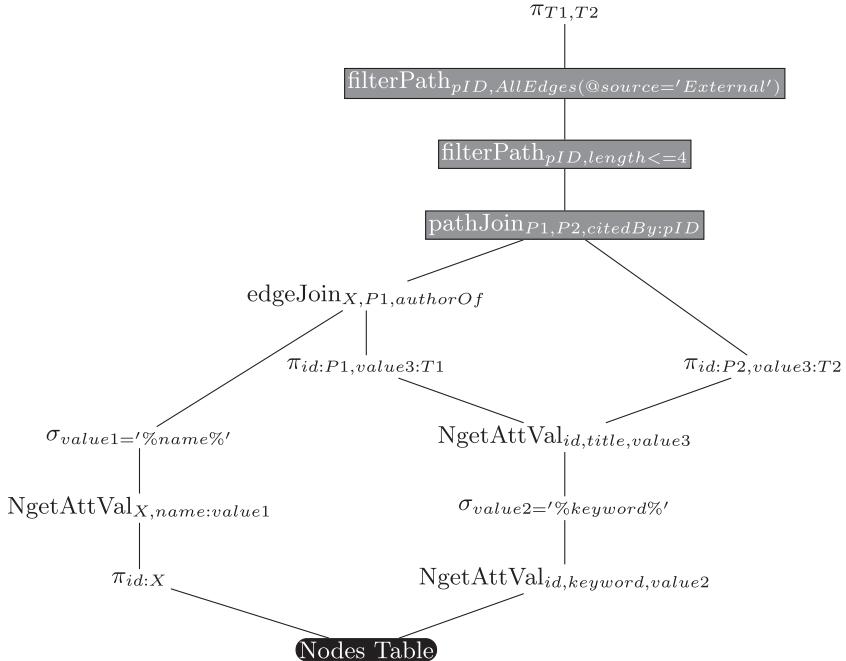


Fig. A11. Algebraic plan of Q11.

structure where each node occupies a disk page and is labeled by a graph that captures its two children. Therefore, if a node N has two children, N_1 and N_2 , then the graph labeling node N would be a k -merge of the graphs labeling its children. The gStore index [64] stores an RDF graph as a disk-based adjacency list table where each vertex in the RDF graph is assigned a bitstring as its vertex signature according to its adjacent edge labels and neighbor vertex labels. Hence, an RDF graph is converted into a data signature graph. During query processing, the input query is encoded in the same way

and a matching process is applied. In general, RDF graphs represent a special kind of attributed graphs that are considered in our approach. Therefore, our graph representation and algebraic-based querying methods are more general and can be easily adopted to answer queries on large RDF graphs.

The idea of splitting the execution of database queries between two different environments has been previously employed in the HadoopDB system [3] which has been designed as a query engine with a hybrid architecture for processing data warehousing queries and analytical

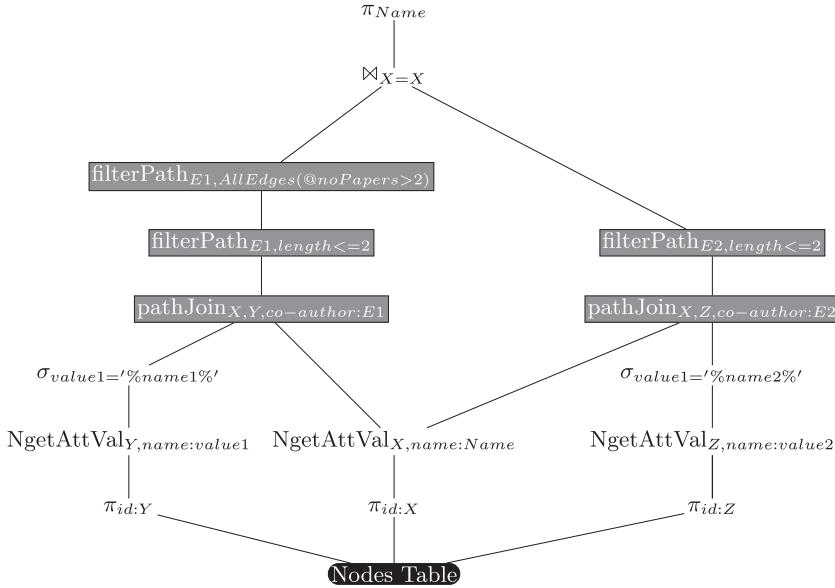


Fig. A12. Algebraic plan of Q12.

workloads using the MapReduce framework and relational database.

7. Conclusions

We presented *G-SPARQL*, a novel language for querying attributed graphs. The language has a number of appealing features. It 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 representation of large attributed graphs 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 mechanism for our supported queries based on an extended form of the standard relational algebra and a split of execution mechanism for the generated query plans. Experimental studies on real and synthetic graphs validated the efficiency and scalability of our approach. As a future work, we are planning to extend our approach to support the recently introduced SPARQL 1.1 features such as nested queries and aggregate queries.

Appendix A

Figs. A1–A12.

References

- [1] D. Abadi, A. Marcus, S. Madden, K. Hollenbach, Scalable semantic web data management using vertical partitioning, in: VLDB, 2007.
- [2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener, The Lorel query language for semistructured data, *Int. J. Digital Libr.* 1 (1) (1997).
- [3] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Rasin, A. Silberschatz, HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads, *PVLDB* 2 (1) (2009).
- [4] F. Alkhateeb, J. Baget, J. Euzenat, Extending SPARQL with regular expression patterns, *J. Web Sem.* 7 (2) (2009).
- [5] R. Angles, C. Gutiérrez, Survey of graph database models, *ACM Comput. Surv.* 40 (1) (2008).
- [6] K. Anyanwu, A. Maduko, A. Sheth, SPARQ2L: towards support for subgraph extraction queries in RDF databases, in: WWW, 2007.
- [7] M. Bröcheler, A. Pugliese, V.S. Subrahmanian, DOGMA: a disk-oriented graph matching algorithm for RDF databases, in: ISWC, 2009.
- [8] D. Cai, Z. Shao, X. He, X. Yan, J. Han, Community mining from multi-relational networks, in: PKDD, 2005.
- [9] D. Chakrabarti, Y. Zhan, C. Faloutsos, R-MAT: a recursive model for graph mining, in: SDM, 2004.
- [10] A. Chebotko, S. Lu, F. Fotouhi, Semantics preserving SPARQL-to-SQL translation, *DKE* 68 (10) (2009).
- [11] C. Chen, X. Yan, P. Yu, J. Han, D. Zhang, X. Gu, Towards graph containment search and indexing, in: VLDB, 2007.
- [12] J. Cheng, J. Yu, On-line exact shortest distance query processing, in: EDBT, 2009.
- [13] E. Cohen, E. Halperin, H. Kaplan, U. Zwick, Reachability and distance queries via 2-hop labels, *SIAM J. Comput.* 32 (5) (2003).
- [14] G. Copeland, S. Khoshafian, A decomposition storage model, in: SIGMOD, 1985.
- [15] T. Cormen, R. Rivest, C. Leiserson, C. Stein, *Introduction to Algorithms*, third edition, The MIT Press, 2009.
- [16] I. Cruz, A.O. Mendelzon, P.T. Wood, A graphical query language supporting recursion, in: SIGMOD Conference, 1987.
- [17] R. Cyganiak, A relational algebra for SPARQL, Technical Report HPL-2005-170, HP Laboratories Bristol, 2005.
- [18] B. Elliott, E. Cheng, C. Thomas-Ogbuji, Z. Meral Özsoyoglu, A complete translation from SPARQL into efficient SQL, in: IDEAS, 2009.
- [19] P. Gassner, G. Lohman, K. Schiefer, Y. Wang, Query optimization in the IBM DB2 family, *IEEE Data Eng. Bull.* 16 (4) (1993).
- [20] G. Gou, R. Chirkova, Efficiently querying large XML data repositories: a survey, *TKDE* 19 (10) (2007).
- [21] G. Graefe, Sorting and indexing with partitioned B-trees, in: CIDR, 2003.
- [22] T. Grust, M. Mayr, J. Rittinger, S. Sakr, J. Teubner, A SQL: 1999 code generator for the pathfinder XQuery compiler, in: SIGMOD, 2007.
- [23] T. Grust, M. Mayr, J. Rittinger, T. Schreiber, FERRY: database-supported program execution, in: SIGMOD, 2009.
- [24] T. Grust, S. Sakr, J. Teubner, XQuery on SQL hosts, in: VLDB, 2004.
- [25] R.H. Güting, GraphDB: modeling and querying graphs in databases, in: VLDB, 1994.
- [26] M. Gyssens, J. Paredaens, D.V. Gucht, A graph-oriented object database model, in: PODS, 1990.
- [27] H. He, A. Singh, Graphs-at-a-time: query language and access methods for graph databases, in: SIGMOD, 2008.

- [28] J. Huan, W. Wang, D. Bandyopadhyay, J. Snoeyink, J. Prins, A. Tropsha, Mining protein family specific residue packing patterns from protein structure graphs, in: RECOM, 2004.
- [29] H. Jagadish, L.V.S. Lakshmanan, D. Srivastava, K. Thompson, TAX: a tree algebra for XML, in: DBPL, 2001.
- [30] R. Jin, Y. Xiang, N. Ruan, D. Fuhry, 3-HOP: a high-compression indexing scheme for reachability query, in: SIGMOD, 2009.
- [31] S. Klinger, J. Austin, Chemical similarity searching using a neural graph matcher, in: ESANN, 2005.
- [32] Juchang Lee, Yong Sik Kwon, Franz Färber, Michael Muehle, Chulwon Lee, Christian Bensberg, Joo-Yeon Lee, Arthur H. Lee, Wolfgang Lehner, SAP HANA distributed in-memory database system: transaction, session, and metadata management, in: ICDE, 2013, pp. 1165–1173.
- [33] U. Leser, A query language for biological networks, *Bioinformatics* 21 (2) (2005).
- [34] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: SIGMOD, 2010.
- [35] F. Manola, E. Miller, RDF primer, February 2004. (<http://www.w3.org/TR/2004/REC-rdf-primer-20040210/>).
- [36] T. Neumann, G. Moerkotte, Characteristic sets: accurate cardinality estimation for RDF queries with multiple joins, in: ICDE, 2011.
- [37] T. Neumann, G. Weikum, RDF-3X: a RISC-style engine for RDF, *PVLDB* 1 (1) (2008) 34.
- [38] P. Peng, L. Zou, L. Chen, X. Lin, D. Zhao, Subgraph search over massive disk resident graphs, in: SSDBM, 2011.
- [39] J. Pérez, M. Arenas, C. Gutierrez, Semantics and complexity of SPARQL, *TODS* 34 (3) (2009).
- [40] H. Pirahesh, T. Cliff Leung, W. Hasan, A rule engine for query transformation in starburst and IBM DB2 C/S DBMS, in: ICDE, 1997.
- [41] S. Powers, Practical RDF, O'Reilly Media, 2003.
- [42] E. Prud'hommeaux, A. Seaborne, SPARQL query language for RDF, W3C recommendation, (<http://www.w3.org/TR/rdf-sparql-query/>), January 2008.
- [43] Michael Rudolf, Marcus Paradies, Christof Bornhövd, Wolfgang Lehner, The Graph Story of the SAP HANA Database, BTW, 2013, pp. 403–420, (<http://www.btw-2013.de/proceedings/The%20Graph%20Story%20of%20the%20SAP%20HANA%20Database.pdf>).
- [44] S. Sakr, GraphREL: a decomposition-based and selectivity-aware relational framework for processing sub-graph queries, in: DASFAA, 2009.
- [45] S. Sakr, G. Al-Naymat, Relational processing of RDF queries: a survey, *SIGMOD Rec.* 38 (4) (2009).
- [46] S. Sakr, A. Awad, A framework for querying graph-based business process models, in: WWW, 2010.
- [47] S. Sakr, S. Elnikety, Y. He, G-SPARQL: a hybrid engine for querying large attributed graphs, in: CIKM, 2012.
- [48] M. Sarwat, S. Elnikety, Y. He, G. Kliot, Horton: online query execution engine for large distributed graphs, in: ICDE, 2011.
- [49] L. Sheng, Z.M. Özsoyoglu, G. Özsoyoglu, A graph query language and its query processing, in: ICDE, 1999.
- [50] L. Sidirorgos, R. Goncalves, M. Kersten, N. Nes, S. Manegold, Column-store support for RDF data management: not all swans are white, *PVLDB* 1 (2) (2008).
- [51] Vishal Sikka, Franz Färber, Anil K. Goel, Wolfgang Lehner, SAP HANA: the evolution from a modern main-memory data platform to an enterprise application platform, *PVLDB* 6 (11) (2013) 1184–1185.
- [52] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, S. Zdonik, C-Store: a column-oriented DBMS, in: VLDB, 2005.
- [53] F. Suchanek, G. Kasneci, G. Weikum, Yago: a core of semantic knowledge, in: WWW, 2007.
- [54] H. Tong, C. Faloutsos, B. Gallagher, T. Eliassi-Rad, Fast best-effort pattern matching in large attributed graphs, in: KDD, 2007.
- [55] S. Trißl, U. Leser, Fast and practical indexing and querying of very large graphs, in: SIGMOD, 2007.
- [56] C. Wang, J. Han, Y. Jia, J. Tang, D. Zhang, Y. Yu, J. Guo, Mining advisor–advisee relationships from research publication networks, in: KDD, 2010.
- [57] F. Wei, TEDI: efficient shortest path query answering on graphs, in: SIGMOD, 2010.
- [58] X. Yan, P. Yu, J. Han, Graph indexing: a frequent structure-based approach, in: SIGMOD, 2004.
- [59] S. Zhang, M. Hu, J. Yang, TreePi: a novel graph indexing method, in: ICDE, 2007.
- [60] S. Zhang, J. Li, H. Gao, Z. Zou, A novel approach for efficient supergraph query processing on graph databases, in: EDBT, 2009.
- [61] S. Zhang, S. Li, J. Yang, GADDI: distance index based subgraph matching in biological networks, in: EDBT, 2009.
- [62] P. Zhao, J. Han, On graph query optimization in large networks, *PVLDB* 3 (1) (2010).
- [63] L. Zou, L. Chen, M. Özsu, D. Zhao, Answering pattern match queries in large graph databases via graph embedding, *VLDB J.* 32 (5) (2011).
- [64] L. Zou, J. Mo, L. Chen, M. Özsu, D. Zhao, gStore: answering SPARQL queries via subgraph matching, *PVLDB* 4 (8) (2011).