An SQL-Based Query Language and Engine for Graph Pattern Matching

Christian Krause[1(✉)], Daniel Johannsen[1], Radwan Deeb[1], Kai-Uwe Sattler[2], David Knacker[1], and Anton Niadzelka[1]

[1] SAP SE, Potsdam, Germany
{christian.krause01,daniel.johannsen,radwan.deeb,
david.knacker,anton.niadzelka}@sap.com
[2] Technische Universität Ilmenau, Ilmenau, Germany
kus@tu-ilmenau.de

Abstract. The interest for graph databases has increased in the recent years. Several variants of graph query languages exist – from low-level programming interfaces to high-level, declarative languages. In this paper, we describe a novel SQL-based language for modeling high-level graph queries. Our approach is based on graph pattern matching concepts, specifically nested graph conditions with distance constraints, as well as graph algorithms for calculating nested projections, shortest paths and connected components. Extending SQL with graph concepts enables the reuse of syntax elements for arithmetic expressions, aggregates, sorting and limits, and the combination of graph and relational queries. We evaluate the language concepts and our experimental SAP HANA Graph Scale-Out Extension (GSE) prototype (This paper is not official SAP communication material. It discusses a research-only prototype, not an existing or future SAP product. Any business decisions made concerning SAP products should be based on official SAP communication material.) using the LDBC Social Network Benchmark. In this work we consider only complex read-only queries, but the presented language paves the way for a SQL-based graph manipulation language formally based on graph transformations.

1 Introduction

In contrast to relational database management systems, graph databases employ dedicated data structures and algorithms tailored for analytical and transactional graph processing. Current applications in the domains of social network analysis (e.g., Facebook, LinkedIn), business network analysis (e.g., Ebay, SAP Ariba) and knowledge graphs (e.g., Google Search, Microsoft Office) show that there is a high demand for efficient reasoning on large-scale graph data. There exist a number of low-level graph programming models, the most prominent being Bulk Synchronous Parallel [16]. However, in the context of enterprise applications there is a need for high-level, declarative graph query languages that enable complex analysis scenarios. While SQL is the accepted standard query

R. Echahed and M. Minas (Eds.): ICGT 2016, LNCS 9761, pp. 153–169, 2016.
DOI: 10.1007/978-3-319-40530-8_10

language in the world of relational databases, there currently is no consensus on a standard for a general-purpose graph query language.

OpenCypher [10] is an initiative by the inventors of the Neo4j graph database to define a common graph query language based on their Cypher language. For historical reasons, many companies today use Cypher. However, the language in its current form is ad hoc and lacks tool support by other vendors. SPARQL [18] is the standard query language in the semantic web domain. While it supports graph query concepts, it is geared into the triple store (subject-predicate-object) concept of the Resource Description Framework (RDF). General graph analysis applications may be encoded in SPARQL/RDF, but due the dedicated focus on the semantic web domain, there is a limit for applications with a different scope.

SQL is widely accepted as the standard query language for relational database systems. While extensions for hierarchical [2], geospatial [15] and time series data exist, graph queries have not been considered in the past. Reasons may be the complexity of graph queries (graph pattern matching, path expressions etc.) and too much focus on methods for encoding graph data in relational database tables. Moreover, specifying graph queries directly in SQL is cumbersome and often leads to inefficient query executions. Particularly graph pattern matching and transitive closures usually require dedicated graph query languages and engines.

In this paper, we propose a novel, high-level graph query language that is based on SQL. We build on the syntax and semantics of SQL, transfer its query concepts into the realm of graph databases, and extend them with dedicated graph features. In particular, our language supports graph pattern matching with nested graph conditions [3,7], expressions for traversing paths of fixed length, calculation of transitive closures and definition of distance constraints between matched nodes. The pattern matching is in general non-injective, but it can be customized by adding injectivity constraints for pairs of node variables. Moreover, dedicated functions for computing shortest paths and connected components are included. The general structure of queries follows the one of SQL. The syntax for arithmetic expressions, aggregates, sorting, limits etc. can be reused entirely. Since the result of graph queries are tables, they can be embedded as subqueries in standard relational SQL queries, thereby enabling a smooth integration with relational and other types of engines in heterogeneous database management systems. For instance, when agreeing on SQL as common base language, graph queries could be combined with relational, geospatial or even time series queries. Although the engine implementations are typically separate, a common base language enables the usage of a common query processing infrastructure including parsing, plan generation and query optimization.

We provide an execution engine for the proposed language, referred to as the SAP HANA Graph Scale-Out Extension (GSE) prototype in the rest of this paper. To achieve a high query performance, our engine uses optimized graph data structures instead of relational tables. We evaluate the expressive power of our query language and the performance of the GSE implementation using the LDBC Social Network Benchmark [4]. We focus in this paper on complex, read-only queries. However, our query language lays the foundation for an SQL-based

graph manipulation language based on the theory of graph transformations. Therefore, it paves the way for transferring formal methods, such as critical pair analysis for confluence checking [8], into the graph database realm.

Organization: Sect. 2 first gives an overview of the graph model we use. It then discusses graph pattern matching with nested formulas and subsequent relational evaluation. Section 3 introduces our SQL-based graph query language. Section 4 provides an evaluation of the GSE implementation based on an LDBC benchmark. Section 5 gives an overview of related work. Section 6 contains conclusions and future work.

2 Graph Pattern Matching with Relational Evaluation

In this section, we present the models and concepts that form the foundation of our query language and engine.

2.1 Graph Model

We consider directed and undirected graphs with typed nodes, typed edges and typed node properties. Each of these types has a fixed value range which is part of the graph definition. Figure 1 shows the corresponding graph model.

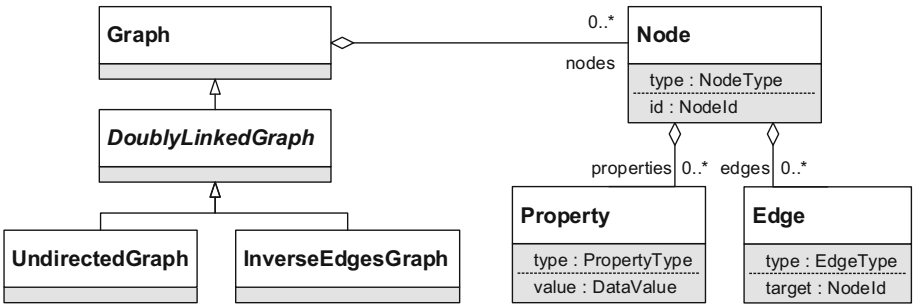


Fig. 1. Typed graph model with node properties and inverse edges

Every node has a unique numeric ID, a list of properties, and a list of edges. A property has a primitive data value of a data type derived from the property type (e.g., string, integer, float). The value of a property can be NULL independently of its data type. We consider the ID and the type of a node as special properties with the respective property types NODEID and NODETYPE.

An edge identifies its target node by its ID. Every node can have at most one edge of the same type and with the same target, i.e., parallel edges of the same type are not allowed. However, edges may target their source node (*loops*).

In a *doubly linked* graph, every edge has a corresponding edge with swapped source and target nodes. This corresponding edge either has the same type as the original edge or an implicitly defined inverse edge type. In the first case, the graph is *undirected*. In the second case, it is an *inverse edge* graph where all edges of regular type are outgoing and all edges of inverse type are incoming.

Our graph model deliberately omits edge properties (e.g., edge weights). This design choice enables an efficient implementation, particularly in a distributed setting. The resulting restriction can be overcome by modeling edges with properties by auxiliary edge nodes with one incoming and one outgoing edge.

2.2 Graph Pattern Matching

In graph pattern matching, the task is to find all matches between a set of pattern variables and the nodes of a target graph that satisfy a set of conditions. Figure 2 shows our graph pattern model and Fig. 3 a basic example pattern.

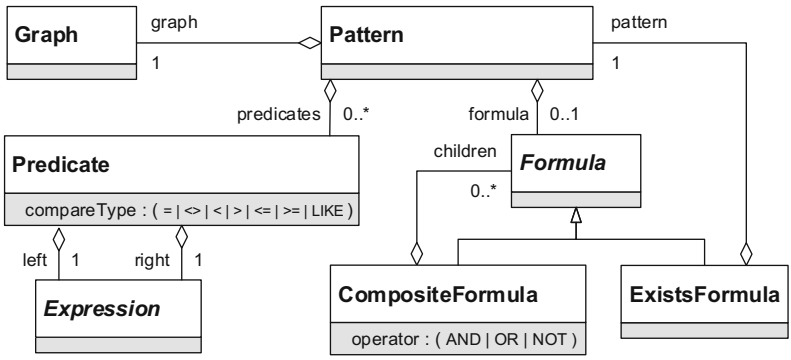


Fig. 2. Pattern model with predicates and nested graph constraints

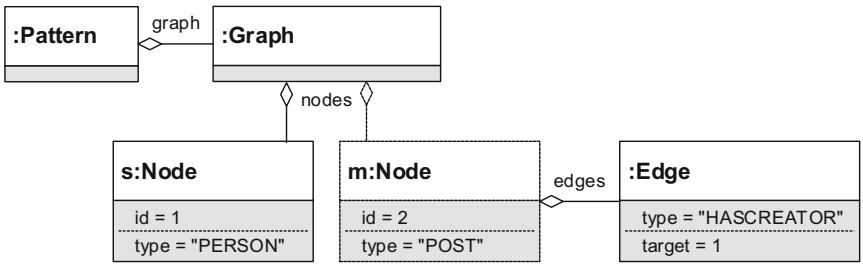


Fig. 3. Basic graph pattern consisting only of a pattern graph with two typed nodes and a typed edge between these nodes

A *pattern* consists of a pattern graph, a set of predicates, and an optional nested formula. The predicates and the formula are defined over a set of *pattern variables* that represent the *pattern nodes*, i.e., the nodes of the pattern graph.

A *match* of a pattern with respect to a given target graph is a map of the pattern variables to nodes of the target graph (called the *target nodes* of the match), such that (i) each pattern node is matched to a target node of the same type, (ii) each pattern edge is matched to a target edge of the same type, (iii) all predicates are satisfied, and (iv) the logical formula is satisfied.

Formally, conditions (i) and (ii) describe a typed graph homomorphism from the pattern graph into the target graph. In general, this graph homomorphism does not have to be injective. Instead, predicates comparing the IDs of the pattern nodes can be used to guarantee that pattern nodes are matched to different target nodes. The same approach is used to assure that a certain set of target nodes is matched only once if the pattern graph has symmetries.

In order to be able to omit the type constraint from conditions (i) and (ii), we add the implicitly defined type ANY to the lists of node types and edge types of the pattern graph. Thus, if a pattern node or edge has the type ANY, it can be matched to any target node or edge, respectively.

A predicate is a binary comparison between two expressions. Figure 4 depicts how we model expressions. Expressions are defined over the pattern variables and, given a potential match, evaluate to primitive data values. Literals simply evaluate to their constant value. Arithmetic expressions and functions are evaluated recursively, i.e., after evaluating their arguments they are evaluated as expressions over primitive data values.

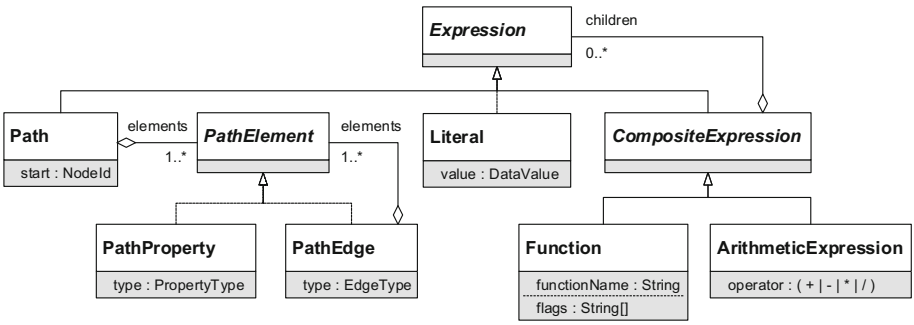


Fig. 4. Expression model

Path expressions form the link between the pattern variables and the properties of the target nodes. In their most basic form, paths consist of a pattern variable (given by the start node ID) and a property type (given by a path property). Such a path is called a *node property path* and evaluates to the property value with the given type of the target node matched to the given variable. All path expressions that occur in a graph pattern have to be node property paths.

This does not restrict the expressive power of the model, since complex path conditions can be expressed in the graph part of the pattern and by nested formulas. We discuss the evaluation of more complex paths in the next section where expressions containing such paths are introduced.

Note that we may access the ID of a target node and use it in other expressions, most notably functions. Such functions are called *graph functions* and are parametrized by optional function flags (see Table 1). They have access to the complete target graph, allowing them to explore the neighborhood of a node (e.g., to calculate degrees) as well as to traverse the global graph structure (e.g., to compute distances, shortest paths, and connected components).

Table 1. Summary of currently supported graph functions

Function	Args	Flag	Result
DEGREE	1		Degree of argument node
		IN	Compute in-degree
		OUT	Compute out-degree (default)
		INOUT	Compute in-degree + out-degree
DISTANCE	2		Node distance (NULL if unconnected)
SHORTEST_PATH	2		Shortest paths sub-graph (as JSON)
		DIRECTED	Traverse edges regularly (default)
		UNDIRECTED	Traverse edges in both directions
		INVERSE	Traverse edges in inverse direction
CONNECTED_COMPONENT	1		Id of node's connected component
		STRONG	Assume strong connectivity (default)
		WEAK	Assume weak connectivity
All graph functions:		EDGETYPE type	Edge type restriction (default: ANY)

Formulas are used to model nested graph conditions. They are either composite (a logical operator) or an existential quantification of a nested graph pattern. Note that this is a recursive tree structure which terminates at patterns. There is an implicit mapping of the nodes of a pattern graph into its child pattern graphs given by their node IDs, which for every match of the parent pattern graph induces a pre-match of the child pattern graph.

2.3 Graph-Relational Evaluation and Nested Paths

The result of the graph pattern matching described in the previous section is a list of matches from the pattern variables to nodes in the target graph. Based on a feature list, *graph-relational evaluation* computes a table of primitive data values from these matches. A *feature* is an expression as defined in the previous section. Thus, given a list of features and a list of matches, we can compute a table where each column corresponds to a feature. For each match, we create a row in the table by evaluating the feature expressions over the match.

In the following, we extend this concept by introducing expressions that do not evaluate to single values but rather to lists of data value tuples.

In the previous section, we introduced the concept of node property paths which evaluate to a single value. Now we extend this notion to nested projection paths which we specify in Fig. 4. Such a nested projection path describes a nested traversal of the target graph and evaluates to a list of property value tuples. For this, path elements are recursively evaluated. Unlike for composite expressions, this evaluation is carried out top down. This means that the parent path element is evaluated first and then passes a target node as an argument to its children.

For example, consider the nested projection path depicted in Fig. 5. It consists of an edge followed by at nested projection to a property and a subpath formed by a second edge and another property. If the pattern variable (start) is matched to a node representing a person, the path evaluation traverses all outgoing WORKSAT edges to find all companies the person worked for. For every company and each location of the company, the evaluation creates a pair of the company name and the location name. Each company generates at least one pair, even if it has no ISLOCATEDIN edge. Likewise, the whole traversal generates at least one pair, even if the person has no outgoing WORKSAT edge (the NULL pair).

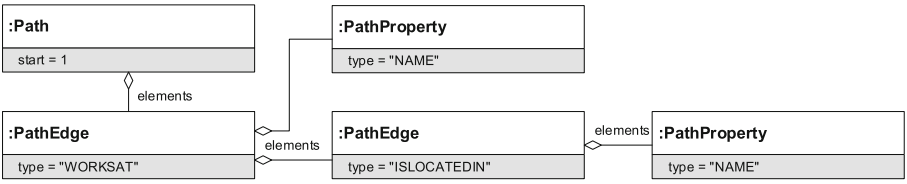


Fig. 5. Example of a nested projection path.

In the following, we discuss nested path expressions in detail, beginning with basic *traversal paths*. A traversal path expression consists of a pattern variable (start) and a sequence of path edges followed by a path property. When evaluated, the path expression first determines the target node matched to its variable and passes it as argument to the first path edge. Every subsequent path edge is evaluated by traversing all edges of its argument node that match its type. For each of these traversed edges, the end-node is passed as argument to the next path edge in the sequence.

The recursion terminates at a path property or if the argument target node of a path edge does not have any edges of the respective type. In the first case, the result is the respective property of the target node that was passed as argument. In the second case, the path expression returns a set containing a single NULL value. From a relational perspective, a traversal path expresses a left outer join over the edge relation. From a graph-theoretic perspective, it expresses a path traversal of the target graph starting at the start node of the path expression and using the edge types defined by the path edges. The result is the list of

property values of the traversal path end-nodes specified by the property type of the path property.

If one or more of the expressions forming a feature list contains a traversal path, the one-to-one correspondence between matches and rows of the result table becomes a one-to-many correspondence. When evaluating the feature list for a particular match, we first evaluate all traversal path expressions. Next, we create the cross product of the data value lists that result from the path traversals. Finally, we create a row in the result table for every tuple in the cross product. For this, we first replace all traversal paths in the feature expressions by the corresponding cross product tuple element. Then we evaluate this modified feature list for the current match.

In addition to traversal paths, we also define *nested projection paths*. A nested projection path expression and its path edges may have more than one child path elements. In this case, the result is a list of tuples which is derived by evaluating all child elements and creating the cross product of their result lists. Note that the result lists of the child elements may already be tuple lists. In this case we stay in line with relational algebra and treat the tuples as *shallow*, i.e., concatenate the tuples when creating the cross product (in contrast to cre-

Table 2. Notation used to specify the syntax of our query language

Construct	Notation	Comments
Grammar rule	`rule`	Grammar rules use lowercase letters and underscores
Definition	`=`	Definitions are represented by a single equal signs
Alternation	`...|...`	Alternatives are separated using vertical bars
Grouping	`(...)`	Grouping is represented by enclosing parentheses
Option	`[...]`	Optional parts are represented by enclosing square brackets
Repetition	`...*`	Zero or more repetitions are indicated by the suffix `*`
Terminal symbol	`KEYWORD`	Language keywords are written in uppercase letters
Terminal character	`"."`	Single-character language symbols are set in double quotes
Terminal literal	`literal`	Literals represent typed string and numerical constants
Terminal identifier	`identifier::id`	Identifiers are indicated by the suffix `::id`
List abbreviation	`rule::list`	Comma-separated lists are abbreviated by the suffix `::list.list_rule = rule (","rule )*`

ating tuples of tuples). From a relational perspective, a nested path expresses a cross join over its child elements. From a graph-theoretic perspective, it expresses a tree traversal of the target graph.

During evaluation, nested projection paths are treated like traversal paths, i.e., the result lists of the nested projection paths become part of the cross product created over the result lists of the traversal paths. However, a feature represented by a nested projection path expression corresponds to several table columns, one for every element in the result tuples of the nested projection path. A nested projection paths may appear as a subexpression of a compound expression (e.g., a function). The compound expression is evaluated for each tuple generated by the nested path expression.

3 Graph Query Language

The syntax and semantics of our graph query language is described in this section. It is closely aligned with the SQL standard. It uses, where possible, SQL syntax and extends it with graph-specific features. Some of these extensions can be also found in a similar form in the query language of SAP HANA Core Data Services (CDS) [14]. We discuss three complex example queries in Sect. 4.

To specify the syntax of our query language, we use the notation defined in Table 2 which is inspired by the Extended Backus-Naur Form (EBNF). For the sake of brevity, we consider identifiers and literals as additional syntax terminals beside the traditional syntax terminals.

Listing 1.1. Overview of graph query syntax

```
query      = SELECT ( "*"| feature::list ) FROM variable::list
             [ USING GRAPH graph::id ] [ WHERE condition ]
             [ GROUP BY expression::list ] [ ORDER BY expression::list ]
             [ LIMIT literal ]

feature    = expression [ AS expression_alias::id ]
variable   = node_type::id [ [ AS ] variable_alias::id ]

condition  = formula | predicate | ( path IN path )

formula    = ( "("condition ( AND | OR ) condition ")")
             | ( "("NOT condition ")")
             | ( EXISTS variable::list WHERE "("condition ")")
predicate  = ( expression comparator expression ) | ( path IS [ NOT ] NULL )
comparator = "="| "<>"| "<"| ">"| "<="| ">="| LIKE

expression = literal | arithmetic | function | path
arithmetic = "("expression ( "+"| "-"| "*"| "/") expression ")"
             | "(""-"expression ")"
function   = function_name "("flag::id* expression::list ")"

path       = variable::id [ projection ]
projection = "."element | ( "."{"element::list "}" )
element    = property::id | ( edge_type::id [ projection ] )
```

3.1 Query Structure

A graph query is a read-only operation that performs graph pattern matching on a given target graph followed by relational evaluation to generate a primitive-typed result table. This table can be subsequently consumed by other relational

operators. Therefore, graph queries can in principle be embedded in and combined with standard SQL queries. Syntactically, graph queries follow closely the structure of SELECT statements in SQL. Listing 1.1 summarizes the syntax definition of our graph query language.

The query language syntax provides a clear separation between the graph pattern matching and the graph-relational evaluation. This is achieved by defining each operation in different clauses (with the only overlap of the variable definition in the FROM clause). The pattern matching is defined by the FROM, USING GRAPH, and WHERE clauses, whereas the relational evaluation is defined by the SELECT, FROM, GROUP BY, ORDER BY, and LIMIT clauses.

The following query finds all matches to the graph pattern shown in Fig. 3:

```
SELECT s.ID, m.CONTENT, s.WORKSAT.{NAME,ISLOCATEDIN.NAME} AS COMPANY
FROM PERSON s, POST m
WHERE s IN m.HASCREATOR
```

It then evaluates for each matched person and post the nested projection path depicted in Fig. 5. Each matched person and post together with a traversed company and location produce a row in the result table, containing the respective properties.

3.2 Pattern Matching

As discussed in Sect. 2.2, in graph pattern matching we compute all matches from the node variables of a graph pattern to the nodes of a target graph that satisfy the conditions of a given graph pattern.

The USING GRAPH clause of a graph query identifies the target graph by its name. Thus, multiple graphs stored in a graph database can be distinguished.

The variables in the FROM clause consist of a node type and a variable alias. Although syntactically similar to standard SQL, the semantics of these variables differs from relational queries. In a graph query, the variables represent the pattern variables introduced in Sect. 2.2. During the graph pattern matching, these variables are matched to nodes in the target graph rather than to relational tuples. Moreover, the node type of a variable declaration is already part of the graph pattern and defines the type of the corresponding pattern node.

The main part of the graph pattern is defined in the condition part of the WHERE clause. The syntax of formulas, predicates, expressions, and paths directly translates to the corresponding models discussed in Sect. 2.2.

Semantically, only paths differ from standard SQL, since they are defined over pattern variables and not over relational variables. Note that not all syntactically correct paths are admitted at every position in a graph query. Depending on whether the position allows for one result or a result list and for a single value or a tuple, traversals and nested projections may be forbidden. Moreover, the end-property of a path is optional and defaults to NODEID.

The main difference between standard SQL conditions and graph query conditions is the syntax and semantics of the IN keyword. In an IN condition, the path before the IN keyword must be a node property path. The path after the IN

keyword has to be a traversal path. Semantically, the IN conditions are mainly used to define the edges and the edge types of the pattern graph.

3.3 Relational Evaluation

The relational part of a graph query is defined by the SELECT, FROM, GROUP BY, ORDER BY, and LIMIT clauses. The optional GROUP BY, ORDER BY, and LIMIT clauses of a graph-relational evaluation follow the standard SQL syntax and semantics (see Listing 1.1), generating tables as result of graph queries.

The graph-specific part of the relational evaluation is defined in the SELECT and FROM clauses. In the previous section, we already discussed the FROM clause and established that semantically it defines the pattern variables. The feature list syntax directly reflects and maps to the concepts introduced in Sect. 2.3. There, we already discussed in detail the evaluation of feature lists, including the implicit cross-join introduced by traversal paths and nested paths.

We also allow the SELECT * syntax known from standard SQL. In a graph query, the star symbol is expanded to a feature list which contains a nested projection path `x.{NODEID, NODTYPE, P1,..., Pk}` for every variable `x` in the FROM clause, where `P1,..., Pk` are all property types for which at least one node of the variable's type has a non-NULL value.

Besides the graph functions defined in Table 1, function expressions can also be classical SQL aggregates such as COUNT or AVG and arithmetic functions such as ABS or MOD. Since the result of the graph-relational evaluation is a table, the semantics of aggregates (in particular in conjunction with a GROUP BY clause) carries over from standard relational queries.

4 Evaluation

In this section, we evaluate our graph query language and the GSE prototype engine implementation using the LDBC Social Network Benchmark.

4.1 Implementation

In the following we give an overview of the GSE prototype implementation of a graph query engine that supports major parts of our query language.

The high-level architecture of the GSE is shown in Technical Architecture Modeling notation in Fig. 6. Application development is supported via a high-level programming API (C/C++, Java) and a Neo4j-compatible REST-API. An additional connector provides an integration with SAP HANA Vora [5] – a scale-out extension of SAP HANA for massively parallel data processing integrating with the Hadoop framework.

The core of the graph engine uses a distributed in-memory graph store which implements the graph model shown in Fig. 1. Data adapters enable the loading and saving of graph data from (distributed) file systems such as HDFS, and relational tables in SAP HANA/SAP HANA Vora.

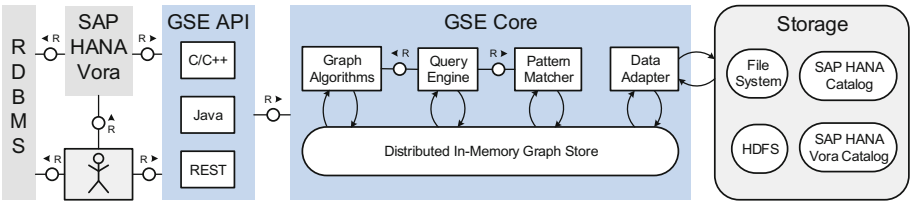


Fig. 6. High-level GSE architecture

The query engine parses textual queries as described in Sect. 3, translates them to pattern models (Fig. 2) and uses the pattern matcher and graph algorithm implementations to execute queries (see Table 1 for supported graph functions). The query execution and pattern matching are then parallelized and distributed across a cluster. The pattern matching engine is based on an encoding into a constraint satisfaction problem [13].

4.2 LDBC Social Network Benchmark

The Linked Data Benchmark Council (LDBC) is a non-profit organization defining benchmarks for graph data management software. We use here the Social Network Benchmark (SNB), specifically the Interactive Workload [4], consisting of 29 queries which are split into three categories: complex read, short read and update queries. This benchmark includes different "choke points" for query engines, such as aggregation performance and data access locality. Graphs to run this workload against can be generated with the help of a given data generator, which produces a social network graphs of a given scale factor.

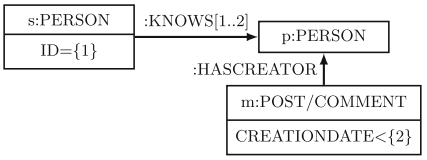
4.3 Complex Read Queries

There are 14 complex read queries in the LDBC Social Network Benchmark. Out of these, 13 can be represented in our graph query language. Figure 7 shows three selected queries including informal descriptions and graphical representations of their respective pattern graphs.

In Query 9, the implicitly defined node type ANY (line 4) is used in combination with NODETYPE (lines 7–8) enabling a simple kind of node type inheritance. The predicate involving the DISTANCE-function (line 9) ensures that the two PERSON nodes are connected by a KNOWS-path of length at most 2. Note that the :KNOWS[1..2]-path in the graphical notation refers to the length of a shortest path between the two matched nodes. The predicate s<>p is used to selectively ensure injective matching of the two PERSON nodes.

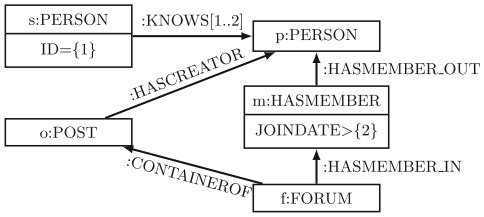
Query 5 consists of a cyclic pattern graph. This query is also an example of a conversion of edges with attributes to nodes and node properties (see Sect. 2.1). The original edge HASMEMBER from PERSON to FORUM is modeled as a node of type HASMEMBER, which contains all properties of the original

```
SELECT p.ID, p.FIRSTNAME, p.LASTNAME,
  m.ID, m.CONTENT, m.IMAGEFILE,
  m.CREATIONDATE
FROM PERSON s, PERSON p, ANY m
WHERE s.ID={1}
AND s<>p
AND (m.NODETYPE='POST' OR
     m.NODETYPE='COMMENT')
AND DISTANCE(EDGETYPE KNOWS s,p)<=2
AND p IN m.HASCREATOR
AND m.CREATIONDATE<{2}
ORDER BY m.CREATIONDATE DESC, m.ID ASC
LIMIT 20
```



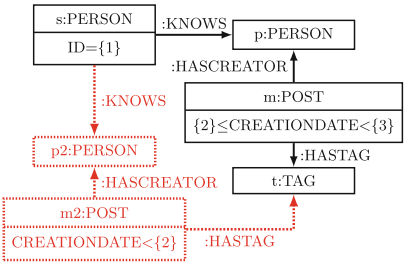
Query 9: Recent posts and comments by friends or friends of friends: Given a start Person, find the (most recent) Posts/Comments created by that Persons friends or friends of friends (excluding start Person). Only consider the Posts/Comments created before a given date (excluding that date). Return the top 20 Posts/Comments, and the Person that created each of those Posts/Comments. Sort results descending by creation date of Post/Comment, and then ascending by Post/Comment identifier.

```
SELECT f.TITLE, COUNT(o)
FROM PERSON s, PERSON p, POST o,
     HASMEMBER m, FORUM f
WHERE s.ID={1}
AND s<>p
AND DISTANCE(EDGETYPE KNOWS s,p)
                              <= 2
AND m IN f.HASMEMBER_IN
AND p IN m.HASMEMBER_OUT
AND m.JOINDATE>{2}
AND o IN f.CONTAINEROF
AND p IN o.HASCREATOR
GROUP BY f.ID
ORDER BY COUNT(o) DESC, f.ID ASC
LIMIT 20
```



Query 5: New Groups: Given a start Person, find the Forums which that Persons friends and friends of friends (excluding start Person) became Members of after a given date. Return top 20 Forums, and the number of Posts in each Forum that was Created by any of these Persons. For each Forum consider only those Persons which joined that particular Forum after the given date. Sort results descending by the count of Posts, and then ascending by Forum identifier

```
SELECT t.NAME AS TAGNAME, COUNT(t) AS tc
FROM PERSON s, PERSON p, TAG t, POST m
WHERE s.ID={1}
AND s IN p.KNOWS
AND t IN m.HASTAG
AND p IN m.HASCREATOR
AND m.CREATIONDATE>={2}
AND m.CREATIONDATE<{3}
AND NOT EXISTS PERSON p2, POST m2
WHERE (p2 IN s.KNOWS
    AND p2 IN m2.HASCREATOR
    AND m2.CREATIONDATE<{2}
    AND t IN m2.HASTAG
    AND m<>m2
)
GROUP BY t.NAME
ORDER BY COUNT(t) DESC, t.NAME ASC
LIMIT 10
```



Query 4: New Topics: Given a start Person, find Tags that are attached to Posts that were created by that Persons friends. Only include Tags that were attached to friends Posts created within a given time interval, and that were never attached to friends Posts created before this interval. Return top 10 Tags, and the count of Posts, which were created within the given time interval, that this Tag was attached to. Sort results descending by Post count, and then ascending by Tag name.

Fig. 7. Selected queries of the LDBC Social Network Benchmark

edge. Two auxiliary edges of types HASMEMBER_IN and HASMEMBER_OUT connect the new node with the source and target nodes. Note that our Query 5 slightly differs from the original SNB query. Specifically, our query is missing an optional match part which is currently not implemented in GSE.

Query 4 exhibits the usage of subpatterns with the help of the EXISTS-keyword (lines 9–15). In particular, it is combined with NOT to specify a negated graph condition. Note that graph conditions can in general be nested and combined using Boolean operators.

4.4 Performance

For performance evaluation, we use generated LDBC data of scale-factor 3, which corresponds to a graph with 23.7M nodes and 84.8M edges. Minor adaptations of the generated data include a conversion of string-based dates into integer-encoded timestamps. We generate property indices for the property types ID, NAME and FIRSTNAME. The benchmarks were executed on a 24-core Intel(R) Xeon(R) X5650 workstation at 2.67 GHz and 96 GB main memory. The total memory consumption of the LDBC graph at scale-factor 3 is 12.5 GB.

Our benchmark is implemented as a Java application using the Java APIs of GSE and Neo4j. We use the Community Edition of Neo4j in version 2.3.0. The Neo4j benchmark queries are taken from [11]. Both systems were benchmarked independently from each other on the same machine, using the same query parameters generated by the SNB data generation tool. The queries were executed between 20 and 50 times, depending on query complexity. To warm up both engines, all queries were executed with randomly chosen parameters beforehand.

Out of the 14 SNB queries, 8 queries were implemented completely. For the queries 1, 3 and 5, the GSE implementation is currently missing some functionality, specifically the addition of aggregates, nested projections, and optional pattern nodes (which, if not regularly matched, still generate a match to an implicitly defined null-vertex without any attributes). Therefore, we used slightly altered versions of those queries for our benchmark. Theses changes were also reflected in the respective Cypher queries to ensure comparability of the results. For Query 7, GSE currently lacks support for the SQL CASE-keyword, for Query 10, a DISTINCT flag for COUNT and modulo-operation. Notice, that with the exception of optional nodes, all missing implementation belongs to the relational part of the query and does not represent limitations of the pattern model or the query language. Optional nodes cannot be currently expressed in the query language (but might be by implicitly adding a special null-vertex to every host graph). Query 14 is the only query for which a major extension of our query language is required, since it includes shortest paths weighted by sub-patterns.

Table 3 shows the query run-times of Neo4j and GSE in milliseconds and the respective speed-up factors. GSE is faster for all implemented queries, staying under the one-second mark for most of them. The long running queries, particularly Query 9, yield large match sets and result tables which need to be sorted.

Our analysis shows that the pattern matching is efficient in these queries, but the generation of the result table and its sorting requires most of the time.

Table 3. LDBC complex read benchmark: mean of runtimes in milliseconds and speed-up factors. Queries marked with * include minor modifications.

Query	1*	2	3*	4	5*	6	8	9	11	12	13
Neo4j	7,041	2,122	5,496	12,262	10,074	44,625	161	405,457	197	5186	5
GSE	40	195	83	105	2,468	1,325	30	13,616	12	42	2
Speed-up	174	10	66	116	4	33	5	29	16	123	2

5 Related Work

SPARQL [18] supports graph pattern matching using an RDF-triple syntax, e.g.

```
PREFIX LDBC: <http://ldbcouncil.org/developer/snb>
SELECT ?m ?n WHERE { ?x LDBC:NAME ?m . ?x LDBC:KNOWS ?y . ?y LDBC:NAME ?n }
```

Note that both edges and primitive-valued properties are described using triples. Conditions on properties are defined using FILTER, existential quantification using EXISTS, and alternative patterns using the UNION keyword. Furthermore, optional patterns parts can be specified using the OPTIONAL keyword.

The openCypher [10] query language uses a different query structure, e.g.

```
MATCH (x:PERSON)-[:KNOWS]-(y:PERSON) RETURN x.NAME, y.NAME
```

Alternative pattern parts are defined using UNION and optional patterns using OPTIONAL MATCH. There is no existential quantification of patterns but negation and collections can be used to define more complex graph conditions. Both the SPARQL and openCypher syntax make use of SQL-keywords. However, the general query structure is too specific to be integrated with SQL. In contrast, our query language is closely aligned to the SQL standard and therefore enables a smooth integration with relational database systems.

The Gremlin [12] language which is part of the Apache TinkerPop project can be characterized as a functional embedded DSL for graph traversals.

A recent performance comparison of graph and relational databases is given in [6]. The paper shows that state-of-the-art relational databases can compete or even exceed the performance of graph databases in certain graph pattern matching scenarios. However, the employed benchmark uses rather simple graph patterns without additional (nested) graph conditions.

An encoding of graph transformation rules in SQL is presented in [17]. A graphical syntax is used for modeling the graph transformation rules. In general, except for the limitation of being read-only, our query language provides a

number of features that are usually not found in graph transformations, such as distance constraints, shortest paths, and relational operations including sorting, limits and aggregations. Note also that graph transformations usually operate on *one* match, whereas our approach always considers *all* matches.

6 Conclusions and Future Work

We presented a novel SQL-based graph query language supporting graph pattern matching with nested graph conditions [7] and distance constraints, as well as calculation of nested projections, shortest paths and connected components. Since it is based on SQL, the syntax for arithmetic expressions, aggregations, sorting etc. can be reused entirely, and graph queries can be embedded as subqueries in relational queries. We evaluated the language features and the GSE prototype implementation using the LDBC Social Network Benchmark.

As future work, we plan to incorporate optional matching (see [10,18]) and concepts for shortest paths weighted by sub-patterns as required in LDBC-SNB Query 14. We further plan to define a SQL-based graph manipulation language formally based on amalgamated graph transformations [1]. The planned graph manipulation language will build on the pattern matching syntax proposed in this paper and extend it to provide the same expressive power as graph transformation rules. The syntax of FROM and WHERE clauses will be reused and extended by an UPDATE clause for specifying created and deleted graph parts, and updates of property values. Semantically, the relational evaluation will be replaced by an amalgamated graph transformation step. The use of graph transformations is potentially enabling formal analysis techniques, such as critical pair analysis for confluence checking [8]. For the implementation, we plan to add nested projections and optional matching to the GSE query engine, to incorporate Bulk Synchronous Parallel [9,16] for distributed graph algorithms, and to compare the performance of our (distributed) engine with other graph database engines such as Sparksee and Virtuoso.

References

1. Boehm, P., Fonio, H., Habel, A.: Amalgamation of graph transformations: a synchronization mechanism. J. Comput. Syst. Sci. **34**(2/3), 377–408 (1987)
2. Brunel, R., Finis, J., Franz, G., May, N., Kemper, A., Neumann, T., Färber, F.: Supporting hierarchical data in SAP HANA. In: Proceedings of ICDE 2015, pp. 1280–1291. IEEE (2015)
3. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.-H.: Constraints and application conditions: from graphs to high-level structures. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 287–303. Springer, Heidelberg (2004)
4. Erling, O., Averbuch, A., Larriba-Pey, J., Chafi, H., Gubichev, A., Prat-Pérez, A., Pham, M., Boncz, P.A.: The LDBC social network benchmark: interactive workload. In: Proceedings of 2015 ACM SIGMOD, pp. 619–630. ACM (2015)

5. Goel, A.K., Pound, J., Auch, N., Bumbulis, P., MacLean, S., Färber, F., Gropengießer, F., Mathis, C., Bodner, T., Lehner, W.: Towards scalable real-time analytics: an architecture for scale-out of OLxP workloads. PVLDB **8**(12), 1716–1727 (2015)
6. Gubichev, A., Then, M.: Graph pattern matching - do we have to reinvent the wheel? In: Proceedings of GRADES 2014, pp. 8:1–8:7. ACM (2014)
7. Habel, A., Pennemann, K.: Correctness of high-level transformation systems relative to nested conditions. Math. Struct. Comput. Sci. **19**(2), 245–296 (2009)
8. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of typed attributed graph transformation systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 161–176. Springer, Heidelberg (2002)
9. Krause, C., Tichy, M., Giese, H.: Implementing graph transformations in the bulk synchronous parallel model. In: Gnesi, S., Rensink, A. (eds.) FASE 2014 (ETAPS). LNCS, vol. 8411, pp. 325–339. Springer, Heidelberg (2014)
10. Neo Technology Inc., OpenCypher (2015). http://www.opencypher.org
11. Prat, A., Boncz, P., Larriba, J.L., Angles, R., Averbuch, A., Erling, O., Gubichev, A., Spasić, M., Pham, M.D., Martínez, N.: LDBC Social Network Benchmark (SNB) - v0.2.2 first public draft (2015). http://github.com/ldbc/ldbc_snb_docs/blob/master/LDBC_SNB_v0.2.2.pdf
12. Rodriguez, M.A.: The Gremlin graph traversal machine and language (invited talk). In: Proceedings of DBPL 2015 (DBPL 2015), pp. 1–10. ACM (2015)
13. Rudolf, M.: Utilizing constraint satisfaction techniques for efficient graph pattern matching. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 238–252. Springer, Heidelberg (2000)
14. SAP SE: SAP HANA core data services (CDS) reference (2015). http://help.sap.com/hana/SAP_HANA_Core_Data_Services_CDS_Reference_en.pdf
15. SAP SE: SAP HANA spatial reference (2015). http://help.sap.com/hana/sap_hana_spatial_reference_en.pdf
16. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
17. Varró, G., Friedl, K., Varró, D.: Implementing a graph transformation engine in relational databases. Softw. Syst. Model. **5**(3), 313–341 (2006)
18. World Wide Web Consortium (W3C): SPARQL 1.1 query language (2013). http://www.w3.org/TR/2013/REC-sparql11-query-20130321