On using object-relational technology for querying LOD repositories

Iztok Savnik
Department of Computer Science
Faculty of Mathematics, Natural sciences and Information Technology
University of Primorska, Koper, Slovenia
iztok.savnik@upr.si

***Abstract*—Query engines for managing RDF repositories based on relational technology represent an alternative to query engines based on triple-stores. The paper presents adaptation of object-relational technology for managing RDF data. The architecture of query engine for RDF databases is proposed: i) mapping of RDF graph model to object-relational representation is described, and ii) internal structures and methods used for the implementation of query engine are discussed. While following the architecture of object-relational systems we adapt it for the specific operations defined on RDF data.**

***Keywords*-databases; RDF databases; query processing system; database system implementation.**

I. Introduction

Linked Open Data (abbr. LOD) repositories storing RDF files represent primary means to publish data on the Internet. They serve recently to disseminate large amounts of data in the areas of Life-science, Biology, Chemistry, Medicine, Geography and Media as well as about the institutions such as Governments. It is estimated that the data sets on the above stated areas include more than 31 Giga triples [15].

The first and most common approach to storage and manipulation of RDF data is the use of triple-storage system which is based on few relational tables and a collection of indexes built on these tables. Examples of such triple-store based systems are 3store [13] and Bigdata [3]. SPARQL queries are converted into some form of relational calculus expressions which are further converted to access paths implemented by indexes.

The second approach that is also common due to availability of relational and object-relational systems is the use of ordinary RDBMS by converting triples into relational or object-relational tables with three or four columns. SPARQL queries are converted into relational queries. Indexes can be created as needed and RDBMS uses its optimizer to create fast plans for SPARQL queries. Example of DBMS using such approach is Virtuoso [29].

Both of the above stated approaches to querying RDF repositories treats complete repository as one single database (or table of triples) on top of which indexes are created, either that we use standard object-relational indexes or special indexes that depend on a concrete system.

In this paper we present architecture of a database system for storing RDF triples as classes, properties and concrete objects. RDF triples are converted to specific kind of entities when they are loaded into the database. We follow therefore *conceptual* representation of RDF triples that uses RDFS types or least general types (classes) covering all properties of individual concrete triples, in the case RDFS types are not used. Class extensions now represent collections of concrete objects.

The problems that appear with querying collections of highly structured entities are in a way common to object-oriented database systems [2], object-relational systems [19], [28], XML databases [5] and RDF databases [20]. Among the most important problems that are characteristic for querying complex entities organized in collections are: accessing components of complex entities using path expressions [8], [17], restructuring and creating complex entities [28], [21], and querying complex conceptual schemata [23].

A practical solution to the problems of querying collections of complex objects is presented. Query processing is rooted in relational and object-relational technology. The existing techniques have been adapted and simplified to obtain database system tuned to process collections of complex objects structured by means of RDF data modeling constructs.

In the following section we present design decisions for architecture of query processing system for querying RDF databases. The architecture of query processing system is based on previous work on querying system for internet data sources [25], [23] and on formal definition of the structural part of object-oriented model [24].

Firstly, the storage manager and underlying data model is presented. Further, mapping between RDF and internal model of a database system is described. We overview the architecture of query processor including parsing, type checking, query optimization and query evaluation. Finally, bulk loader for RDF is described. Section III gives a short overview of related work and Section IV presents conclusions and further work.

II. Architecture of query processor

Our work focuses on the design of robust and flexible query execution system for querying and integration of RDF

 ISBN: 978-1-61208-247-9

data. The design of query execution system is rooted in the existing work on relational and object-relational query execution systems [11], [10], [9], [14], [6].

A. Storage manager

Every object has an identifier denoted as *object identifier* or *oid*. Objects can have functional or multi-valued attributes. The term "attribute" corresponds to what is also called object component or data member. The range of the functional attribute can be an atomic value: `number`, `string` or `object` (oid). Multi-valued attribute is an array of atomic values. The data model of is formally presented in [24].

From a programmer's perspective, an object is a reference to the associative (hash) array and an attribute is a hash entry which can be either a typed scalar or a reference to an array of typed scalars. Storage manager differentiates between two kinds of objects: class objects and individual objects (instances). Each class object is associated to the set of its sub-classes and its instances. The storage manager implementation is based on the hierarchical and network data models: class, its instances and sub-classes are linked in a ring realized by a double linked list.

1) Types: We distinguish between two sorts of types: *atomic types* and *tuple-structured types*. Atomic types are: `number`, `string` and `object`. Type `object` is oid of the root class object. Its instances are all oids. Tuple-structured types have attributes that are either functional or multi-valued (implemented as refs to arrays). The range of functional and multi-valued attributes can only be atomic types.

As suggested by the above description, types together with the corresponding identifiers form *class objects*. The structure of database can then be viewed uniformly as partially ordered set of objects: class objects and instances (individual objects) are interrelated by the inheritance (isa) and instantiation relationship.

2) Record manager: The record storage manager is based on the Berkeley DB storage system providing access to different storage structures such as for instance hash-based index or B+ tree. Record manager implements a data store for records representing individual and class objects. Records are treated as arrays of bytes, the structure of which is known at the object level. Each record has a record identifier (abbr. rid) implemented as system generated identifier which is used as the key for the hash-based index in Berkeley DB. Therefore, records are stored as oid/value pairs where the values are packed in the sequences of bytes. Record manager includes the methods for the work with object identifiers, routines for reading and writing records, methods for the realization of the hierarchical database model, and access to main memory indexes that are associated to the class object.

3) Object manager: The object manager serves as a cache of objects loaded from Web as well as for storing intermediate results during query processing. Persistent objects are objects that are tied to the database via object manager. Each object has an identifier which is implemented by means of record identifier from the subordinate level. External identifiers which are unique within the datafile can be assigned to objects when they are created. Object cache is realized using LRU (least recently used) strategy for the selection of objects to be removed from cache. The size of object cache can be set as the system parameter via the configuration record of a datafile as well as at run-time.

Object manager handles relationships between instances and class records (objects) in a similar manner as in early network and hierarchical database storage systems. Each class is realized as a list including all instances as well as sub-classes. Ground objects are added at the beginning of the list and the class objects are added at the end of the list.

The implementation of objects is based on associative arrays (or mappings) which map object identifier to the value of the specified attribute. The module includes simple and uniform routines for the manipulation of the instance objects and class objects. The routines can create and delete objects, set and retrieve (get) the values of object attributes, relate objects to inheritance hierarchy and update operations. Two different get and set methods are implemented for reading and updating single and multivalued attributes.

Objects can be accessed either directly using object identifiers, or through scan operator (iterator). When accessed using object identifiers, objects are manipulated through the main memory pointers to objects in cache. User is responsible that object is accessed each time before it is used. Routines for update operation mark the object "changed" when components are altered, unchanged objects do not need to be treated thereafter. When objects are accessed using scans they are obtained by means of iterator. The following scan operators are provided: sequential scan over class extension and sequential scan over all class instances of a class, and hash or B-Tree index scan on class extension as well as the scan over all class instances.

B. Parser

This module includes the implementation of a parser for the algebra expressions. The algebra expressions are checked for syntactical errors and then translated into query trees.

Lexical analysis is implemented using a simple lexer which converts a query into a sequence of tokens including operation codes and constants. Parsing and translation are realized by a top-down examination of the query expressions based on methods for LL(1) grammars.

The query expressions are represented during the query compilation and execution as query trees. A *query tree* is a graph (dag) which vertices are query nodes. The vertices are linked with respect to the parse tree. Two types of nodes are used for the representation of queries: the *query nodes* are tree nodes representing set operations (e.g. operation join),

 ISBN: 978-1-61208-247-9

and the *parameter nodes* are tree nodes for the representation of parameter expressions (eg. join parameter expression). The vertices represent all model operations: arithmetic operations, logical operations, comparison operations, schema manipulation operations as well as variables.

The basic skeleton of the query tree is constructed during the parsing process. The variables and the names of the data sources and spans are stored in symbol table `symtab`. The query nodes represent the operations, spans linking rules, and access methods. The data for the different phases of query processing is stored in the same tree nodes. Each particular module (e.g. query optimization) manipulates its own view of query nodes.

1) Representing predicates: The parameters of the set operations select, project and join are abstracted using *parameter nodes*. Parameter node is an abstraction of parameter expression. It stores information about the query subtree corresponding to the parameter expression: it includes references to the root of expression sub-tree, reference to the variables, and references to the data sources of variables. The parameter nodes play vital role for the representation of rules as well as for the rule matching procedure. After a parameter expression is represented by means of a tree and tied to the query tree, a method for the evaluation of parameter node can compute the output from given inputs by means of a tree representing parameter expression.

Symbol table `symtab` is used for handling query variables during parsing and type-checking, and for preparing rules for matching. A symbol name appearing in the query expression can be either the name of data source, query variable or the name of a span. The entry of symbol table includes the name of symbol and the reference to the query node representing data source, variable or span.

2) Path expressions: Path expressions of RDF query language SPARQL [27] called *property paths* are more expressive than the path expressions of classical relational and object-relational systems.

The path expressions that have to be implemented are expressions on properties that are based on regular expressions. Query processing system treats complex path expressions as predicates of relational system. For each objects of a collection complex path expression is evaluated separately.

C. Query optimizer

The design of the query optimizer is based in many aspects on the design of System R [1] and on the work of Graefe [10], [12]. The query optimizer performs a global optimization at master site.

The query expressions are in the query optimizer represented by query trees [14]. The inner vertices of the query tree represent operations comprising the query. The leaves of the query tree represent access paths. The logical equivalences among the query expressions are presented by means of transformation rules [7]. The internal representation of the transformation rules is also based on the query tree representation.

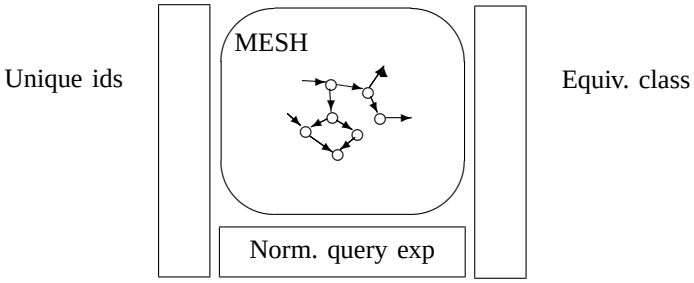

Figure 1. Mesh with access paths

The query optimization subsystem is composed of the following main modules. The query tree manipulation module includes routines for manipulation of query trees, application of rules on the nodes of query trees, and property functions by which the physical and logical properties of the query tree nodes are determined.

The cost function is realized by a separate module. The extensions of the cost functions used for the relational query optimizer are defined. The optimization module comprises the algorithms for the optimization of query expressions. The algorithms which are studied are the exhaustive search and an algorithm based on dynamic programming.

The core of the module is data structure `mesh` for the representation of sets of queries. Common sub-expressions of queries are shared ie. each query is represented in `mesh` only once. Queries are organized into equivalence classes. Let us first present the data structure Mesh.

1) Mesh: `mesh` stores query expressions as query trees organized as a directed acyclic graph (dag). The query trees share common parts hence there is only one representation of a query expression in `mesh`. Further, the query trees are organized into equivalence classes including logically equivalent queries. `mesh` has three entry points.

First, queries can be accessed using the unique identifiers which are created when query is entered in `mesh`. The mapping is realized between query trees (roots of) and the entries in `mesh` holding the queries.

Second, queries in `mesh` can be accessed through the equivalence classes which again have unique identification generated on the creation of the equivalence class from the first query expression. The inverse relationship from the query trees (roots of) to the equivalence classes is also defined.

Third, queries can be accessed using the normalized query expressions (character strings) as the hash index targeting roots of corresponding query trees.

In spite of various access methods to `mesh`, queries are defined free of the cover data structures. Each query node includes solely a reference to the equivalence class to which it belongs.

 ISBN: 978-1-61208-247-9

2) Optimization algorithm: The algorithm for query optimization is based on dynamic programming. The optimization is performed top-down: the procedure starts in the root, descends recursively to sub-trees, computes all logically equivalent queries, places them in the equivalence class, picks the next query from equivalence class and starts at the beginning until all alternatives are considered.

From the point of view of the actual computation of optimal query tree, this query optimization algorithm works bottom-up: the leaves of the query tree are optimized first progressing then upwards toward the root of the query tree.

We use the variant of dynamic programming algorithm called *memoisation* which stores the optimal results of the sub-queries and uses them in the computation of the composed queries. Memoisation is realized in a very simple manner. Every time a query is to be optimized we first check Mesh if the optimal query already exists for a given equivalence class of input query.

At this point we can see the use of Mesh for *plan caching*. If we do not clean Mesh after the execution of a given query that the optimization of the subsequent queries can make use of the existing optimized queries in Mesh. The optimization time is reduced significantly.

Let us now present some aspects of the complexity of query optimization algorithm. The equivalence class comprises a sorted list of queries that are the candidates for the optimization. The cost function is used to order the list. The search space can be reduced by selecting only the most promising queries. This type of optimization is called a *heuristic beam search*. The algorithm is sub-optimal since heuristic function based on cost estimation is not admissible or monotonic.

The practical experiences show that the presented algorithm based on dynamic programming is fast enough for the optimization of complex bushy trees with up to 10 classes.

3) Rules: The *query transformation rules* are used for the transformation of query trees into logically equivalent query trees that have different structure and potentially a faster evaluation method. The logical optimization rules are in Qios specified in a language that follows strictly the syntax and the semantics of the Qios query expressions.

A rule is composed of the *input pattern* and the *output pattern*. The input and output patterns are connected by means of common variables and common data sources. The links among the variables define the meaning of the rule. The input and output patterns, ie. query expressions, are ”terminated” using special operators called *spans*. A span is a virtual operation which can match any algebra operation. The rule (i/o) patterns can then be seen, from the perspective of parse trees, as the upper parts of query trees with the abstract leaves.

The query transformation rules are from the external form (augumented query expressions) transformed into the representation which is based on the query tree representation of query expressions. The input and output patterns of the rule are represented by the input and output query trees. These are connected by the links relating variables, data sources and parameters of the input and output patterns of rule as presented in Figure 2.

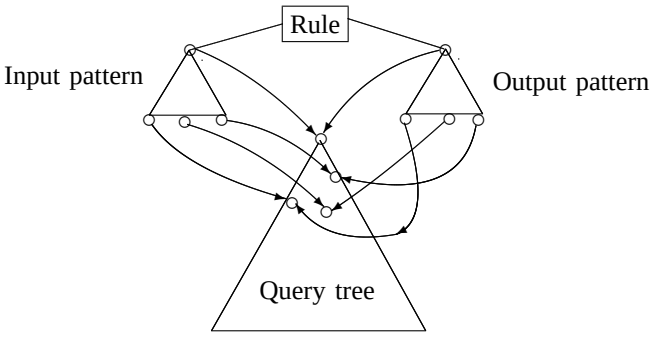

Figure 2. Query transformation

Given a query tree and a rule, the matching procedure can be viewed as an attempt to cover the root of query tree with the input rule pattern ie. query tree. If matching is successful, types of the output pattern of rule are checked in order to verify correctness of query expression. If type checking procedure succeeds the output pattern of the rule is duplicated generating in this way a logically equivalet query of different structure.

D. Query evaluation system

The query evaluation module is based on the iterator-tree representation of the query evaluation plans. The physical query execution plan is computed from the optimized query trees by adding to the existing query nodes information about physical operation that will implement given logical operation (query node). The query nodes already contain information about the statistics, index selection, and cost estimation.

The main strategy which was used in the implementation of query execution is to select reasonably fast access methods on-the-fly without considering alternatives. Simple rules are used for index selection. Firstly, if selection or join is based on equality of attributes than hash-based index is generated. Secondly, if selection or join involves range predicate we use B-tree index. The selection of query execution plan is implemented by the procedure which computes physical operations for all logical operations (query nodes) forming the physical query tree in a bottom-up manner.

The physical algebra operations implemented are sequential scan, nested-loop join, hash-based index, and index-based nested-loop join. The hash-based access methods allow the efficient evaluation of the queries including equality and index-based plan serves for the evaluation of range queries. The procedures for the selection of physical opera-

 ISBN: 978-1-61208-247-9

tions are integrated in the routines that open scan operator for query nodes (logical operators).

After the selection of the physical operations the same skeleton of the query tree is employed as the iterator tree for the evaluation of physical query. The query evaluation can be seen as the tree structured pipeline where sub-nodes need to provide the next tuple for the evaluation of the current query node [10], [12]. The result of the logical query optimization are bushy query trees. The results of the inner sub-trees have to be materialized in order to avoid repeating evaluation.

E. RDF loader

RDF loader can read RDF files from Internet or locally. Loader is implemented using RDF::Trine module. Document is loaded into internal data model triple at a time. Schema must be loaded first to construct classes and types of the repository. Further, concrete triples are loaded to construct objects which must be instances of previously loaded classes.

There are many examples of RDF data repositories on the Internet that do not include schema part. For this purpose, RDF loader can compute on-the-fly the least general type (class) covering all instance of a given kind. Algorithm for RDF schema discovery is simple since we build objects on the basis of common object identifier that gathers triples with the same domain.

Currently we do not implement complete RDFS. More study is needed to be able to adequately model properties as objects i.e. instances of classes that can be specialised.

III. Related work

3store is designed to handle up to 100M serialized triples. They use three layered model; RDF representation can be syntactical, model-based in the form of triples and stored in MySql database system. Two tables are used for indexing resources and literals. Index value (hash) serves as unique identifier of resources and literals. These unique identifiers are then used for table representing triples. SPARQL queries are translated into relational calculus that can be directly transformed to SQL.

Bigdata is a triple-store that uses three main relations to store RDF database: Lexicon, Statements (triples) and StatementTypes. Exhaustive set of indexes is created for relation Statements which represents triples. RDF query is based on statement patterns. Bigdata defines a perfect access path for each type of access pattern. System includes also a form of reasoning with triples as well as reasoning on the basis of RDFS statements. Furthermore, the system is scalable to several 100 machines.

Virtuoso is a relational system including functionalities for relational and object-relational data management, XML and RDF data management, Web services deployment, text content management services, full-text indexing as well as Web document server and Linked-data server. RDF data is stored in a table including triples which can be indexed using object-relational system. SPARQL is implemented by translating queries into SQL—equivalent queries can be expressed directly in SQL.

Let us now present the work related to the implementation of the presented query execution system. Firstly, the implementation is closely related to the implementation of Query Algebra originally proposed by Shaw and Zdonik in [26] and implemented by Mitchell [18]. In particular, we have used a similar representation of query expressions by means of query trees. Furthermore, the representation of query expressions in Qios is optimized by using single operation nodes and query trees during all phases of query processing.

The design of the query execution system was based on the design of Exodus optimizer generator [9] and its descendant Volcano [12]. The data structure MESH used in Exodus query optimizator generator is improved by adding additional access paths. The data structure can be accessed through: unique identifier, normalized query expression, and equivalence class. The algorithm for query optimization is rooted in Graefe's work on Volcano optimizer algorithm [12]. This algorithm uses top-down search guided by the possible "moves" that are associated to a query node. The algorithm uses memoisation to avoid repeated optimization of the same query. The search is restricted by the cost limit which is a parameter in optimization.

IV. Conclusions

The paper presents the architecture of query engine used for querying RDF databases. While the architecture is based on relational and object-relational technology there are many specific components and techniques which are implemented to reflect RDF data model and the nature of RDF databases.

Firstly, since RDF model uses triples for modeling conceptual schemata and instances the data model of storage system maintains a single set of object identifiers. Object values are tuples including concrete values while class values include class identifies and represent types. Classes are in many ways treated as ordinary objects reflecting the graph data model of RDF. Secondly, path expressions of RDF data model are more expressive than path expressions of object-relational data model [27]. For this reason treatment of predicates of selection and join operations must be considered much more carefully. Finally, algebra of query processing system have to include operations for grouping and restructuring to be able to support complete functionality of SPARQL.

References

[1] M. Astrahan, Et.al., 'System R: relational approach to database management', ACM Transactions on Database Systems, Vol. 1, Issue 2 June 1976, pp. 97-137.

 ISBN: 978-1-61208-247-9

[2] F. Bancilhon, 'Object-Oriented Databases', The Computer Science and Engineering Handbook, 1997.

[3] 'Bigdata Scale-out Architecture', Whitepaper, SYSTAP, 2009.

[4] E.F. Codd, 'A relational model of data for large shared data banks', Comm. of the ACM, Vol. 13 , Issue 6, June 1970, pp. 377-387.

[5] A. de Jonge, 'Comparing XML database approaches', IBM Corporation, 2008.

[6] D. Daniels, P. Selinger, L. Haas, B. Lindsay, C. Mohan, A. Walker, P. Wilms, 'An introduction to distributed query compilation in R*', IBM Research Report RJ3497 (41354), June 1982.

[7] J.C. Freytag, 'A rule-based view of query optimization', Proc. of ACM Conf. on Management of data, 1987.

[8] G. Gardarin, J.-R. Gruser, Z.-H. Tang, 'Cost-based selection of path expression processing algorithms in object-oriented databases', In Proc. Int. Conf. on VLDB, 1996.

[9] G. Graefe, D.J. DeWitt, 'The EXODUS Optimizer Generator', Proceedings of ACM SIGMOD international conference on Management of data, 1987, pp. 160-172.

[10] G. Graefe, 'Query Evaluation Techniques for Large Databases', *ACM Comp. Surveys*, Vol.25, No.2, June 1993, pp. 73-170.

[11] G. Graefe, 'Query processing', Slides, ICDE Influential Paper Award, 2005.

[12] G. Graefe, W. McKenna, 'The Volcano optimizer generator: extensibility and efficient search', Proc. of IEEE Conf. on Data Engineering, April 1993, p.209.

[13] S. Harris, N. Gibbins, '3store: Efficient Bulk RDF Storage', Proceedings 1st International Workshop on Practical and Scalable Semantic Web Systems, 2003.

[14] M. Jarke, J. Koch, 'Query optimization in database systems', *ACM Comp. Surveys*, Vol.16, No.2, June 1984.

[15] 'Linked Data - Connect Distributed Data across the Web', http://linkeddata.org/, 2012.

[16] A. Marathe, 'Batch Compilation, Recompilation, and Plan Caching Issues in SQL Server 2005', Microsoft, White paper, Dec. 2006.

[17] T. Milo, D. Suciu, 'Index structures for path expressions', In Proc. 7th Int. Conf. on Database Theory, 1999.

[18] G.A. Mitchell, 'Extensible Query Processing in an Object-Oriented Database', Ph.D. thesis, Brown University, 1993.

[19] 'Oracle Database, Documentation Library (11.2)', Oracle Corporation, 2012.

[20] 'Resource Description Framework (RDF)', http://www.w3.org/RDF/, 2004.

[21] M.A. Roth, H.F. Korth, A. Silberschatz 'Extended algebra and calculus for non 1NF relational databases', *ACM Trans. Database Systems*, 13(4), 1988, 389-417.

[22] I. Savnik, 'Algebra for distributed data sources', Dagstuhl Seminar "Multimedia Database Support for Digital Libraries", Schloss Dagstuhl, Germany, Aug 1999.

[23] I. Savnik, Z. Tari, T. Mohorič, 'QAL: A Query Algebra of Complex Objects', *Data & Knowledge Eng. Journal*, North-Holland, Vol.30, No.1, 1999, pp.57-94.

[24] I. Savnik, 'On formalization of object model by unifying intensional and extensional representations', Information Modelling and Knowledge Bases XXI, Frontiers in Artificial Intelligence and Applications, IOS Press, 2010.

[25] I. Savnik, Z. Tari, 'QIOS: Querying and Integration of Internet Data', http://osebje.famnit.upr.si/~savnik/qios/, Famnit, University of Primorska, Koper, 2009.

[26] G.M. Shaw, S.B. Zdonik, 'A Query Algebra for Object Oriented Databases', Proc. of IEEE Conf. on Data Engineering, 1990, pp. 154-162.

[27] S. Harris, A. Seaborne, 'SPARQL 1.1 Query Language', W3C Working Draft, http://www.w3.org/TR/sparql11-query/, 2012.

[28] 'Information technology, Database languages, SQL', ISO/IEC JTC 1/SC 32, ANSI, 2007.

[29] 'OpenLink Virtuoso Universal Server: Documentation', OpenLink Software Documentation Team, 2009.

 ISBN: 978-1-61208-247-9