OneQL: An Ontology-based Architecture to Efficiently Query Resources on the Semantic Web

Tomas Lampo[1] and Edna Ruckhaus[1] and Javier Sierra[1] and María-Esther Vidal[1] and Amadís Martínez[1,2]

[1] Universidad Simón Bolívar, Caracas, Venezuela
{tomas,ruckhaus,javier,mvidal}@ldc.usb.ve
[2] Universidad de Carabobo, Valencia, Venezuela {aamartin}@uc.edu.ve

Abstract. The widespread explosion of Web accessible resources has led to new problems on the traditional tasks of query evaluation and efficient data access. With this is mind, we have developed the ontology-based OneQL system which provides optimization and query evaluation techniques to scale up to large RDF/RDFS documents and complex queries, i.e., queries of any shape and with a large number of triple patterns. Efficiency of OneQL relies on the following components:

- Query optimization and evaluation techniques that focus on cost models to estimate the execution time of a plan, and on searching the space of plans of any shape, i.e., bushy plans can be generated according to their estimated cost.
- *Bhyper*: A directed hypergraph-based representation of RDF documents to directly access triples that share the same subject and property values, or the same property and object values.

We report on the quality of the developed strategies, and have observed that implementing RDF documents with *Bhyper* and producing low estimated cost bushy plans, can speed up the evaluation time by up to four orders of magnitude.

1 Introduction

Emerging infrastructures such as the Semantic Web, the Semantic Grid and Service Oriented architectures, support on-line access to a wealth of ontologies, data sources and Web services. Ontologies play an important role in the Semantic Web and provide the basis for the definition of concepts and relationships that make information integration possible. Knowledge represented in ontologies can be used to annotate data, distinguish similar concepts, and generalize and specialize concepts published by data sources or produced by Web services. A great number of ontologies have become available under the umbrella of the Semantic Web; some of these ontologies can be very large, impacting in this way the tasks of ontology query answering and reasoning; for instance, MeSH, NCI Cancer, and GO are good examples of ontologies comprised of thousands of concepts. Furthermore, the number of available Web data sources and services has exploded during the last few years. For example, currently, the molecular biology databases collection includes 1,078 databases [12], that is 110 more than the previous year [11]; tools and services as well as the number of instances published

by these resources, follow a similar progression [6]. In addition, thanks to this wealth, users rely more on various digital tasks such as data retrieval from public data sources and data analysis with Web tools or services organized in complex workflows. Thus, in order to be capable of scaling up, Web architectures have to be tailored for query processing on large number of resources and instances. We have aimed at these problems, and have proposed the OneQL system.

OneQL is based on query optimization and evaluation techniques to efficiently execute SPARQL queries. Ontologies are implemented as a deductive database whose predicates represent knowledge explicitly expressed in the ontology, and the semantics of the vocabulary terms. To efficiently store and index the RDF documents where ontologies are defined, we have proposed a directed hypergraph formal model named *Bhyper*. Basically, a *Bhyper* structure is defined by a set of nodes and a set of hyperarcs; each hyperarc connects a set of source nodes to a set of target nodes. In a *Bhyper* structure, the information is stored only in the nodes, and the hyperarcs preserve the role of each node and the concept of direction of RDF graphs. Thus, each resource (subject, property, or value) is stored only once, and the space complexity of an RDF document is reduced if a resource appears several times in the document. Besides, *Bhyper* structures define implicit position-based indices [25] for an RDF document, which can support efficient evaluation of queries over the document.

This paper is comprised of seven sections. The next section summarizes the related work. In section 3 we briefly describe the OneQL system architecture. We then discuss our research in query optimization and evaluation. Section 5 describes *Bhyper*, a *hypergraph*-based representation for RDF documents. The experimental study is reported in section 6, and finally, section 7 outlines our conclusions and future work.

2 Related Work

In the context of the Semantic Web, several query engines have been developed to access RDF documents efficiently [4, 13–16, 20, 31]. Jena [15, 32] provides a programmatic environment for SPARQL, and it includes the ARQ query engine and indices which provide an efficient access to large datasets. The ARQ-Optimizer is a system that implements heuristics for selectivity-based Basic Graph Pattern optimization, proposed by Stocker et al. [28]. These heuristics range from simple triple pattern variable counting to more sophisticated selectivity estimation techniques; the optimization process is based on a greedy optimization algorithm which may explore a reduced portion of the space of possible plans, i.e., only left linear plans. Hence, ARQ-Optimizer query plans can sometimes be far from the optimal plans. Tuple Database or TDB [16] is a persistent graph storage layer for Jena. TDB works with the Jena SPARQL query engine (ARQ) to support SPARQL together with a number of extensions (e.g., property functions, aggregates, arbitrary length property paths). It is a pure-Java component that employs memory mapped I/O, and a customized implementation of B^+-trees to index three different triple patterns permutations, i.e., *spo*, *pos*, and *osp*.

Sesame [31] is an open source Java framework for storage and querying RDF data. It supports SPARQL and SeRQL queries which are translated to Prolog; the join operator is implemented as sideways-passing of variable bindings, which is similar to our

Index Nested Loop Join (NJoin) operator. YARS2 (Yet Another RDF Store, Version 2) [13] is a federated repository for queries against indexed RDF documents. YARS2 supports three types of indices that enable keyword lookups, perform atomic lookup operations on RDF documents, and speed up combinations of patterns or values. Indices are implemented by using an in-memory sparse index data structure that refers to on-disk block entries which contain the indexed entry; six combinations of triple patterns are indexed. A general query processor on top of a distributed Index Manager was implemented, and SPARQL queries are supported; however, no SPARQL specific optimization or evaluation techniques have been developed.

RDF-3X [20] focuses on an index system, and its optimization techniques were developed to explore the space of plans that benefit from these index structures. RDF-3X query optimizer implements a dynamic programming-based algorithm for plan enumeration, which imposes restrictions on the size of queries that can be optimized and evaluated. Indeed, in certain cases, these index-based plans could coincide with OneQL optimized plans; however, the RDF-3X optimization strategies are not tailored to identify any type of bushy plans or to scale up to queries with at least one Cartesian product.

AllegroGraph [4] uses a native object store for on-disk binary tree-based storage of RDF triples. AllegroGraph also maintains six indices to manage all the possible permutations of subject (s), predicate (p) and object (o). The standard indexing strategy is to build indices whenever there are more than a certain number of triples. The query optimizer is based on join ordering for the generation of execution plans; no bushy plans are generated. Hexastore [30] is a main memory indexing technique that uses the triple nature of RDF as an asset. RDF data is also indexed in six possible ways, one for each possible triple pattern permutation. However, the prime drawback of the Hexastore lies in storage space usage; it may require a five-fold increase in storage space compared to a triple table; also, the same resource can appear in multiple indices. Furthermore, two second memory index-based representations and evaluation techniques are presented in [8, 19]. [8] propose indexing the universe of RDF resource identifiers, regardless of the role played by the resource; although they are able to reduce the storage costs of RDF documents, since the proposed join implementations are not closed, the properties of the index-based structures can only be exploited in joins on basic graph patterns. In contrast, [19] propose an index-based representation for RDF documents that maintains the results for subject-subject joins, object-object joins and subject-object joins. Although these structures can speed up the evaluation of joins, this solution may not scale up to strongly connected very large RDF graphs.

GiaBATA [14] is a SPARQL engine built on top of the dlvhex reasoning engine for HEX-programs, and the DLVDB [29] ASP solver with persistent storage. GiaBATA does not implement an RDF-based cost model, but purely relies on join reordering optimizations of DLV and optimizations of the underlying relational database system.

Finally, [1, 2, 27] propose different RDF store schemas to implement an RDF management system on top of a relational database system. They empirically show that a physical implementation of vertically partitioned RDF tables, may outperform the traditional physical schema of RDF tables. Similarly to some of the existing state-of-the-art RDF systems, the optimization techniques are not tailored to identify bushy plans.

3 Architecture

Figure 1 presents the architecture of the OneQL system; it is comprised of a *Query Planner*, a *Query and Reasoning engine*, and an *Ontology catalog* [23].

Ontologies are modeled as a deductive database (DOB) which is composed of an extensional and an intensional database. The extensional database (EDB) is comprised of meta-level predicates that represent the information explicitly modeled by the ontology; for each ontology language built-in vocabulary term, there exists a meta-level predicate (e.g., $subClassOf$). The intensional database (IDB) corresponds to the deductive rules that express the semantics of the vocabulary terms (e.g., the transitive properties of the $subClassOf$ term).

Queries are described as SPARQL queries and are posted to OneQL through a SPARQL-based API which translates each query into a conjunctive query on the predicates in DOB. The conjunctive query is then passed to the optimizer which uses statistics stored in the catalog and Magic Sets rewriting techniques to identify an efficient query execution plan. Next, the plan is given to the query and reasoning engine, which evaluates it against DOB.

The statistics that describe the ontologies stored in DOB include: cost of inferring intensional facts, cardinality of extensional and intensional facts, and number of resources. These statistics are used by the *hybrid cost model* to estimate the cost of a given query plan. Finally, a *hypergraph*-based structure named *Bhyper* is used to index predicates in DOB and to speed up the tasks of query processing and reasoning.

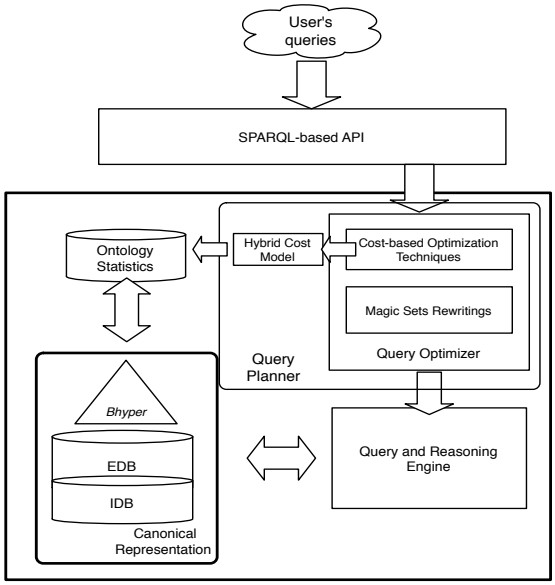


Fig. 1. The OneQL System Architecture

4 Optimizing and Evaluating SPARQL queries

OneQL implements optimization and evaluation techniques to support the execution of SPARQL queries. The proposed optimization techniques are based on a cost model that estimates the execution time or facts inferred during query evaluation; they are able to produce query plans of any shape.

The *Query Planner* component in OneQL (Figure 1) is built on top the following two sub-components [22–24]: a hybrid cost model that estimates the cardinality and evaluation cost of the predicates that represent the ontology's explicit and implicit facts, and a twofold optimization strategy to identify bushy plans. The *Query Engine* relies on several physical operators and *Bhyper*-based indices to efficiently evaluate SPARQL queries.

4.1 The Hybrid Cost Model

In the hybrid cost model, evaluation cost is measured in terms of the number of intermediate inferred facts, and the cardinality corresponds to the number of valid answers of the query pattern. This model estimates the cost and cardinality of explicit and implicit facts, as follows:

- To estimate the cardinality and cost of the intensional predicates that represent implicit facts, we have applied the Adaptive Sampling Technique [17]. This method does not need to extract, store or maintain information about the data that satisfies a particular predicate, and does not make any assumptions about statistical characteristics of the data, such as distribution. Sampling stop conditions are defined to ensure that the estimates are within an appropriate confidence level.
- To estimate the cardinality and cost of the extensional predicates, and the cost of a query plan, we use a cost model *à* la System R [26]. Similarly to System R, we store information about the number of ground facts corresponding to an extensional predicate, and the number of different values (constants) of each predicate variable. Formulas for computing the cost and cardinality are similar to the different physical *join* formulas in relational queries.

4.2 The TwoFold Optimization Technique

A twofold optimization strategy that combines cost-based optimization and Magic Sets techniques was developed. In the first stage of the query optimization component, dynamic-based or randomized algorithms can be applied to identify a good ordering or grouping of the patterns in a SPARQL query. On one hand, the dynamic-programming algorithm works on iterations, and during each iteration the best intermediate sub-plans are chosen based on the cost and the cardinality that were estimated using the hybrid cost model. In the last iteration of the algorithm, final plans are constructed and the best plan is selected in terms of the estimated cost. This optimal ordering reflects the minimization of the number of intermediate inferred facts using a top-down evaluation strategy. This dynamic-based algorithm is performed on queries with a small number of

triple patterns in the where clause, and it is able to produce only left-linear plans which are not always the best solution for RDF-based queries.

On the other hand, the randomized algorithm performs random walks over the search space of bushy execution plans; the query optimizer implements a Simulated Annealing algorithm. Random walks are performed in stages, where each stage consists of an initial *plan generation step* followed by one or more *plan transformation steps*. An equilibrium condition or a number of iterations determines the number of transformation steps. At the beginning of each stage, a query execution plan is randomly created in the plan generation step. Then, successive *plan transformations* are applied to the query execution plan during the plan transformation steps, in order to obtain new plans. The probability of transforming a current plan p into a new plan p' is specified by an acceptance probability function $P(p, p', T)$, that depends on a global time-varying parameter T called the *temperature*; it reflects the number of stages to be executed. The function P may be nonzero when $cost(p') > cost(p)$, meaning that the optimizer can produce a new plan even when it is worse than the current one, i.e., it has a higher cost. This feature prevents the optimizer from becoming stuck in a local minimum. Temperature T is decreased during each stage and the optimizer concludes when $T = 0$. Transformations applied to the plan during the random walks correspond to the SPARQL axioms of the physical operators implemented by the query and reasoning engine. The Simulated Annealing-based optimizer scales up to queries of any shape and number of triple patterns, and is able to produce execution plans of any shape.

In the second stage, the optimizer applies Magic Set optimization techniques [21] to the execution plan obtained in the first stage. Magic Sets combines the benefits of both, top-down and bottom-up evaluation strategies and tries to avoid repeated computations of the same subgoals, and unnecessary inferences. The deductive database program DOB is rewritten w.r.t. the optimal execution plan, and then evaluated with a bottom-up strategy. "Magic predicates" are inserted into the program to represent bounded arguments in the query, and "Supplementary predicates" are included to represent sideways information-passing in rules. It should be noted that we implemented the general Magic Sets technique for Datalog with the two improvements suggested by [3] to eliminate the first and last redundant supplementary predicates, and to merge consecutive sequences of extensional predicates in rule bodies.

4.3 The OneQL Query Engine

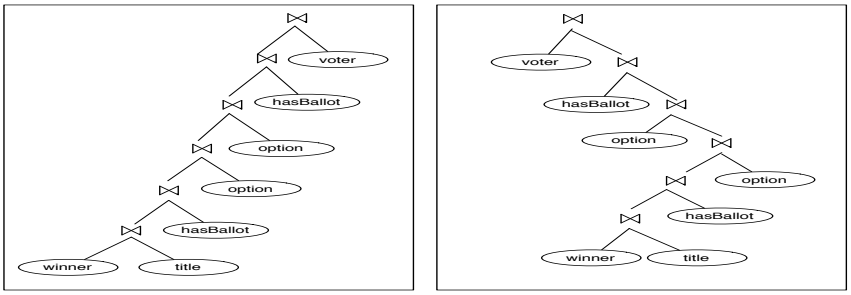
The query and reasoning engine implements different strategies (operators) used to retrieve and combine ontology facts. We have defined different operators that implement the retrieval and combination of ontology facts, and make use of the direct access provided by the *Bhyper*-based structures:

1. Index Nested-Loop Join. For each matching triple in the first pattern, we retrieve the matching triples in the second pattern, i.e., the join arguments[3] are instantiated in the second pattern through the sideways passing of variable bindings. The Index Nested-Loop Join was implemented by extending the sideways-passing of information inherent to Prolog rules, with *Bhyper* indices that allow a direct access to

[3] The join arguments are the common variables in the two predicates that represent the patterns.

the inner pattern triples that match the join variable values of each outer pattern triple; once a result is produced, the computation of the operator is forced to fail, and backtracking takes place to produce a new answer.

2. Group Join. The main idea of this operator is to partition the patterns that appear in the 'WHERE' clause of a query into groups that are comprised of a relatively small number of triples. The Group Join was implemented by first evaluating each group independently, and then asserting in the SWI-Prolog main memory database the results produced by each group; the main memory predicates used to temporally store the results of each group, are indexed by using SWI-Prolog indices. Finally, the main memory stored results are checked to identify matches. Similarly to the Index Nested-Loop Join, the Group Join control is implemented by forcing the computation of the operator to fail when a solution is produced, and using backtracking to generate more solutions.



(a) Left-linear tree (b) Bushy tree

Fig. 2. Query Execution Plans

To illustrate the behavior of the proposed optimization and evaluation techniques, consider the dataset that publishes the US Congress bills voting process [4]. Suppose that the following SPARQL query is posed against OneQL: *Select all the bills and their title where "Nay" was the winner, and at least one voter voted for the same option than the voter L000174.*

```
PREFIX  vote: <tag:govshare.info,2005:rdf/vote/>
PREFIX  dc: <http://purl.org/dc/elements/1.1/>
PREFIX people:
<http://www.rdfabout.com/rdf/usgov/congress/people/>
SELECT  ?E ?T FROM <http://example.org/votes>
WHERE {?E  vote:winner 'Nay' .
       ?E dc:title ?T . ?E vote:hasBallot ?I .
       ?I  vote:option ?X .?J vote:option ?X .
       ?E vote:hasBallot ?J .
       ?J vote:voter 'people:L000174'}
```

[4] http://www.govtrack.us/data/rdf/

Following the optimization techniques reported in [28], only left linear plans as the one reported in Figure 2(a) will be produced; for this left linear plan, the evaluation time is 8,466 secs. On the other hand, our proposed optimization techniques are able to produce bushy trees as the one reported in Figure 2(b) whose evaluation time is 122 secs, i.e., one order of magnitude cheaper than the left linear plan.

5 Bhyper: A Hypergraph-based representation for RDF/RDFS documents

OneQL stores RDF triples using a directed hypergraph-based representation [18]. Basically, a directed hypergraph is defined by a set of nodes and a set of hyperarcs, each one of them connecting a set of source nodes (named tail of the hyperarc) to a set of target nodes (named head of the hyperarc). Directed hypergraphs have been successfully used as a modeling tool to represent concepts and structures in many application areas: formal languages, relational databases, production and manufacturing systems, public transportation systems, topic maps, among others [5, 9, 10]. An RDF directed hypergraph is defined as follows:

Let D be an RDF document. We define a *Bhyper* RDF representation D as a tuple $\mathcal{H}(D) = (W, E, \rho)$ such that:

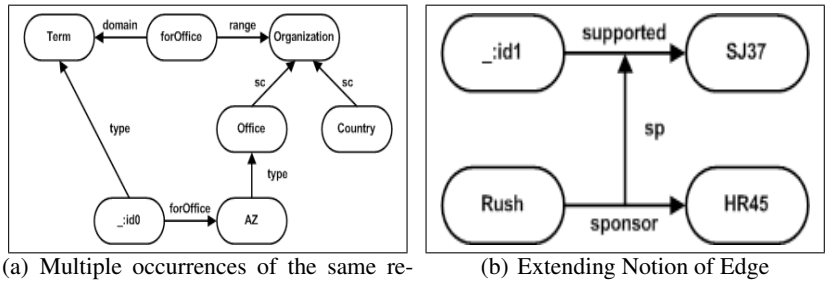
- $W = \{w : w \in univ(D)\}$ is the set of nodes.
- $E = \{e_i : 1 \leq i \leq |D|\}$ is the set of hyperarcs.
- $\rho : W \times E \rightarrow \{s, p, o\}$ is the role function of nodes w.r.t. hyperarcs. Let $t \in D$ be an RDF triple, $e \in E$ an hyperarc, and $w \in W$ a node such that $w \in head(e) \cup tail(e)$. Then the following must hold:
  - $(\rho(w, e) = s) \Leftrightarrow (w \in tail(e)) \wedge (w \in sub(\{t\}))$
  - $(\rho(w, e) = p) \Leftrightarrow (w \in tail(e)) \wedge (w \in pred(\{t\}))$
  - $(\rho(w, e) = o) \Leftrightarrow (w \in head(e)) \wedge (w \in obj(\{t\}))$

The *Bhyper* representation reduces space complexity to store the RDF document and speeds up the data recovery process. To illustrate the benefits of the *Bhyper*-based representation and the main drawbacks of the traditional graph-based representation, we use some examples extracted from the US Congress bills voting process dataset.

First, consider the RDF document D_1 ={*(_:id0, type, Term), (_:id0, forOffice, AZ), (AZ, type, Office), (Office, subClassOf, Organization), (Country, subClassOf, Organization), (forOffice, range, Organization), (forOffice, domain, Term)*}, where the resource *forOffice* occurs as a predicate and a subject. This situation can be modeled by allowing multiple occurrences of the same resource in the resulting labeled directed graph, as arcs or nodes labels (Figure 3(a)). However, this violates one of the most important aspects of graph theory: the intersection between the nodes and arcs labels must be empty.

Second, a predicate may relate other predicates in an RDF document. For example, in the RDF document D_2 = {*(Rush, sponsor, HR45), (_:id1, supported, SJ37), (sponsor, subPropertyOf, supported)*} the predicate *subPropertyOf* relates the predicates *sponsor* and *supported*. This situation can be modeled extending the notion of arc by allowing the connection between arcs (Figure 3(b)). However, the resulting structure is not a

graph in the mathematical sense, because the set of arcs must be a subset of the Cartesian product of the set of nodes. Since these two simple situations violate some of the graph constraints, it is not possible to use concepts and search algorithms of graph theory to manipulate RDF documents. Thus, while the labeled directed graph model is the most widely used representation, it cannot be considered a formal model for RDF [7].



(a) Multiple occurrences of the same resource

(b) Extending Notion of Edge

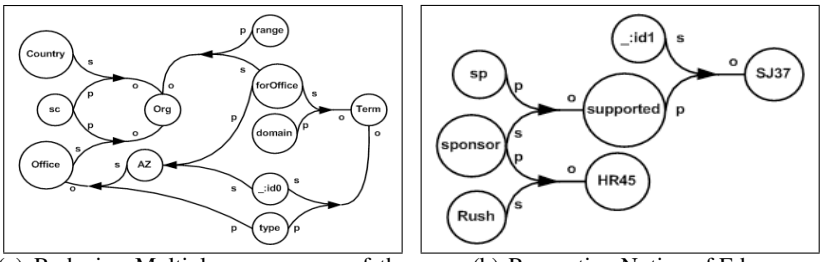
Fig. 3. RDF document properties

Figures 4(a) and 4(b) show the RDF directed hypergraphs representing RDF documents D_1 and D_2, respectively. In *Bhyper*, given an RDF document D, each node corresponds to an element $w \in univ(D)$. Thus, the information is only stored in the nodes, and the hyperarcs only preserve the role of each node and the concept of direction of RDF graphs. An advantage of this representation is that each resource (subject, property, or value) is stored only once, and the space required to store an RDF document is reduced if a resource appears several times in the document. In this way, the space complexity of our approach is lower than the complexity of the graph-based RDF representation. Besides, concepts, techniques, and algorithms of hypergraph theory can be used to manipulate RDF documents more efficiently.

The *Bhyper* indices were implemented in Prolog by using two extensional predicates: *subject(S,P,Lo)* and *object(O,P,Ls)*. The predicate *subject* associates a given subject value S with the property P that relates it with the object values in the list Lo. Similarly, the predicate *object* maps an object value O with the property P that relates it with the subject values in the list Ls. Both predicates are indexed on the first and second arguments with the SWI-Prolog indices. OneQL predicates *subject* and *object* resemble the property tables implemented in Jena2 to speed up queries over the same subject or object values [32].

6 Experimental Results

We conducted an experimental study to empirically analyze the effectiveness of the OneQL optimization and evaluation techniques. We report on the evaluation time performance of bushy plans comprised of groups and identified by our proposed query optimizer.



(a) Reducing Multiple occurrences of the same resource (b) Respecting Notion of Edge

Fig. 4. The *Bhyper*-based representation

Dataset and Query Benchmark: We use the real-world dataset on US Congress vote results of the 2004 bills voting process described in Figure 5(b). The entire dataset was downloaded and locally stored in flat files; the total size is 3.613 MB and 67,392 triples. We considered two sets of queries. Benchmark one is a set of nine queries which are described in Figure 5(a) in terms of the number of patterns in the WHERE clause and the answer size; all the queries have at least one pattern whose object is instantiated with a constant. Benchmark two is a set of 60 queries which have between one and seven GJoin(s) among small size groups of patterns and have more than 12 triple patterns. These two benchmarks are published in http:www.ldc.usb.ve/˜mvidal/OneQL/datasets.

Evaluation Metrics: We report on runtime performance, which corresponds to the *user time* produced by the *time* command of the Unix operation system. OneQL was implemented in SWI-Prolog (Multi-threaded, 64 bits, Version 5.6.54). The randomized optimizer was run for 20 iterations at an initial temperature of 700. The experiments were evaluated on a Solaris machine with a Sparcv9 1281 MHz processor and 16GB of RAM.

query	#patterns	answer size
q1	4	3
q2	3	14033
q3	7	3908
q4	4	14868
q5	4	10503
q6	4	47
q7	3	6600
q8	3	963
q9	7	13177

(a) Benchmark One Query Set

property	# triples	# values subject	# values object
voter	21600	21600	100
winner	216	216	2
hasBallot	21600	216	21600
option	21600	21600	3
title	216	216	216

(b) Cardinality and number of values govtrack.us 2004

Fig. 5. Experiment Configuration Set-Up

6.1 Predictive Capability of the Hybrid Cost Model

We studied the predictive capability of the OneQL hybrid cost model. We generated 190 different bushy tree plans for a query with four patterns (Figure 5(a)), and computed the estimated evaluation time using the OneQL hybrid cost model. Additionally, we executed the 190 plans in the OneQL query engine and measured the evaluation time in terms of the total number of inferences. Figure 6 plots the actual versus the estimated evaluation costs; we can observe a positive trend between the estimated and actual cost, and a correlation between both costs of 0.76. Both results indicate that there is a linear relation between the estimated and the actual costs, and they suggest that the OneQL hybrid cost model is able to predict the runtime performance of the OneQL query engine.

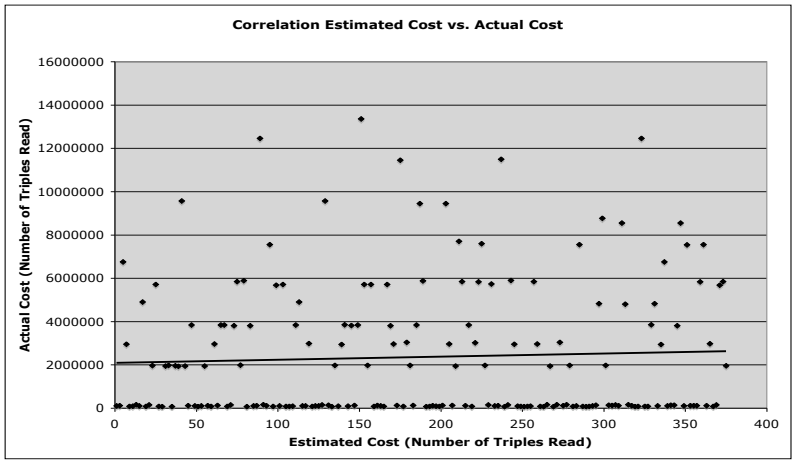


Fig. 6. Correlation actual cost vs. estimated cost

6.2 Effectiveness of the OneQL Optimization Techniques

We studied the effectiveness of the OneQL optimization techniques by empirically analyzing the quality of the optimized plans w.r.t. the rest of the plans of the corresponding query, and the runtime performance of the optimized plans.

To analyze the quality of the optimized plans, we generated all the plans for queries in benchmark one with three and four patterns, and computed the percentile in which the optimal plan falls. The average percentile is 97 and the lowest is 92. These results

indicate that the optimizer is able to identify execution plans that are at least better than 92% of the execution plans of the query.

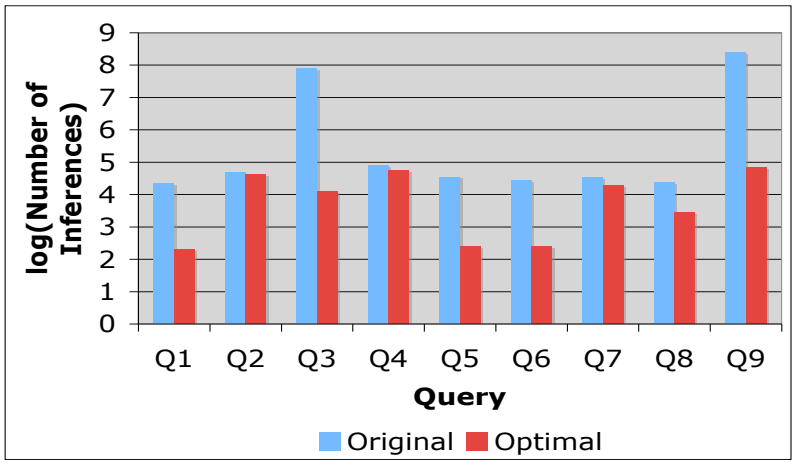


Fig. 7. Cost of Original vs. Optimal Queries (Inferences-logarithmic scale)

We also report on the runtime performance of the optimized queries. Figure 7 compares the number of inferred triples of the non-optimized and optimized versions of benchmark one in logarithmic scale. In general, we can observe that the optimized query has a significantly lower cost than the original query, speeding up the evaluation time in some cases by more than one order of magnitude. Plans with the most significant performance improvements correspond to bushy trees, and they are comprised of *Group* joins with small size groups.

6.3 Effectiveness of the OneQL Physical Operators

We have conducted an empirical analysis on the benefits of the evaluation techniques implemented on OneQL, and have executed 60 queries of benchmark two. Figure 8 compares the evaluation time (logarithmic scale) of the queries comprised of Group Joins (GJoin) against queries with Index Nested Loop Joins (Njoin). We can observe that the plans composed of GJoins overcome the Njoin plans by at least one order of magnitude when the GJoins are comprised of small size groups and low join selectivity.

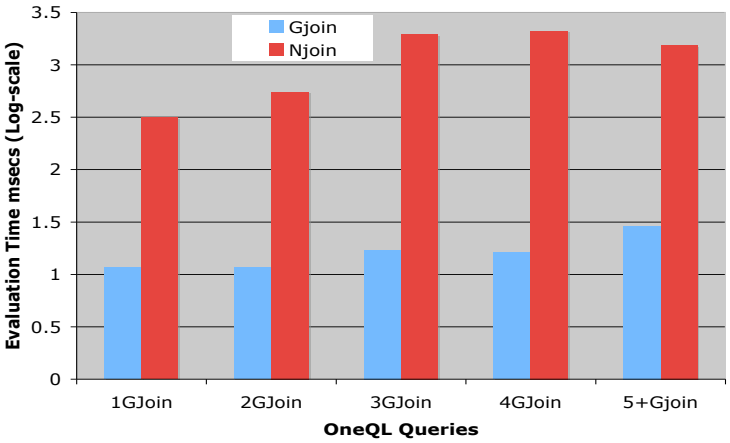


Fig. 8. Performance of the OneQL Physical Operators

6.4 Effectiveness of the OneQL Bhyper-based representation

Figure 9 compares the evaluation cost of twenty queries of benchmark two in logarithmic scale, when the *Bhyper* structures are used to index the RDF data, and when the *Bhyper* structures are not used to index the RDF data. The results indicate that the indices improve the performance of the physical operators, and the evaluation time is reduced by up to two orders of magnitude in queries comprised of a large number of GJoins composed of groups of instantiated triples.

6.5 Effectiveness of the OneQL Optimization and Evaluation Techniques

Finally, we studied the benefits of the optimization and evaluation techniques implemented by OneQL by empirically analyzing the quality of the OneQL optimized plans w.r.t. the plans optimized by the RDF-3X query optimizer. Queries of benchmark one were optimized by OneQL and RDF-3X and the generated plans were run in OneQL with and without *Bhyper* indices. Each RDF-3X optimized plan was run using the *GJoin* and *NJoin* operators to evaluate the groups in the bushy plans. Figure 10 reports the evaluation time (logarithmic scale) of these combinations of queries. We can observe that *Bhyper*-based representation is able to speed up the evaluation time of all the versions of the queries comprised of instantiated triples. Second, the evaluation time of the OneQL and RDF-3X optimized plans are competitive, except for queries *q1* and *q6* where OneQL was able to identify plans where all the triples are instantiated, and the most selective ones are evaluated first. These results indicate that the OneQL optimization and evaluation techniques may be used in conjunction with the state-of-the-art

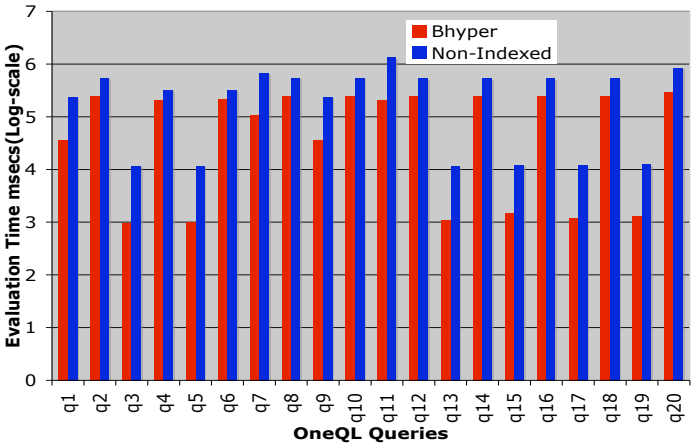


Fig. 9. Performance of the OneQL Bhyper-based index representation

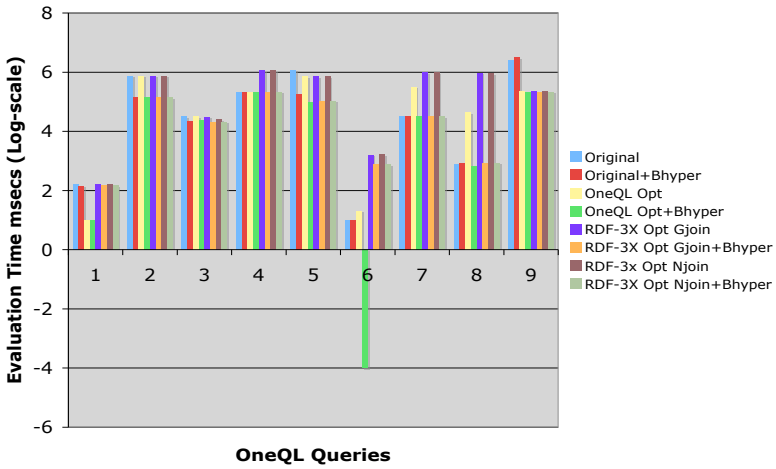


Fig. 10. Performance of the OneQL Optimization and Evaluation Techniques

techniques to provide more efficient query engines; they have encouraged us to develop our physical operators in existing RDF engines. So far, we have implemented the *GJoin* operator in the Jena engine, and we have observed in initial experiments that our *GJoin* implementation outperforms the evaluation time by up to three orders of magnitude. In the future, we also plan to implement these techniques in RDF-3X and conduct a more exhaustive empirical study to corroborate the effects of the developed techniques.

7 Conclusions

We have presented the OneQL system for efficiently evaluating SPARQL queries. We have addressed the challenges of scaling up to large RDF documents and complex SPARQL queries. We report on the results of our optimization and evaluation techniques for SPARQL queries. Then, we describe a *Bhyper*-based representation for RDF documents that reduces the space and time complexity of the tasks of storing and querying RDF documents. In the future, we plan to enhance the hybrid cost model with Bayesian inference capabilities to consider correlations between the different patterns that can appear in a SPARQL query; implement our operators in existing SPARQL query engines; and finally, extend the set of physical operators to better exploit the properties of the *Bhyper*-based representation.

8 Acknowledgments

This research has been partially supported by the DID-USB and the Proyecto ALMA Mater-OPSU. The authors are very grateful to Eduardo Ruiz for his programming support.

References

1. D. J. Abadi, A. M. 0002, S. Madden, and K. Hollenbach. SW-Store: a vertically partitioned DBMS for Semantic Web data management. *VLDB J.*, 18(2):385–406, 2009.
2. D. J. Abadi, A. M. 0002, S. Madden, and K. J. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *VLDB*, pages 411–422, 2007.
3. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Publishing Company, 1995.
4. AllegroGraph. http://www.franz.com/agraph/allegrograph/.
5. P. Auillans, P. O. de Mendez, P. Rosenstiehl, and B. Vatant. A Formal Model for Topic Maps. In *Proceedings of the Third International Semantic Web Conference (ISWC 2004)*, 2002.
6. G. Benson. Editorial. *Nucleic Acids Research*, 35(Web-Server-Issue):1, 2007.
7. F. Dau. RDF as Graph-Based, Diagrammatic Logic. In *Proceedings of the 16th International Symposium on Methodologies for Intelligent Systems (ISMIS 2006)*, 2006.
8. G. Fletcher and P. Beck. Scalable Indexing of RDF Graph for Efficient Join Processing. In *CIKM*, 2009.
9. G. Gallo, G. Longo, S. Pallottino, and S. V. Nguyen. Directed Hypergraphs and Applications. In *Discrete Applied Mathematics*, 2003.
10. G. Gallo and M. G. Scutella. Directed Hypergraphs as a Modelling Paradigm. In *Tech. Rep. TR-99-02, Universita di Pisa*, 1999.

11. M. Y. Galperin. The Molecular Biology Database Collection: 2007 update. *Nucleic Acids Res*, 35(Database issue), January 2007.
12. M. Y. Galperin. The Molecular Biology Database Collection: 2008 update. *Nucleic Acids Res*, 36(Database issue):D2–D4, Jan 2008.
13. A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *ISWC/ASWC*, pages 211–224, 2007.
14. G. Ianni, T. Krennwallner, A. Martello, and A. Polleres. A Rule System for Querying Persistent RDFS Data. In *Proceedings of the 6th European Semantic Web Conference (ESWC2009)*, Heraklion, Greece, May 2009. Springer. Demo Paper.
15. The JenaOntology Api. http://jena.sourceforge.net/ontology/index.html.
16. Jena TDB. http://jena.hpl.hp.com/wiki/TDB.
17. R. Lipton and J. Naughton. Query Size estimation by adaptive sampling (extended abstract). In *Proceedings of SIGMOD*, 1990.
18. A. Martinez and M. Vidal. A Directed Hypergraph Model for RDF. In *KWEPSY*, 2007.
19. J. McGlothlin and L. Khan. RDFJoin: A Scalable of Data Model for Persistence and Efficient Querying of RDF Dataasets. In *VLDB*, 2009.
20. T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.
21. R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.
22. E. Ruckhaus, E. Ruiz, and M. Vidal. Query Evaluation and Optimization in the Semantic Web. In *Proceedings ALPSWS2006: 2nd International Workshop on Applications of Logic Programming to the Semantic Web and Semantic Web Services*, 2006.
23. E. Ruckhaus, E. Ruiz, and M. Vidal. OnEQL: An Ontology Efficient Query Language Engine for the Semantic Web. In *Proceedings ALPSWS2007*, 2007.
24. E. Ruckhaus, E. Ruiz, and M. Vidal. Query Evaluation and Optimization in the Semantic Web. *TPLP*, 2008.
25. R. Sacks-Davis, T. D. J. A. Thom, and J. Zobel. Indexing documents for queries on structure, content, and attributes. In *Proceedings of the International Conference on Digital Media Information Bases*, 1997.
26. P. Selingerl, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. *Proceedings of ACM Sigmod*, 1979.
27. L. Sidirourgos, R. Goncalves, M. L. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. *PVLDB*, 1(2):1553–1563, 2008.
28. M. Stoker, A. Seaborne, A. Bernstein, C. Keifer, and D. Reynolds. SPARQL Basic Graph Pattern Optimizatin Using Selectivity Estimation. In *WWW*, 2008.
29. G. Terracina, N. Leone, V. Lio, and C. Panetta. Experimenting with recursive queries in database and logic programming systems. *Theory Pract. Log. Program.*, 8(2):129–165, 2008.
30. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.
31. J. Wielemaker. An Optimised Semantic Web Query Language Implementation in Prolog. In *ICLP*, pages 128–142, 2005.
32. K. Wilkinson, C. Sayers, H. Kuno, D. Reynolds, and J. Database. Efficient RDF Storage and Retrieval in Jena2. In *EXPLOITING HYPERLINKS 349*, pages 35–43, 2003.