SPARQL query answering with bitmap indexes

Julien Leblay

SPARQL query answering with bitmap indexes *

Julien Leblay
Inria Saclay & LRI, Université Paris-Sud
Bât. 650, 91405 Orsay Cedex, France
julien.leblay@inria.fr

ABSTRACT

When querying RDF data, one may use reasoning to reach intensional data, i.e., data defined by sets of rules. This is usually achieved through forward chaining, with space and maintenance overheads, or backward chaining, with high query evaluation and optimization costs. Recent approaches rely on pre-computing the *terminological closure* of the data rather than the full saturation. In this setting, one can even query the data without resorting to backward chaining, using a so-called *semantic index*. However, these techniques are limited in the type of queries they can support.
In this paper, we introduce a data storage technique which mitigates the space issues of forward-chaining. We show that it can also be used with a semantic index. We propose a new structure for the index that relies on bitmaps making it resilient to updates. Our experimental results demonstrate that our storage model significantly reduces the space required to store the data. We show that the indexes can be computed quickly and fit well in memory even for very large ontologies. Finally, we analyze how query answering is affected by the data layout.

Categories and Subject Descriptors

H.2.2 [**Database management**]: Physical design; H.2.4 [**Database management**]: Systems—*Query Processing*

General Terms

Languages, Performance, Experimentation

Keywords

SPARQL, storage model, query answering

1. INTRODUCTION

The RDF data model is now a well-established standard for representing data on the web. Its flexibility makes it both easy to understand and powerful enough to express and integrate complex data. An RDF dataset can be seen either (i) as a graph where nodes are resources or values and edges are relationships between them, or (ii) as sets of statements of the form ($subject$, $property$, $object$). A $subject$ takes its values from $\mathcal{U} \cup \mathcal{B}$, where $\mathcal{U}$ is the set of URIs and $\mathcal{B}$ is the set of blank nodes (i.e., resources of unknown URIs). A $property$ takes its values from $\mathcal{U}$ and an $object$ takes its values from $\mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$, where $\mathcal{L}$ is the set of all literals (i.e., constant values). SPARQL is the standard query language for accessing RDF data. The evaluation of a SPARQL query against a graph is defined through graph matching. At the core of a SPARQL query is the notion of triple pattern, a triple made of constants and variables. In this work, we focus on the conjunctive fragment of SPARQL, i.e., queries containing a conjunction of triple patterns, possibly with shared variables to express joins (BGP queries). For instance, the following query retrieves the names of artists, by matching the first triple pattern with all statements with a "$:name$" property, the second triple pattern with all statements typing the subject as an "$:Artist$", and joining them by their subjects:

```
SELECT ?x ?y
WHERE { ?x :name ?y. ?x rdf:type :Artist}
```

The upcoming SPARQL 1.1 specification defines the notion of entailment regimes, whereby the DBMS may override the semantics of the triple pattern matching, typically to retrieve triples that are entailed by a set of rules and the extensional data. In the sequel, we concentrate on the RDFS entailment rules [11], and queries on the assertional facts, i.e., facts that are not part of the terminology.

Finding all the triples entailed w.r.t. a dataset and a set of rules is generally done through forward or backward chaining. The former works in an offline fashion, the latter is used upon query evaluation. Forward chaining consists in applying the rules on the data instance until a fixpoint is reached and materializing the derived triples along with the extensional data. The *closure* of the data instance contains both extensional and intensional data. When evaluating queries on the data closure, no entailment-specific processing needs to take place. However, materializing the closure comes at a price. First, it requires additional storage space. As we show in Section 4, commonly used ontologies can lead to significant space overheads. Second, if the data is subject to updates, maintaining consistency can be expensive. In the worst case, i.e., when the schema itself is affected by a change, one may have to recompute the closure of the entire dataset.

Backward chaining approaches do not require any pre-processing of the data. Instead, rules are used at evaluation time to rewrite an incoming conjunctive query into a union of conjunctive queries. As rewritings grow exponentially in the size of queries, they are hard to optimize and evaluate in practice.

Recent works [8, 9] have shown there are other viable options between these approaches. In this paper, we present a storage model

*This work has been partially funded by ANR-11-EITS-003 and the EIT ICT Labs "DataBridges" activity.

SWIM 2012, May 20, 2012, Scottsdale, Arizona, USA.

that can be used in conjunction with a forward chaining approach or with a variant of semantic indexes introduced in [8]. The basic idea is to store as *sets* all classes a resource belongs to, and all properties that exist between a pair of RDF nodes. These sets are stored in bitmap indexes. Single *synthetic* facts that represent all classes of a resource (resp. all properties between two nodes) are stored in a relational table. When a query is evaluated, the bitmap indexes are consulted to modify an execution plan such that it remains optimizable with off-the-shelf algorithms. We redefine semantic indexes as bitmap indexes and explain how to further modify an execution plan to rely on them, e.g., in a context where using forward chaining is not an option.

In Section 2, we detail how data is organized under our model. In Section 3, we discuss how semantic indexes can be adapted to avoid storing the closure of a data instance. Section 4 shows how our technique affects storage space and query evaluation time for commonly used datasets. Section 5 covers the related works. We conclude this paper by discussing future work in Section 6.

2. GENERAL APPROACH

2.1 Data storage

Several storage models have been proposed for RDF [1, 6, 7], the most popular one being the *triple table*. A triple table is a relational table T with three attributes s, p and o (for $subject$, $property$ and $object$) containing a record for each fact of the data instance. Facts are often stored as triples of integers, where each integer is a key to a dictionary table containing all distinct strings of the instance. *Dictionary encoding* is a simple yet effective way to reduce the amount of space consumed by the data. RDF facts fall into two categories: (i) statements of the form *(:Alice rdf:type :Person)*, which assign a type to a resource, (ii) statements of the form *(:Alice :knows :Bob)*, which define a relationship between two resources, or between a resource and a value. In practice, the triple table can be partitioned along these two categories to reduce the time required to retrieve triples of either type, but for simplicity, we will only refer to T hereafter.

Facts are atomic in nature, therefore, a resource "$:Alice$" belonging to n classes will be stored as n records. Similarly, if resources "$:Alice$" and "$:Bob$" are linked with m properties, there will be m records to represent those facts. This type of redundancy is relatively common in real-world datasets and the process of materializing the closure of the data typically makes it worse. However, it is possible to mitigate this by storing a single *synthetic* fact all classes a resource belongs to (resp. all property between two nodes). Let $\mathcal{D}$ be an RDF data instance containing extensional *and* intensional facts w.r.t. to RDFS entailment rules.

DEFINITION 2.1 (CONCISE CLASS). *Let C be an ordered set of all classes in $\mathcal{D}$, and x a resource in $\mathcal{D}$. $C[i]$ denotes the class at position i in C. The* concise class *of x is a bitmap where every bit at position i is set to 1 if (x rdf:type $C[i]$) $\in \mathcal{D}$ and to 0 otherwise.*

DEFINITION 2.2 (CONCISE PROPERTY). *Let P be an ordered set of all properties in $\mathcal{D}$, x a resource in $\mathcal{D}$ and y a resource or a literal in $\mathcal{D}$. $P[i]$ denotes the property at position i in P. The* concise property *of the pair (x, y) is a bitmap where every bit at position i is set to 1 if (x $P[i]$ y) $\in \mathcal{D}$ and to 0 otherwise.*

Note that the orders of C and P are arbitrary and fixed over time. We compute concise classes for all resources in $\mathcal{D}$ and concise properties for all pairs of nodes in $\mathcal{D}$, and assign a unique ID number to each distinct concise class and property. We define a special concise property representing the set {rdf:type} to which we assign the ID 0. We store concise classes (resp. concise properties) and their IDs in a bitmap index, called CLIDX (resp. PRIDX). Henceforth, we will write CLIDX[i] (resp. PRIDX[i]) to denote the concise class (resp. property) obtained by looking up CLIDX (resp. PRIDX) for the ID i. Finally, we create a dictionary-encoded triple table containing for each resource r with a non-empty concise class c, the triple ($Key(r)$, 0, $Id(c)$), where $Key(r)$ is the key of r in the dictionary and $Id(c)$ is the ID of c in CLIDX. Similarly, for each pair of nodes (x, y) such that there exists a non-empty concise property p between x and y, we store the triple (x, $Id(p)$, y), where $Id(p)$ is the ID of p in PRIDX. We call such records *concise facts*.

Advantages of using bitmaps. Each concise class (resp. concise property) represents a subset of C (resp. P). Bitmaps are commonly used in databases for representing subsets as they can be efficiently compressed and set operations translate to simple bitwise operations. Many compression schemes are now available, including schemes that allow for bitwise operations without decompression [3, 4]. The best compression ratios are usually obtained on sparse or dense bitmaps, as long chains of zeros or ones can be drastically summarized. The sizes of C and P are generally small compared with the size of the data instance. Moreover, concise classes and properties represent small subsets of C and P, making for sparse bitmaps that easily fit in memory once compressed, even when one considers the closure of the data instance. Since trailing zeros are not stored at all, compression rates can be further improved by ordering C and P by decreasing frequencies of classes and properties in the instance.

2.2 Query answering

In a relational setting, a typical execution plan for a conjunctive query is a tree made of selection (σ), projection (π) and join ($\bowtie$) operators. The leaves of the tree are scans over the tables. Evaluating a conjunctive query over the triple table works in a similar fashion where each leaf is a scan over the same relation T and corresponds to a triple pattern in the query. For example, the query given in Section 1 can be translated into a plan such as:

$$\pi_{1,6}(\sigma_{p=\text{rdf:type}\wedge o=:Artist}(T) \bowtie \sigma_{p=:name}(T))$$

We now define a new scan operator, called T', that traverses the triple table and reconstructs the actual facts from the concise classes and properties read. For each record t of the form (s, p, o) in the triple table:

1. If $p = 0$, i.e., t is a typing statement, then for each class $k \in$ CLIDX[o], the triple (s, rdf:type, k) is returned instead of t.
2. If $p \neq 0$, then for each property $k \in$ PRIDX[p], the triple (s, k, o) is returned instead of t.

Observe that executing T' over a concise triple table produces the same result as the conventional scan T over a plain triple table.

Next, we explain how to produce a plan to execute over the concise triple table, by modifying a trivial plan as it would have been created on a triple table. Let q be a conjunctive SPARQL query and q' a plan for evaluating q, where each scan T in q' correspond to a triple patterns in q. For each triple pattern in q, we make the following modifications to q', depending of the pattern type:

1. For a pattern (?x ?y ?z) or (?x rdf:type ?z), the corresponding scan operator T in q' is replaced with T'.
2. For a pattern (?x k ?z), where $k \neq$ rdf:type, there is a selection operator in q' featuring the predicate $p = k$.
   This predicate is replaced by $p \in S$, where S is the set of IDs of concise properties to which the property k belongs, formally $S = \{Id(i) \mid \exists i \text{ PRIDX}[i] \cap \{k\} \neq \emptyset\}$.

3. For a pattern (?x rdf:type k), selection operators with predicates $p = \text{rdf:type}$ and $o = k$ are present in q'.
   The former is replaced by $p = 0$, the latter by $o \in S$, where S contains IDs of concise classes to which the class k belongs, more formally $S = \{Id(i) \mid \exists i \, \text{CLIDX}[i] \cap \{k\} \neq \emptyset\}$.

All other patterns, e.g., patterns where the $subject$ is bound, are sub-cases of the above and thus, processed in a similar way. The construction of S is achieved by a single traversal of CLIDX or PRIDX. The intersection operation translates to a bitwise AND between compressed bitmaps. Overall, these sets are inexpensive to build and they can easily be cached.

3. SEMANTIC INDEX-BASED APPROACH

3.1 Data storage

A semantic index, as presented in [8], is a table encoding the hierarchies in an ontology using a DAG labeling scheme. At query evaluation time, the index is used to rewrite each atom to a SQL range query retrieving all the classes (or properties) that are subsumed by a given class (or property).

In our setting, semantic indexes are redefined as bitmap indexes. The procedure to build the concise triple table, CLIDX and PRIDX is the same as the one described in Section 2 except that $\mathcal{D}$ now only contains extensional facts. The semantic indexes are built by computing the *terminological closure*, i.e., the closure on the facts that belong to the schema. More precisely, we compute for each class, the sets of its sub-classes, its super-classes, the properties of which the class is in the domain and the properties of which it is in the range. The four sets associated with each class are encoded into bitmaps using the procedure described in Section 2.

We store these sets in four new indexes, named SUBCL, SUPCL, DOMCL and RNGCL. We proceed similarly for each property to obtain the sets of its sub-properties, super-properties, domains and ranges, which we store in indexes SUBPR, SUPPR, DOMPR and RNGPR. We use the same notation as for CLIDX and PRIDX to refer to the sets contained in these indexes. For example, SUBCL[c] is the set of sub-classes of c.

3.2 Query answering

First, we define two new scan operators T'' and T_c, where c is a constant. T'' works as follows. For each triple t of the form (s, p, o) in the triple table:

1. If $p = 0$, i.e., t is a typing statement, then for each $k \in S$ where $S = \bigcup_{i \in \text{CLIDX}[o]} \text{SUPCL}[i]$, the triple ($s$, rdf:type, k) is returned.
   S is the union of super-classes of the classes encoded in o.
2. If $p \neq 0$, then
   - for each $k \in S_1$, where $S_1 = \bigcup_{i \in \text{PRIDX[p]}} \text{SUPPR}[i]$, the triple ($s$, k, o) is returned,
   - for each $k \in S_2$, where $S_2 = \bigcup_{i \in \text{PRIDX[p]}} \text{DOMPR}[i]$, the triple ($s$, rdf:type, k) is returned,
   - for each $k \in S_3$, where $S_3 = \bigcup_{i \in \text{PRIDX[p]}} \text{RNGPR}[i]$, the triple ($o$, rdf:type, k) is returned.

The scan operator T_c behaves essentially like T with the following exceptions. For each triple t of the form (s, p, o):

- if $p = 0 \wedge o \in \text{SUBCL}[c]$, it returns the triple (s, rdf:type, c),
- if $p \in \text{DOMCL}[c]$, it returns the triple (s, rdf:type, c),
- if $p \in \text{RNGCL}[c]$, it returns the triple (o, rdf:type, c),
- otherwise, t is ignored.

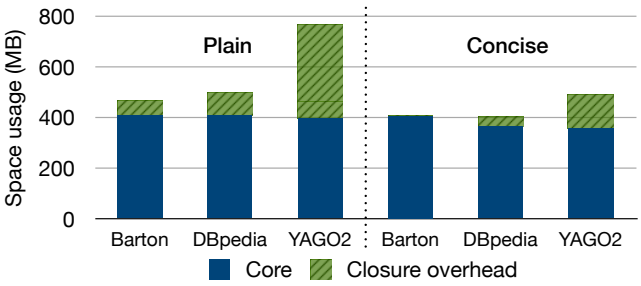

Figure 1: Space usage (in MB) of the triple table for 3 datasets with plain facts (left) and concise facts (right)

Both T'' and T_c may produce duplicates. These can be avoided using simple caching techniques.

Next, as we did in Section 2.2, we explain how to modify a conventional plan to evaluate a query in the presence of semantic indexes. Let q be conjunctive SPARQL query and q'' a plan for evaluating q on a plain triple table, where each scan T in q'' corresponds to a triple pattern in q. For each triple pattern in q, we modify q'', as follows:

1. For a pattern (?x ?y ?z) or (?x rdf:type ?z), the corresponding scan operator T is replaced with T''.
2. For a pattern (?x k ?z), where $k \neq$ rdf:type, there is a selection operator in q'' featuring the predicate $p = k$.
   The predicate is replaced by $p \in S$, where S contains the IDs of concise properties to which a sub-property of k belongs, i.e., $S = \{Id(i) \mid \exists i \, \text{PRIDX}[i] \cap \text{SUBPR}[k] \neq \emptyset\}$.
3. For a pattern (?x rdf:type k), the two selection predicates $p = \text{rdf:type}$ and $o = k$ must be removed from q'' and the operator T associated with the pattern is replaced with T_k.

4. IMPLEMENTATION & EXPERIMENTS

We implemented the storage model and the plan operators in Java 1.6.0_29 (64bits). All our tests were performed on a single machine with 8 Intel Xeon CPUs running at 2.13GHz with 4096 KB of cache each. We allocated 2GB of RAM to the virtual machine.

4.1 Tables and indexes space requirement

We first compare the space required to store the triple table with three widely used datasets: Barton, DBpedia and YAGO2. We chose these datasets for two main reasons. First, they come with very different schemas. The terminological closure of the schema used with Barton counts 201 statements, while the schemas of DBpedia and YAGO2 contain 7,745 and 3,983,638 statements respectively. Second, after cleaning the data and computing the *core*, i.e., the set of facts that cannot be derived through entailments, each dataset featured approximately 33M facts. This allows to compared how materializing the closure affects space usage. Figure 1 shows the amount of space (in MB) occupied by the plain triple table (left), against the concise triple table along with CLIDX and PRIDX (right)[1]. The blue (dark solid) bars represent the core data, while the overhead incurred by the materialization is shown in (light hatched) green. The size of the schema has a clear impact on storage space with a conventional approach. For instance, YAGO2 almost doubles in size as a result of materialization, jumping from 33M to 64M triples. On the other hand, the concise approach reduces this overhead to nearly 33% for YAGO2. The core also occupies less space with a concise approach because groups of facts are summarized into single ones. The space occupied by the closure of

[1]The dictionary tables are not included on the figure.

	Schema size	Space (KB)	Time (sec.)
Barton	201	17	0.07
Wordnet	524	11	9.56
Uniprot	1,283	39	0.13
Dbpedia	7,745	329	0.54
Sumo	52,909	1,505	7.11
Galen	217,724	2,565	2.18
Gene Onto.	532,118	4,465	6.91
YAGO 2	3,983,638	44,868	72.82

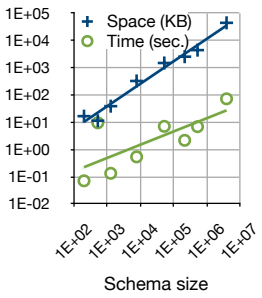

Figure 2: Time (in seconds) and space (in KB) required to build and store indexes for ontologies of various sizes in memory.

DBpedia is thus comparable to that occupied by the core dataset in a conventional triple store.

4.2 Semantic indexes construction

Next, we take a look at the space and time required to build the bitmap semantic indexes. For this, we used a set of freely available ontologies of various sizes. The table in Figure 2 shows their sizes (first column), as the number of triples in the terminological closure. The second and third columns detail the total space occupied by the compressed indexes in memory (in KB) and the time to compute them (in seconds), respectively. The graph on the right displays the same figures on a scatter plot. Notice the log scale on both axes. The data clearly shows that the time and space requirements grow sub-linearly in the size of the schema and remain very reasonable for nowadays systems. YAGO2, the largest schema, loads in 72 seconds and fits in 45MB of memory only.

4.3 Impact on query evaluation

To assess how query evaluation performs in this model, we ran four queries on the three datasets mentioned before. Each query comprised one of the triple patterns described in Sections 2.2 and 3.2. Q_1 and Q_2 are of the form (?x ?y ?z) and (?x rdf:type ?z) respectively. Q_3 corresponds to (?x rdf:type k), and Q_4 to (?x k ?y), where k is the class (resp. property) that belongs to the most concise classes (resp. properties), i.e., the most adverse cases for evaluation. Figure 3 reports evalution times (in seconds) averaged over 5 runs. Plain refers to the evaluation on a conventional triple table, Concise-1 refers to the concise triple table with materialized closure, Concise-2 to the semantic index approach. Tables were partitioned across typing and relationship statements. For most queries and datasets, Concise-1 performs better than a conventional approach, as the size of the scanned tables dominates the processing cost of the new operators overall. With Concise-2, Q_2 and Q_3 are penalized as they need to scan both partitions to collect all possible results. However, this is amortized for Q_3 with YAGO2 as the plain triple table becomes prohibitively expensive to scan. We plan to explore optimizations in the presence of B+tree indexes and with parallelization in future work.

5. RELATED WORKS

In [9], Urbani et al. observed that by computing the *terminological closure*, one can reduce the backward-chaining phase at query evaluation time. *Semantic indexes*, presented in [8], go a step further by storing the terminological closure in indexes and the assertional facts in relational tables. The indexes map each class (resp. property) to pairs of integers obtained through a traversal of the schema hierarchies. Rewritings of all possible triple patterns to SQL range queries are cached, by-passing the need for backward-chaining. This method is the closest to ours.

Bitmaps have been used in other storage models. In [2], they are used to store an RDF dataset as a cube and process joins with the objective of avoiding large intermediate results. Zou at al. [12] used bitmaps in gStore to model an in-memory graph database. Fernández et al. [5] rely on bitmaps to compress RDF for data exchange over networks. In [10], Orri Erling explains how the commercial system Virtuoso makes uses of bitmap indexes to speed up triples loading time, and reduce space usage.

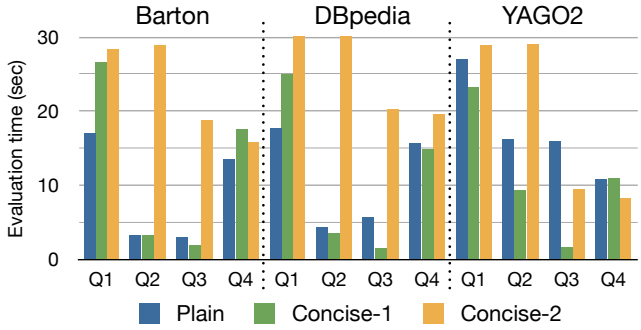

Figure 3: Query run times (in seconds) for 4 simple queries.

6. CONCLUSION & OUTLOOK

In this paper, we introduced a storage model that (i) reduces the storage space required for an RDF data instance, (ii) suffers little space overhead when the closure is materialized in the instance, (iii) improves query evaluation time in the general case, (iv) can be used in conjunction with semantic indexes when forward-chaining must be avoided. Although we presented the model as a variant of a triple table setting, it could also be applied to other storage types such as property tables [6] or RDF cubes [2].

Our next focus will be on query optimization, e.g., in the presence of indexes, as described in [7]. The method will be extended to support other entailment regimes such as the OWL 2 EL Profile and queries on the terminology itself. We think our approach would be particularly well suited to answer queries on RDF data streams and for analytical query processing.

7. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.

[2] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix Bit loaded: a scalable lightweight join query processor for RDF data. In *WWW*, 2010.

[3] A. Colantonio and R. Di Pietro. Concise: Compressed 'n' composable integer set. *Information Processing Letter*, 2010.

[4] F. Deliège and T. B. Pedersen. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In *EDBT*, 2010.

[5] J. Fernández, M. Martínez-Prieto, and C. Gutierrez. Compact representation of large RDF data sets for publishing and exchange. In *ISWC*. 2010.

[6] The Jena Framework. At http://jena.sourceforge.net/.

[7] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *Proc. of VLDB*, 2008.

[8] M. Rodríguez-Muro and D. Calvanese. Semantic index: Scalable query answering without forward chaining or exponential rewritings. In *ISWC*, 2011.

[9] J. Urbani, F. van Harmelen, S. Schlobach, and H. Bal. QueryPIE: Backward reasoning for OWL Horst over very large knowledge bases. In *ISWC*. 2011.

[10] Advances in Virtuoso RDF triple storage.

[11] RDF Semantics. At http://www.w3.org/TR/rdf-mt/, 2004.

[12] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: answering SPARQL queries via subgraph matching. *Proc. of VLDB*, 2011.