Toward Efficient Techniques for Completeness Reasoning

Fariz Darari[1*], Werner Nutt[1], Giuseppe Pirrò[2], and Simon Razniewski[1]

[1]Free University of Bolzano, Italy
[2]ICAR-CNR, Italy
[*]`fariz.darari@stud-inf.unibz.it`

Abstract. The Semantic Web is commonly interpreted under the open-world assumption: The present information captures only a subset of the reality. A consequence of this assumption is that users can never be sure whether the present information fully describes the reality or not, which decreases the value one can derive from querying the Semantic Web. Still, one can observe that the Semantic Web contains complete information for many aspects of reality. From previous work, a theoretical framework allowing one to augment RDF data sources with information about their completeness, was developed. Such information can be used to check whether the answer returned by a query is complete. Yet, it is still unclear how such a check can be done in practice. We devise implementation techniques to make completeness reasoning in the presence of large sets of completeness statements feasible, and evaluate their effectiveness.

Keywords: RDF, SPARQL, Completeness, Reasoning, Indexing

1 Introduction

The increasing amount of structured data made available on the Web is laying the foundation of a global-scale knowledge base. Projects like Linked Open Data (LOD) [8], by inheriting some basic design principles of the Web (e.g, simplicity, decentralization), aim at making huge volumes of data available by the Resource Description Framework (RDF) standard data format [13]. RDF enables one to make *statements* about resources in the form of triples, consisting of a *subject*, a *predicate*, and an *object*. Ontology languages such as RDF Schema (RDFS) [4] and OWL [10] provide the necessary underpinning for the creation of vocabularies to structure knowledge domains. The common path to access such a huge amount of structured data is via SPARQL endpoints, namely, network locations that can be queried upon by using the SPARQL query language [7].

With a large number of RDF data sources covering possibly overlapping knowledge domains, it is natural to observe a wide range of data source quality; indeed, some data sources are manually-curated while others result from crowdsourcing efforts or automatic extraction techniques [11,3]. In this setting, the problem of providing high-level descriptions (in the form of metadata) of their content becomes crucial. Such descriptions will connect data publishers and consumers; publishers will advertise “what” is there inside a data source so that specialized applications can be created for data source discovering, cataloging, selection, and so forth. Proposals like the `VoID` vocabulary [1] touch this aspect. With `VoID` it is possible, among the other things, to provide information about how many instances a particular *class* has, the SPARQL endpoint of a source, and links to other data sources. However, `VoID` mainly focuses on providing *quantitative* information. We claim that toward comprehensive descriptions of data sources, also *qualitative* information is crucial; in particular about completeness.

In previous work, Darari et al. [6] proposed a framework for managing completeness over RDF data sources and introduced the notions of completeness statements to describe complete parts of data sources, and of query completeness. Moreover, they investigated the problem of completeness entailment, namely, the check whether a set of completeness statements entails query completeness. Nevertheless, we expand upon this work by developing indexing techniques for completeness statements that can reduce the number of statements considered in the reasoning. In addition to that, we provide an experimental evaluation to show that our optimization can considerably reduce the running time of reasoning with large sets of completeness statements, and make it comparable to that of query evaluation.

2 Formal Framework

In the following, we remind the reader of RDF and SPARQL, and formalize the basic notions of the completeness management framework as in [6].

RDF and SPARQL. We assume there are three[1] pairwise disjoint infinite sets I (*IRIs*), L (*literals*), and V (*variables*). We collectively refer to IRIs and literals as *RDF terms* or simply *terms*. A tuple $(s, p, o) \in I \times I \times (I \cup L)$ is called an *RDF triple* (or a *triple*), where s is the *subject*, p the *predicate* and o the *object* of the triple. An *RDF graph* or *data source* consists of a finite set of triples [13]. For simplicity, we omit namespaces for the abstract representation of RDF graphs.

The standard query language for RDF is SPARQL. The basic building blocks of a SPARQL query are *triple patterns,* which resemble RDF triples, except that in each position also variables are allowed. SPARQL queries include graph patterns, built using the `AND` operator, and more sophisticated operators, including `OPT` (for “optional”), `FILTER`, `UNION`, and so forth. In this article, we consider the operators `AND` and `OPT`. Graph patterns with only the `AND` operator are called *basic graph patterns* (BGPs). Alternatively, one may use a set of triple patterns to represent a BGP. A *mapping* μ from variables to terms is defined as a partial function $\mu\colon V \rightarrow I \cup L$. Given a BGP P, the expression μP returns a graph where all the variables in P are replaced with terms according to μ. Evaluating a graph pattern P over an RDF graph G results in a set of mappings from the variables in P to terms, denoted as $[\![P]\!]_G$.

SPARQL queries come as `SELECT`, `ASK`, or `CONSTRUCT` queries. A `SELECT` query has the abstract form (W, P), where P is a graph pattern and W is a subset of the variables in P. A `SELECT` query $Q = (W, P)$ is evaluated over a graph G by projecting the mappings in $[\![P]\!]_G$ to the variables in W, written as $[\![Q]\!]_G = \pi_W([\![P]\!]_G)$. Syntactically, an `ASK` query is a special case of a `SELECT` query where W is empty. For an `ASK` query Q, we write

[1] In this work, we do not consider blank nodes.

also $[\![Q]\!]_G = \texttt{true}$ if $[\![Q]\!]_G \neq \emptyset$, and $[\![Q]\!]_G = \texttt{false}$ otherwise. A `CONSTRUCT` query has the abstract form (P_1, P_2), where P_1 is a BGP and P_2 is a graph pattern. In this article, we only use `CONSTRUCT` queries where also P_2 is a BGP. The result of evaluating $Q = (P_1, P_2)$ over G is the graph $[\![Q]\!]_G$, that is obtained by instantiating the pattern P_1 with all the mappings in $[\![P_2]\!]_G$. We focus here on the class of *basic queries*: queries (W, P) where P is a BGP and which return *bags* of mappings (as it is the default in SPARQL). Further information about SPARQL can be found in [16].

2.1 Completeness Statements

To tackle the problem of completeness management in RDF data sources, we proceed in two steps: *(i)* we formalize mechanism allowing one to specify which parts of a data source are complete; *(ii)* we devise techniques to check when a query is complete over a potentially incomplete data source.

When talking about the completeness of a data source, one implicitly compares the information *available* in the source with what holds in the world and therefore should *ideally* be also present in the source.

Definition 1 (Incomplete Data Source). *We identify data sources with RDF graphs. Then, adapting a notion introduced by Motro [15], we define an* incomplete data source *as a pair $\mathcal{G} = (G^a, G^i)$ of two graphs, where $G^a \subseteq G^i$. We call G^a the* available *graph and G^i the* ideal *graph.*

Example 1. Consider the DBpedia data source and suppose that the only movies directed by Tarantino are Reservoir Dogs, Pulp Fiction, and Kill Bill, and that Tarantino was starred exactly in the movies Desperado, Reservoir Dogs, and Pulp Fiction. For the sake of example, suppose also the fact that he was starred in Reservoir Dogs is missing in DBpedia.[2] Using Definition 1, we can formalize the incompleteness of the DBpedia data source $\mathcal{G}_{dbp}$ as follows:

$$\begin{aligned} G^a_{dbp} = \{ &(\mathit{reservoirDogs}, \mathit{director}, \mathit{tarantino}), (\mathit{pulpFiction}, \mathit{director}, \mathit{tarantino}), \\ &(\mathit{killBill}, \mathit{director}, \mathit{tarantino}), (\mathit{desperado}, \mathit{actor}, \mathit{tarantino}), \\ &(\mathit{pulpFiction}, \mathit{actor}, \mathit{tarantino}), (\mathit{desperado}, a, \mathit{Movie}), \\ &(\mathit{reservoirDogs}, a, \mathit{Movie}), (\mathit{pulpFiction}, a, \mathit{Movie}), (\mathit{killBill}, a, \mathit{Movie})\} \\ G^i_{dbp} = G^a_{dbp} &\cup \{\, (\mathit{reservoirDogs}, \mathit{actor}, \mathit{tarantino}) \,\}. \end{aligned}$$

We now introduce *completeness statements*, which are used to describe the parts of a data source that are complete, that is, the parts for which the ideal and available graph coincide.

Definition 2 (Completeness Statement). *A* completeness statement $\mathit{Compl}(P_1 \mid P_2)$ *consists of a non-empty BGP P_1 and a BGP P_2. We call P_1 the* pattern *and P_2 the* condition *of the completeness statement.*

[2] as it was the case on 20 June 2015

For example, we express that a source is complete for all pairs of triples that say "*?m is a movie and ?m is directed by Tarantino*" using the statement

$$C_{dir} = Compl((?m, a, Movie), (?m, director, tarantino) \mid \emptyset), \tag{1}$$

whose pattern matches all such pairs and whose condition is empty. To express that a source is complete for all triples about actors in movies directed by Tarantino, we use

$$C_{act} = Compl((?m, actor, ?a) \mid (?m, director, tarantino), (?m, a, Movie)), \tag{2}$$

whose pattern matches triples about actors and whose condition restricts the actors to those of movies directed by Tarantino. The condition in C_{act} does not imply that the data source contains triples of the form $(?m, director, tarantino)$ and $(?m, a, Movie)$. If we move the condition to the pattern, however, we impose that the data source contains the triples.

We now define when a completeness statement is satisfied by an incomplete data source. To a statement $C = Compl(P_1 \mid P_2)$, we associate the `CONSTRUCT` query $Q_C = (P_1, P_1 \cup P_2)$. Note that, given a graph G, the query Q_C returns a graph consisting of those instantiations of the pattern P_1 present in G for which also the condition P_2 can be satisfied. For example, the query $Q_{C_{act}}$ returns the cast of the Tarantino movies in graph G.

Definition 3 (Satisfaction of Completeness Statements). *An incomplete data source $\mathcal{G} = (G^a, G^i)$ satisfies the statement C, written $\mathcal{G} \models C$, if $[\![Q_C]\!]_{G^i} \subseteq G^a$ holds.*

The above definition naturally extends to the satisfaction of a set $\mathbf{C}$ of completeness statements, that is, $\mathcal{G} \models \mathbf{C}$ iff for all $C \in \mathbf{C}$, it is the case that $[\![Q_C]\!]_{G^i} \subseteq G^a$.

Intuitively, an incomplete data source (G^a, G^i) satisfies a completeness statement C, if the subgraph of G^i identified by C is also present in G^a. For instance, to see that $\mathcal{G}_{dbp}$ satisfies the statement C_{dir}, observe that the query $Q_{C_{dir}}$ returns over G^i_{dbp} all triples with the predicate *director* and all *a* (type) triples for Tarantino movies, and that all these triples are also in G^a_{dbp}. However, $\mathcal{G}_{dbp}$ does *not* satisfy C_{act}, because $Q_{C_{act}}$ returns over G^i_{dbp} the triple $(reservoirDogs, actor, tarantino)$, which is not in G^a_{dbp}.

An important tool in later characterizations of completeness entailment is the *transfer operator* $T_{\mathbf{C}}$, which maps graphs to graphs. Given a set $\mathbf{C}$ of completeness statements and a graph G, the operator is defined as

$$T_{\mathbf{C}}(G) = \bigcup_{C \in \mathbf{C}} [\![Q_C]\!]_G. \tag{3}$$

It takes the union of evaluating over G all corresponding `CONSTRUCT` queries of the statements in $\mathbf{C}$. Crucial properties of the transfer operator are summarized in the following proposition:

Proposition 1 (Properties of $T_{\mathbf{C}}$). *Let $\mathbf{C}$ be a set of completeness statements. Then,*

1. $(G^a, G^i) \models \mathbf{C}$ *iff* $T_{\mathbf{C}}(G^i) \subseteq G^a$.

Consequently, for any graph G we have that

(2) the pair $(T_{\mathbf{C}}(G), G)$ is an incomplete data source satisfying $\mathbf{C}$, and
(3) $T_{\mathbf{C}}(G)$ is the smallest available graph for which this holds.

2.2 Query Completeness

When querying a data source, we want to know whether the answer to our query is complete wrt. the real world. For instance, when querying DBpedia for movies starring Tarantino, it would be interesting to know whether we really get *all* such movies. We now formalize query completeness wrt. incomplete data sources.

Definition 4 (Query Completeness). *Let Q be a* SELECT *query. To express that Q* is complete*, we write $Compl(Q)$. An incomplete data source $\mathcal{G} = (G^a, G^i)$ satisfies $Compl(Q)$, if Q returns the same result over G^a as it does over G^i, that is, $[\![Q]\!]_{G^a} = [\![Q]\!]_{G^i}$. In this case we write $\mathcal{G} \models Compl(Q)$.*

In this work, we focus on classes of queries that are monotonic. Therefore, by definition it holds that $[\![Q]\!]_{G^a} \subseteq [\![Q]\!]_{G^i}$ for all incomplete data sources $\mathcal{G} = (G^a, G^i)$.

Example 2. Consider the incomplete data source $\mathcal{G}_{dbp}$ and the two queries Q_{dir}, asking for all movies directed by Tarantino, and $Q_{dir+act}$, asking for all movies both directed by and starring Tarantino:

$$\begin{aligned} Q_{dir} &= (\{\,?m\,\}, \{\,(?m, a, Movie), (?m, director, tarantino)\,\}) \\ Q_{dir+act} &= (\{\,?m\,\}, \{\,(?m, a, Movie), (?m, director, tarantino), (?m, actor, tarantino)\,\}). \end{aligned}$$

Then, it holds that Q_{dir} is complete over $\mathcal{G}_{dbp}$ since $[\![Q_{dir}]\!]_{G^a_{dbp}} = \{\,\{\,?m \mapsto reservoirDogs\,\}, \{\,?m \mapsto pulpFiction\,\}, \{\,?m \mapsto killBill\,\}\,\} = [\![Q_{dir}]\!]_{G^i_{dbp}}$. On the other hand, $Q_{dir+act}$ is *not* complete over $\mathcal{G}_{dbp}$ since $[\![Q_{dir+act}]\!]_{G^a_{dbp}}$ does not contain the mapping $\{\,?m \mapsto reservoirDogs\,\}$, which occurs in $[\![Q_{dir+act}]\!]_{G^i_{dbp}}$.

2.3 Completeness Entailment

Up to this point, we have provided examples with concrete incomplete data sources. In the following definition, we formalize the entailment of query completeness by completeness statements. This way, we 'quantify' over all incomplete data sources such that if a source satisfies the completeness statements, then it must also satisfy the query completeness.

Problem Definition. Let $\mathbf{C}$ be a set of completeness statements and Q be a SELECT query. We say that $\mathbf{C}$ *entails the completeness of Q*, written $\mathbf{C} \models Compl(Q)$, if any incomplete data source that satisfies $\mathbf{C}$ also satisfies $Compl(Q)$.

Example 3. Consider C_{dir} from Equation (1). Whenever an incomplete data source $\mathcal{G}$ satisfies C_{dir}, then G^a contains all triples about movies directed by Tarantino, which is exactly the information needed to answer query Q_{dir} from Example 2. Thus, $\{\,C_{dir}\,\} \models Compl(Q_{dir})$. However, this is not enough to completely answer $Q_{dir+act}$, thus $\{\,C_{dir}\,\} \not\models Compl(Q_{dir+act})$.

Characterizing Completeness Entailment. The query class we consider in this work is the class of queries with a conjunctive body. The standard semantics for such queries is bag semantics, which allows repetition of results. Generally, a basic query Q is complete wrt. a set $\mathbf{C}$ of completeness statements, if for every incomplete data source $\mathcal{G} = (G^a, G^i)$ satisfying $\mathbf{C}$, the query answers over G^i are contained in the query answers over G^a, where duplicates are taken into account. That is, a mapping occurring n times in $[\![Q]\!]_{G^i}$, must occur at least n times in $[\![Q]\!]_{G^a}$. Actually, since $G^a \subseteq G^i$, and since conjunctive queries are monotonic, we always have that the bag $[\![Q]\!]_{G^a}$ is contained in the bag $[\![Q]\!]_{G^i}$. Hence, Q is complete over $\mathcal{G}$ iff every mapping occurring n times in $[\![Q]\!]_{G^i}$ occurs also n times in $[\![Q]\!]_{G^a}$.

We want to provide a characterization of completeness entailment for basic queries. Let us give an example to provide an intuition of the characterization.

Example 4. Consider the set $\mathbf{C}_{dir,act}$ consisting of C_{dir} from Equation (1) and C_{act} from Equation (2). Recall the query $Q_{dir+act} = (\{\,?m\,\}, P_{dir+act})$, where

$$P_{dir+act} = \{\,(?m, a, Movie), (?m, director, tarantino), (?m, actor, tarantino)\,\}.$$

We want to check whether these statements entail the completeness of $Q_{dir+act}$, that is, whether $\mathbf{C}_{dir,act} \models Compl(Q_{dir+act})$ holds.

Suppose that $\mathcal{G} = (G^a, G^i)$ satisfies $\mathbf{C}_{dir,act}$. Suppose also that $Q_{dir+act}$ returns a mapping $\mu = \{\,?m \mapsto m'\,\}$ over G^i for some term m'. Then, G^i contains $\mu P_{dir+act}$, the instantiation by μ of the BGP of our query, consisting of the three triples $(m', a, Movie)$, $(m', director, tarantino)$, and $(m', actor, tarantino)$.

The `CONSTRUCT` query $Q_{C_{dir}}$, corresponding to our first completeness statement, returns over the graph $\mu P_{dir+act}$ the two triples $(m', a, Movie)$ and $(m', director, tarantino)$, while the `CONSTRUCT` query $Q_{C_{act}}$, corresponding to the second completeness statement, returns the triple $(m', actor, tarantino)$. Thus, all triples in $\mu P_{dir+act}$ have been reconstructed by $T_{\mathbf{C}_{dir,act}}$ from $\mu P_{dir+act}$.

Now, we have that

$$\mu P_{dir+act} = T_{\mathbf{C}_{dir,act}}(\mu P_{dir+act}) \subseteq T_{\mathbf{C}_{dir,act}}(G^i) \subseteq G^a,$$

where the last inclusion holds due to $\mathcal{G} \models \mathbf{C}_{dir,act}$. Therefore, our query $Q_{dir+act}$ returns the mapping μ also over G^a. Since μ and $\mathcal{G}$ were arbitrary, this shows that $\mathbf{C}_{dir,act} \models Compl(Q_{dir+act})$ holds.

In summary, in the above example we have reasoned about a set of completeness statements $\mathbf{C}$ and a basic query $Q = (W, P)$. We have considered a generic mapping μ, defined on the variables of P, and applied it to P, thus obtaining a graph μP. Then, we have verified that $\mu P = T_{\mathbf{C}}(\mu P)$. From this, we could conclude that for every incomplete data source $\mathcal{G} = (G^a, G^i)$ we have that $[\![Q]\!]_{G^a} = [\![Q]\!]_{G^i}$. Next, we make this approach formal.

Definition 5 (Prototypical Graph). *Let (W, P) be a query. The* freeze mapping $\tilde{id}$ *is defined as mapping each variable $?v$ in P to a new IRI $\tilde{v}$. Instantiating the graph pattern P with $\tilde{id}$ yields the RDF graph $\tilde{P} := \tilde{id}\,P$, which we call the* prototypical graph *of P.*

Now we can generalize the reasoning from above to a generic completeness check. To check whether the completeness of a query is entailed by a set of completeness statements, we evaluate all the corresponding `CONSTRUCT` queries of the statements over the prototypical graph $\tilde{P}$ and check whether over the evaluation result, we have $\tilde{P}$ back. Intuitively, this means that over any possible graph instantiation for answering the query, the completeness statements guarantee that we have back the graph instantiation in our data source.

Theorem 1 (Completeness of Basic Queries). *Let* $\mathbf{C}$ *be a set of completeness statements and* $Q = (W, P)$ *be a basic query. Then,*

$$\mathbf{C} \models Compl(Q) \qquad \textit{iff} \qquad \tilde{P} = T_{\mathbf{C}}(\tilde{P}).$$

The following complexity result follows as the completeness check is basically evaluating a linear number of `CONSTRUCT` queries over the conjunctive body of the query.

Corollary 1. *Deciding whether* $\mathbf{C} \models Compl(Q)$*, given a set* $\mathbf{C}$ *of completeness statements and a basic query* $Q = (W, P)$*, is NP-complete.*

The result shows that the complexity of completeness reasoning is not higher than that of conjunctive query evaluation, which is also NP-complete [5].

Up to this point, we are now able to formalize completeness statements on the Web. Using those statements, we can check whether a SPARQL query returns complete answers. In general, the technique to check query completeness incorporates the evaluation of the corresponding `CONSTRUCT` queries of the statements. Now, the question arises how much is the cost of the reasoning, in particular, compared to the cost of evaluating a query. Furthermore, in some applications there may be a large number of completeness statements, thus one may ask how feasible it is to perform completeness reasoning, and how to implement completeness reasoning in an efficient way.

3 Implementing Completeness Reasoning

In this section, we present the results of our investigation on efficient completeness reasoning over large sets of completeness statements. We first provide an overview of the problem with reasoning over large sets of statements, followed by a description of the predicate-relevance principle, which can be used to prune the set of statements considered in reasoning, and then a report on several retrieval techniques of predicate-relevant statements.

3.1 Problem Overview

Real-world RDF data sources may contain a large amount of data. For example, from the English Wikipedia, DBpedia extracted 580 million RDF triples.[3] Obviously, neither is all information from those triples complete, nor is its completeness interesting. If a

[3] `http://lists.w3.org/Archives/Public/public-lod/2014Sep/0028.html`

fifth of those triples were captured by completeness statements, each of which accounts for 100 triples, then there would be about 1 million completeness statements in total needed for DBpedia.

Now, the question is, how fast can we perform completeness reasoning with 1 million statements? Using a plain completeness reasoner that implements reasoning by evaluating the `CONSTRUCT` queries of *all* the completeness statements, we observed that reasoning may take 13 minutes, which is about 35,000 times slower than query evaluation (see Table 1).[4] Obviously, in practice this is not feasible as completeness reasoning would be performed as often as query evaluation. Indeed, the reason why a plain reasoner may take very long is that it takes into account all the completeness statements in the reasoning. However, not all statements will contribute to the entailment of query completeness. In fact, according to Theorem 1 that characterizes completeness entailment of basic queries, for a complete query with *n* triple patterns, there is a set of no more that *n* completeness statements that already entails the completeness of that query. From this observation, we might wonder if we could find a priori exactly those *n* completeness statements. However, as there is no obvious way to identify a priori such a set of at most *n* statements, in the worst case all statements have to be checked.

Table 1: Comparison of the running time median for plain completeness reasoning and query evaluation

Query Types	Plain Reasoning	Query Evaluation
Short	758,001 ms	20 ms
Long	781,286 ms	36 ms

Still, we observe that we can rule out a significant number of completeness statements that are clearly irrelevant to entailing query completeness, so that only the relevant ones are considered. Take an example where the query is "All movies directed by Tarantino" and the statement is "All football players of Arsenal." Obviously, there is no way the statement can guarantee the completeness of the query. We propose the *predicate-relevance principle* as a way to distinguish between irrelevant and relevant completeness statements. The principle states that a completeness statement can contribute to entailing query completeness only if all predicates of the completeness statement occur also in the query. We say that a statement satisfying this principle is *predicate-relevant*. We show later that the principle has a high selectivity, meaning it can rule out a considerable amount of completeness statements from being considered in completeness reasoning. As an illustration, among 1,000,000 randomly generated statements and a randomly generated query, on average only about 500 statements are predicate-relevant to the query.

[4] The results are from our experimental evaluation using randomly generated completeness statements and queries

The question is now, how can we efficiently retrieve those predicate-relevant statements? It turns out that the task of retrieving predicate-relevant statements is basically a task of subset querying, which has been well-studied. Therefore, we show later how we adopt existing index structures (i.e., inverted indexes and tries) for subset querying to retrieve predicate-relevant statements. In addition, we also develop a baseline approach based on standard hashing. We then conduct experimental evaluations to analyze the suitability of the different approaches in terms of retrieval time and scalability, and to compare the running times of plain completeness reasoning, completeness reasoning using the predicate-relevance principle, and query evaluation in Section 4.

3.2 Filtering Based on Predicate-Relevance

Let us estimate the complexity of the completeness reasoning task, from which we formulate the predicate-relevance principle.

Let $Q = (W, P)$ be a query and $\mathbf{C}$ be a set of completeness statements. According to Theorem 1, the task of completeness reasoning for basic queries is to check whether $T_{\mathbf{C}}(\tilde{P}) = \tilde{P}$, where $T_{\mathbf{C}}$ is the transfer operator wrt. $\mathbf{C}$, and $\tilde{P}$ is the prototypical graph of Q. While the '$\subseteq$' direction of the equality trivially holds, the interesting part is the '$\supseteq$' direction. It is the problem of finding for each triple $(s, p, o) \in \tilde{P}$ a completeness statement $C \in \mathbf{C}$ such that $(s, p, o) \in [\![Q_C]\!]_{\tilde{P}}$. Note that $T_{\mathbf{C}}(\tilde{P}) = \bigcup_{C \in \mathbf{C}} [\![Q_C]\!]_{\tilde{P}}$. This already tells us that we only find statements that *potentially* match such a triple (s, p, o).

Let $Q = (W, P)$ be a query, $\mathbf{C}$ be a set of completeness statements, and $maxLn(\mathbf{C})$ be the maximum length of statements in $\mathbf{C}$. Take any $C \in \mathbf{C}$. To evaluate the query Q_C over $\tilde{P}$, we have to consistently map the triple patterns of Q_C to triples in $\tilde{P}$. There are at most $|\tilde{P}|^{|Q_C|}$ possible ways to map triple patterns to triples, where $|Q_C|$ and $|\tilde{P}|$ stand for the number of triple patterns and triples in Q_C and $\tilde{P}$, respectively. If we do this for each statement in $\mathbf{C}$, the overall running time is $O(|\mathbf{C}||\tilde{P}|^{maxLn(\mathbf{C})})$. We are considering the case where the query Q is given and the set of completeness statements varies. It also seems reasonable to assume that the maximum length of completeness statements is a global constant, that is, all possible statements are at most of that length. Under this assumption, the cost of reasoning only depends on the size of the set of completeness statements. Therefore, we want to reduce the number of statements employed in completeness reasoning by considering only relevant ones.

Predicate-Relevance Criterion Although in general, predicates can be variables, in reality it is very unlikely that a source can be complete for all properties of resources. Therefore, we are only interested in queries and statements whose predicates are IRIs. We use the operator $pred(C)$ for a completeness statement C to represent the set of all IRIs for predicates in C, and similarly, the operator $pred(Q)$ for a query Q.

Definition 6 (Predicate-Relevant Statements). *The completeness statement C is* predicate-relevant *wrt. the query Q if $pred(C) \subseteq pred(Q)$.*

The following proposition shows that if a statement is not predicate-relevant, then it does not contribute anything to completeness reasoning.

Proposition 2. *Let C be a completeness statement and $Q = (W, P)$ be a query. If C is not predicate-relevant wrt. Q, then $[\![Q_C]\!]_{\tilde{P}} = \emptyset$.*

The proposition holds because of the following: Suppose that C is not predicate-relevant wrt. Q. This means that $pred(C) \nsubseteq pred(Q)$. Thus, it is the case that $[\![Q_C]\!]_{\tilde{P}} = \emptyset$ since we cannot find any match for the triple patterns in Q_C with IRIs of the predicates not in $pred(Q)$.

Selectivity of Predicates We argue that the predicate-relevance principle is effective to prune the set of completeness statements in completeness reasoning. We analyze the selectivity of the principle over randomly generated completeness statements and queries with a uniform distribution of predicates, where we assume, for the sake of simplicity, that any two triple patterns in a query or a statement have distinct predicates.

Let us fix the number of possible IRIs for predicates N_p, the length of the completeness statements L_c,[5] and the length of the query L_q. Then, the number of different predicate sets of completeness statements is $\binom{N_p}{L_c}$, which is the number of ways to choose L_c predicates out of the existing N_p. A statement C is relevant to query Q if $pred(C) \subseteq pred(Q)$. Since there are L_q predicates in $pred(Q)$, and all predicates of a relevant statement have to be among these, there are $\binom{L_q}{L_c}$ many ways to choose the predicate set of a relevant statement. We obtain the *selectivity ratio* by dividing the latter number by the former, that is,

$$\alpha(N_p, L_c, L_q) = \frac{\binom{L_q}{L_c}}{\binom{N_p}{L_c}}.$$

As an illustration, for $N_p = 100$, $L_c = 3$, and $L_q = 10$, the selectivity ratio is $\alpha(N_p, L_c, L_q) = 0.000742$. This means that, among 1 million completeness statements, there are only about 742 predicate-relevant statements. Furthermore, it is the case that the greater the value of the number of predicates, the smaller the selectivity ratio. Similarly, the greater the size of the statements, the smaller the ratio. In constrast, the longer the query, the greater the ratio. We believe that for a large data source with heterogeneous domain (e.g., DBpedia), the value of N_p tends to be large. On the other hand, the values of L_c and L_q tend to be small in a relatively constant range. Eventually, the selectivity ratio $\alpha(N_p, L_c, L_q)$ will become very small, meaning the predicate-relevance principle is likely to have a good selectivity, and thus can reduce the number of statements considered in completeness reasoning.

3.3 Techniques for the Retrieval of Predicate-Relevant Statements

For a set $\mathbf{C}$ of completeness statements, we want to know how to retrieve as efficiently as possible those statements that are predicate-relevant wrt. a given query Q. Here, we give an overview of techniques to retrieve such statements.

The statements in $\mathbf{C}$ that are predicate-relevant to Q are those all of whose predicates appear in Q. We denote this set as $\mathbf{C}_Q$, that is,

$$\mathbf{C}_Q = \{\, C \in \mathbf{C} \mid pred(C) \subseteq pred(Q) \,\}.$$

[5] The length is the number of triple patterns in the body of the correspondingCONSTRUCT query of the statements.

To compute $\mathbf{C}_Q$ from $\mathbf{C}$ and Q, is an instance of the well-established *subset querying problem*, which has been investigated by the AI and database communities (see e.g., [14]).

The subset querying problem itself is defined as follows: Given a set $\mathbf{S}$ of sets, and a query set S_q, retrieve all sets in $\mathbf{S}$ that are contained in S_q. In our setting, $\mathbf{S}$ consists of the predicate sets $pred(C)$ of the completeness statements C, while the query set S_q consists of the predicates in Q, that is, $S_q = pred(Q)$.

We study two retrieval techniques based on specialized index structures for subset querying, namely, inverted indexes and tries. In addition, we develop a baseline technique based on standard hashing. In Section 4, we present experimental evaluations comparing retrieval time and scalability for the three techniques.

Running Example Throughout the description below, we will provide examples referring to a set $\mathbf{C} = \{ C_1, C_2, C_3, C_4 \}$ of completeness statements with

- $pred(C_1) = \{ a, b \}$,
- $pred(C_2) = \{ a, b, c \}$,
- $pred(C_3) = \{ a, b, c \}$,
- $pred(C_4) = \{ d \}$,

and a query Q with $pred(Q) = \{ a, b \}$. It is the case that $\mathbf{C}_Q = \{ C_1 \}$, as C_1 is the only statement in $\mathbf{C}$ all of whose predicates are contained in $pred(Q)$.

We now describe how these retrieval techniques work and how we implemented them for our experiments. The implementation language was Java. We represent completeness statements using a class `CompletenessStatement`, while predicates are simply represented by standard Java strings.

Standard Hashing-based Retrieval In this baseline approach, we translate the problem of subset querying into one of evaluating exponentially many set equality queries. Hashing supports equality queries by performing retrieval of objects based on keys. We store completeness statements according to their predicate sets using a hash map. For each of the $2^{|pred(Q)|} - 1$ non-empty subsets of $pred(Q)$, we generate a set equality query using the hash map to retrieve the statements with exactly those predicates. In our example, the non-empty subsets of $pred(Q)$ are $\{a\}$, $\{b\}$, and $\{a, b\}$. Querying for both $\{a\}$ and $\{b\}$ returns the empty set, while querying for $\{a, b\}$ returns the set $\{C_1\}$. Taking the union of these three results gives us $\{C_1\}$ as the final result.

Implementation To index the statements, we use a standard Java `HashMap`. To each statement, we associate a key that uniquely represents the set of its predicates. We do that by creating a lexicographically ordered sequence of the predicates in the statement. We use the standard Java `List` to represent sequences and the `sort` method of the Java `Collections` class for sorting. Then, for such a key, the value in the hash map is the set of all statements having exactly the predicates mentioned in the key. To compute $\mathbf{C}_Q$, we generate all sequences corresponding to the nonempty subsets of $pred(Q)$, retrieve the values to which they are mapped using the `get` method of the `HashMap`, and take the union of the values.

Inverted Indexing-based Retrieval Inverted indexes have been originally developed by the information retrieval community for search engine applications [18]. In the information retrieval domain, an inverted index is a data structure that maps a word to the set of documents containing that word. Inverted indexes are typically used for finding documents containing all words in a search query, that is, for superset querying.

In database applications, inverted indexes are also used for subset querying. In object-oriented databases, objects may have set-valued attributes. Given an attribute and a query set, one may want to find all the objects whose set of attribute values is contained in the query set. Helmer and Moerkotte [9] compared indexing techniques for an efficient evaluation of set operation queries (i.e., subset, superset and set equality) involving low-cardinality set-valued attributes. The indexing techniques they considered were inverted indexes and three other techniques that are signature-based (i.e., sequential signature files, signature trees, and extendible signature hashing). There, an inverted index maps each value to the objects whose set-valued attribute contains that value. Their experimental evaluations showed that in terms of retrieval costs, inverted indexes overall performed best.

Formalization For a set $\mathbf{C}$ of completeness statements, we let $\mathbf{P} = \bigcup_{C \in \mathbf{C}} pred(C)$ be the set of all predicates in $\mathbf{C}$. We define the map $M \colon \mathbf{P} \to 2^{\mathbf{C}}$ such that $M(p) = \{ C \in \mathbf{C} \mid p \in pred(C) \}$ for every predicate $p \in \mathbf{P}$. In other words, M maps each predicate occurring in $\mathbf{C}$ to the set of completeness statements in $\mathbf{C}$ containing that predicate. We call such a map an *inverted index*. The inverted index M of our example is shown below.

Predicates	Completeness Statements
a	C_1, C_2, C_3
b	C_1, C_2, C_3
c	C_2, C_3
d	C_4

We now want to retrieve predicate-relevant statements using inverted indexes. As a first attempt, for a query Q and the inverted index M of a set $\mathbf{C}$ of completeness statements, we consider the set union $\bigcup_{p \in pred(Q)} M(p)$ of the mappings of the predicates occurring in the query. In our example, this is the set $\{ C_1, C_2, C_3 \}$. However, though the resulting set is smaller than the original set $\mathbf{C}$, it is still bigger than $\mathbf{C}_q$, since it contains statements that are not predicate-relevant (i.e., C_2 and C_3).

Now, instead of the set union, let us consider bag union. For a start, assume that $M(p)$ is now a bag that contains as many copies of a statement C as there are occurrences of p in C. In our running example, each $M(p)$ still contains at most one copy of a statement. Next, we take

$$B_Q = \biguplus_{p \in pred(Q)} M(p),$$

which is the bag of all statements that have at least one predicate in Q, and where a statement occurs as many times as it has occurrences of predicates appearing in the query Q. With respect to our example, $B_Q = M(a) \uplus M(b) = \{\!\{ C_1, C_1, C_2, C_2, C_3, C_3 \}\!\}$. Let us analyze which statements are predicate-relevant. The statement C_1 occurs twice in B_Q and has length 2, hence, all its predicates appear in the query Q. However, the statements C_2 and C_3 both have length 3, but occur only twice in B_Q. This means that they have other predicates that do not appear in the query Q and thus, they are not predicate-relevant. Therefore, we conclude that $\mathbf{C}_Q = \{ C_1 \}$.

We can generalize our example to arrive at a characterization of the set $\mathbf{C}_Q$. The example shows that we need to count the occurrences of completeness statements in B_Q. We denote the count of a statement C in B_Q by $\#_C(B_Q)$. As seen from the example, those statements whose number of occurrences is the same as the number of predicates are the predicate-relevant ones. In this case, for a statement C, we take the bag version of $pred(C)$. Then $\mathbf{C}_Q$ satisfies the equation

$$\mathbf{C}_Q = \{ C \in B_Q \mid \#_C(B_Q) = |pred(C)| \}.$$

Implementation We observe from the formalization that the crucial operations for the retrieval technique using inverted indexes are bag union and count. We chose the Google Guava library[6] as it provides a bag implementation in Java with the class `HashMultiset`, which includes as methods the bag union and count. To implement the inverted index, we use the Java `HashMap`. The index maps each predicate p to the `HashMultiset` representing the bag of completeness statements containing that predicate (i.e., $M(p)$). As shown in the formalization above, to retrieve B_Q, we perform a bag union, using the `addAll` method of the `HashMultiset`, of the map values of the predicates in Q. Then, to retrieve the set $\mathbf{C}_Q$ of predicate-relevant statements, we count the number of occurrences of the statements in B_Q using the `count` method of the `HashMultiset` and check if the count is the same as the size of the statement.

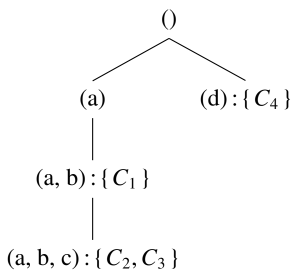
Trie-based Retrieval A trie, or a prefix tree, is an ordered tree for storing sequences, whose nodes are shared between sequences with common prefixes. Tries have been adopted for set-containment queries in the AI community by Hoffmann and Koehler [12] and Savnik [17]. Both studies showed by means of empirical evaluations that tries can be used to efficiently index sets, and perform subset and superset queries upon those sets. Set operations are essential in AI applications, including the matching of a large number of production rules and the identification of inconsistent subgoals during planning.

Formalization We show how to adopt tries to our setting. The sequences we consider are sequences of predicates that are ordered lexicographically. For a set $\mathbf{C}$ of statements, we define $\mathbf{S_C}$ as the set containing for each statement in $\mathbf{C}$ the corresponding sequence of predicates. The *trie* $\mathbf{T_C}$ over the set $\mathbf{S_C}$ of sequences is the tree whose nodes are the prefixes of $\mathbf{S_C}$, denoted as $Pref(\mathbf{S_C})$, where each node $\bar{s} \in Pref(\mathbf{S_C})$ has a child $\bar{s} \cdot p$ iff $\bar{s}{\cdot}p \in Pref(\mathbf{S_C})$, where p is a predicate. On top of this trie, we define $M\colon\ Pref(\mathbf{S_C}) \to 2^{\mathbf{C}}$

[6] `https://github.com/google/guava`

as the mapping that maps each prefix to the set of statements whose predicates are exactly those in the prefix.

In our example, we have that $\mathbf{S_C} = \{ (a,b), (a,b,c), (d) \}$ and $M = \{ (a,b) \mapsto \{ C_1 \}, (a,b,c) \mapsto \{ C_2, C_3 \}, (d) \mapsto \{ C_4 \} \}$. For simplicity, we left out mappings with empty value in M. A graphical representation of the trie $\mathbf{T_C}$ is shown below, which also shows the map value of each node wrt. M.



Having built a trie from completeness statements, we now want to retrieve the predicate-relevant statements wrt. a query. Let us do that for our example. Consider the trie $\mathbf{T_C}$ as above. As $pred(Q) = \{ a, b \}$, the sequence of $pred(Q)$ is therefore $\bar{s}_Q = (a, b)$. The key idea behind our retrieval is that we visit nodes that are subsequences of the query sequence and collect the map values of the visited nodes wrt. M. We start at the root of $\mathbf{T_C}$ with the query sequence (a, b) and an empty set of predicate-relevant statements. The root node is trivially a subsequence of $\bar{s}_Q$ and the mapping of the root obviously returns an empty set. Thus, our set of predicate-relevant statements is still empty.

At this position, we have two options. The first is to retrieve from $\mathbf{T_C}$ all the subsequences containing the head of the current query sequence, that is, the predicate a. By the trie structure, all such subsequences reside in the subtree of $\mathbf{T_C}$ rooted at the concatenation of the root of the current trie and the head of the current query sequence. We then proceed down that subtree. To proceed down, the head of the query sequence has to be removed. Therefore, our current query sequence is now (b). As the map value of the root (a) of the current trie is empty, we still have an empty set of predicate-relevant statements. From this position, we try to visit the subsequences in $\mathbf{T_C}$ that not only contain a, but also one additional predicate from the current query sequence. Therefore, we continue proceeding down the subtree rooted at (a, b), which is the concatenation of the root of the current trie and the head of the current query sequence. From the mapping result of the root (a, b), the set of predicate-relevant statements is now $\{ C_1 \}$. Since our current query sequence is now the empty sequence, we do not proceed further.

Now, let us pursue the second option. We stay at the position at the root of $\mathbf{T_C}$, while simplifying $\bar{s}_Q$ by removing the head of the query sequence, making it now (b). In this case, we want to visit all the subsequences in the trie $\mathbf{T_C}$ that do not contain the predicate a, if they exist. Now, we try to proceed down the subtree rooted at the concatenation of the root of the current trie and the head of the current query sequence. This means we have to proceed down the subtree rooted at (b). Since it does not exist, we stay with the current trie and remove again the head of the query sequence. As the query sequence is now an empty sequence, we do not go further and finish our whole tree traversal. As a final result, we have our set of predicate-relevant statements which contains only C_1.

From our example, we now formalize the retrieval of predicate-relevant statements using tries. We can decompose a non-empty sequence $\bar{s} = (p_1, \ldots, p_n)$ into the head p_1 and the tail $(p_2, \ldots, p_n)$. For a sequence $\bar{s}$ and a trie $\mathbf{T}$, we define $\mathbf{T}/\bar{s}$ as the subtree in $\mathbf{T}$ rooted at the node $\bar{s}$. Note that $\mathbf{T}/\bar{s}$ is the empty tree $\perp$ if such a subtree does not exist. We define $cov(\bar{s}_Q, \mathbf{T_C})$ as the set of completeness statements in $\mathbf{C}$ whose sequences of their predicates are subsequences[7] of $\bar{s}_Q$. It follows from this definition that $cov(\bar{s}_Q, \mathbf{T_C}) = \mathbf{C}_Q$. We observe that the function cov satisfies the following recurrence property, where $\bar{s} = p \cdot \bar{s}'$ is a subsequence of $\bar{s}_Q$ and $\mathbf{T}$ is a subtree of $\mathbf{T_C}$:

$$cov(\bar{s}, \mathbf{T}) = \begin{cases} \emptyset & \text{if } \mathbf{T} = \perp \\ M(root(\mathbf{T})) & \text{if } \bar{s} = () \\ M(root(\mathbf{T})) \cup cov(\bar{s}', \mathbf{T}/root(\mathbf{T}) \cdot p) \cup cov(\bar{s}', \mathbf{T}) & \text{otherwise.} \end{cases}$$

Note that in the above property, the function cov performs pruning: when a subtree in the call $cov(\bar{s}, \mathbf{T}/root(\mathbf{T}) \cdot p)$ does not exist, we cut out all the recursion call possibilities if the subtree existed. Let us give an illustration. For a query sequence $\bar{s}_Q = (p_1, \ldots, p_n)$ of length n, there are at most 2^n possible subsequences. However, half of them (those containing p_1) lie in the tree rooted at the node (p_1). If there is no node (p_1), the size of the search space is immediately reduced to 2^{n-1}.

Implementation. We represent sequences of predicates in $Pref(\mathbf{S_C})$ using the Java class of `List<String>`. For implementing the trie $\mathbf{T_C}$, we create a class `Trie`. For the trie nodes, we create `TrieNode` objects labeled with sequences of predicates. A `TrieNode` has a hash map to store its children, mapping sequences of predicates of the children to the corresponding `TrieNode` objects. Initially, a `Trie` has a `TrieNode` object as its root with an empty sequence as the label. For every insertion of a sequence of the predicates of a completeness statement, we recursively generate children of `TrieNode` objects starting from the root to the leaf node with that sequence as the label. This generates a path of `TrieNode` objects labeled with the prefixes of that sequence. `TrieNode` objects are shared between sequences with the same prefixes. To implement the map M for the trie, a Java `HashMap` similar to the one in the implementation of the standard hashing technique is created.

For the retrieval, we implemented a recursion method based on the recurrence property of the cov function. In the method, for each visited node, we use the `HashMap` of M to map the label of the node to its corresponding set of completeness statements. All the mapping results are collected in a standard Java set which at the end of the method call will be our set $\mathbf{C}_Q$ of predicate-relevant statements.

[7] not necessarily contiguous

4 Experimental Evaluation

We have discussed in Section 3 the predicate-relevance principle as a means to prune the set of completeness statements. We have also introduced three retrieval techniques, based on standard hash maps, inverted indexes, and tries as index structures. We now present an experimental evaluation of completeness reasoning with the aim

1. to compare the retrieval time of the three techniques, and
2. to compare the time needed for completeness reasoning (without and with the predicate-relevance principle) with the time for query evaluation.

4.1 Experimental Setup

We created a framework for the experiments consisting of two components: a completeness reasoner and a random generator of statements and queries.We implemented the framework in Java using the Apache Jena library.[8]

The completeness reasoner includes implementations of the three retrieval techniques as described before and supports two kinds of reasoning: plain reasoning and reasoning based on predicate-relevance. For the former, we simply consider all statements in $\mathbf{C}$, whereas for the latter, we only consider the statements in $\mathbf{C}_Q$.

As RDF completeness statements are not yet available in the real world, we randomly generate queries and sets of completeness statements according to the following parameters:

- number of completeness statements (N_c),
- number of IRIs for predicates (N_p),
- maximum length of completeness statements (L_c), and
- length of queries (L_q).

We describe the rationale behind those parameters. The parameter N_c determines the overall size of the input. We expect that the bigger a data source, the more the completeness statements are declared over that source. The parameter N_p represents the domain heterogeneity of data sources. It is likely that the more heterogeneous a data source, the more heterogeneous are the completeness statements over the source, and therefore, the more varied are the IRIs of the predicates of the statements. While N_c and N_p characterize data sources, the latter two parameters, L_c and L_q characterize the statements and queries. They have been chosen to investigate the sensitivity of each retrieval technique to the size of the completeness statements and the query.

To evaluate the retrieval techniques, we want to observe the influence of each parameter on the retrieval time. Thus, we set up four scenarios, where in each we keep three of the parameters fixed and vary the remaining one. For each of the first three parameters we choose the *default values* $N_p = 2{,}000$, $N_c = 1{,}000{,}000$ and $L_c = 10$. For the query length, we have two default values, to distinguish between short queries ($L_q = 3$) and long queries ($L_q = 10$). We believe that these default values represent a good approximation of realistic parameter values for large RDF data sources, like DBpedia.

[8] `http://jena.apache.org/`

DBpedia has about 2,700 properties and contains 580 million RDF triples, extracted from the English Wikipedia.[9] The first number is close to our value of $N_p = 2{,}000$. Further, if we assume that a fifth of the triples are captured by completeness statements, and that each statement covers 100 triples, then DBpedia would have 1,160,000 completeness statements, which again is close to our value of $N_c = 1{,}000{,}000$. The length of queries were chosen based on the statistics of SPARQL queries over DBpedia. Arias et al. [2] found that 97% of DBpedia queries are of length less than or equal to 3. Therefore, we chose 3 as the length for short queries. On the other hand, 99.9% of queries over DBpedia had length less than or equal to 6, so a value of $L_q = 10$ really stands for exceptionally long queries.

The experiments were run on a standard laptop under Windows 8 with Intel Core i5 2.4 GHz processor and 8 GB RAM. For each combination of parameter values, we ran the experiment 20 times to obtain reliable results (i.e., low variance if we performed the experiments again), and took the median of the running times. We observed that the median was better than the mean as the mean led to some oscillations, though the general trends of the results were the same.

Random Generation of Statements and Queries The statements and queries for the experiments have been generated randomly with a uniform distribution of the predicates. The generated statements were of the form $Compl(P_1 \mid \emptyset)$, that is, they did not have a condition, while the generated queries were of the form $(var(P), P)$, that is, all variables in the body were distinguished. The generation of statements and queries consists essentially in the generation of the triple patterns that are their building blocks.

The triple patterns of a statement are generated as follows. First, we pick a random number between 1 and L_c. This number determines the length of the statement, that is, the number of triple patterns in P_1. Then we randomly choose the predicates of the triple patterns among N_p possible IRIs, where repetitions are allowed. Next, for this collection of predicates, we generate fully-formed triple patterns.

To do that, we instantiate the subjects and objects of triple patterns, by constants or variables. For the instantiation by constants, we always use the same constant, while we do not limit the possibility to introduce new variables. Variables can be reused across triple patterns. We allow only one constant instead of many in order to maximize the chance that statements be applicable to a query. We generate variables in such a way that there is no cross-product join between triple patterns of the statement, that is, the triple patterns with variables form one connected component. Together, the generated triple patterns become the pattern P_1 for that statement. We repeat this process until there are N_c randomly generated statements. Similarly, we generate triple patterns for the query of length L_q, whose collection of predicates are also chosen randomly among the N_p possible IRIs.

4.2 Comparison of Retrieval Techniques for Predicate-Relevant Statements

We now show the experimental results comparing the retrieval time of the three techniques. In each scenario, we vary one of these parameters: number of statements, number of IRIs for predicates, length of completeness statements, and query length.

[9] `http://lists.w3.org/Archives/Public/public-lod/2014Sep/0028.html`

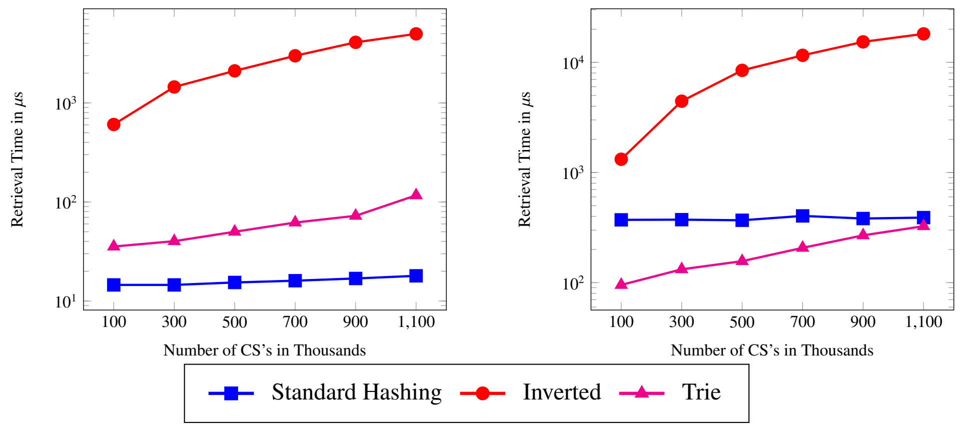


Fig. 1: Increasing the number of completeness statements for short (left) and long queries (right)

Influence of the Number of Completeness Statements In this scenario, we vary the parameter N_c within the range of 100,000 – 1,100,000. Figure 1 shows the resulting retrieval times. The left figure is for short and the right figure for long queries. The y-axis is in log-scale. As can be clearly seen, inverted indexing is always slower than the other techniques for all types of queries. It is on average 40× slower than the tries for short queries and 25× slower than standard hashing for long queries. The performance comparison of standard hashing and the tries, however, depends on the length of the queries. For short queries, standard hashing clearly wins. On average, it is faster by a factor of 3. For long queries, the tries technique is faster, though the retrieval time grows to reach the time for standard hashing. It is also noticeable that standard hashing has a near constant retrieval time, while the other two become slower as the number of statements increases.

One possible reason why inverted indexing is so slow is that it has to process all statements whose predicates overlap with the predicates of the query. Hence, with inverted indexing the probability for a new completeness statement to be processed in the retrieval is much larger than for other retrieval techniques. The other techniques only process statements whose predicates are clearly contained in the query predicates. For standard hashing, the growth is almost unnoticeable because it always evaluates the same set equality queries, albeit over larger collections of statements. For long queries, the tries perform better than the standard hashing. This is likely to be due to the way in which it prunes the set of statements down to the predicate-relevant ones, as described in Section 3.3.

Influence of the Number of IRIs for Predicates In this scenario, we vary the parameter N_p within the range of 400 – 2,800. Figure 2 shows the resulting retrieval times. Both of the graphs clearly show a decrease of the retrieval time for inverted indexing and tries. Interestingly, standard hashing does not share that behavior, as the retrieval time remains constant. Inverted indexing technique still performs worst, as it is on average

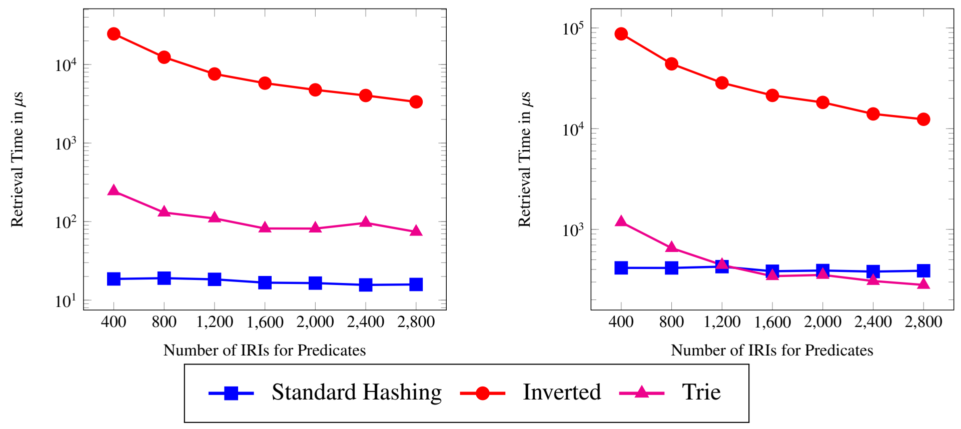


Fig. 2: Increasing the number of IRIs for predicates for short (left) and long queries (right)

about 70× slower than the tries for short queries, and 60× slower for long queries. Though standard hashing clearly outperforms the other techniques for short queries, it starts to be outperformed by the tries for long queries, with the crossover point at $N_p = 1{,}600$.

The decreasing trends for inverted indexing and tries are likely to be due to what we discussed in Section 3.2: in randomly generated statements, the more the IRIs, the less probable it is for a completeness statement to be predicate-relevant. For the standard hashing technique, we do not observe any decrease of the retrieval time for the same reason as in the previous experiment scenario.

Influence of the Length of Completeness Statements In this scenario, we vary the maximum length L_c of completeness statements from 1 to 11. Figure 3 shows the resulting retrieval times. Interestingly, the retrieval time for inverted indexing increases, while the time for tries even decreases. Basically, the retrieval time for standard hashing remains constant, though this time, it shows a little oscillation with no clear pattern. We notice that for long queries, the retrieval times for standard hashing and tries cross over at $L_c = 7$. Again, inverted indexing performs worst, being about 60× slower than the tries for both short and long queries when $L_c = 11$.

These graphs demonstrate the fundamental difference between the inverted indexes and the tries. In the inverted indexes, a completeness statement with just a single predicate overlapping with the query is included in the bag union, to be checked if the statement's occurrences in the union are the same as its length. Thus, the longer the completeness statement, the more probable it is for the statement to be included in the bag union. This does not happen with the trie-based technique as it only processes statements all of whose predicates are contained in the query. When a statement becomes longer, the probability of the statement to be processed by the tries technique decreases. That the growth is constant for standard hashing, is likely to be due to the same reasons as in the previous scenarios.

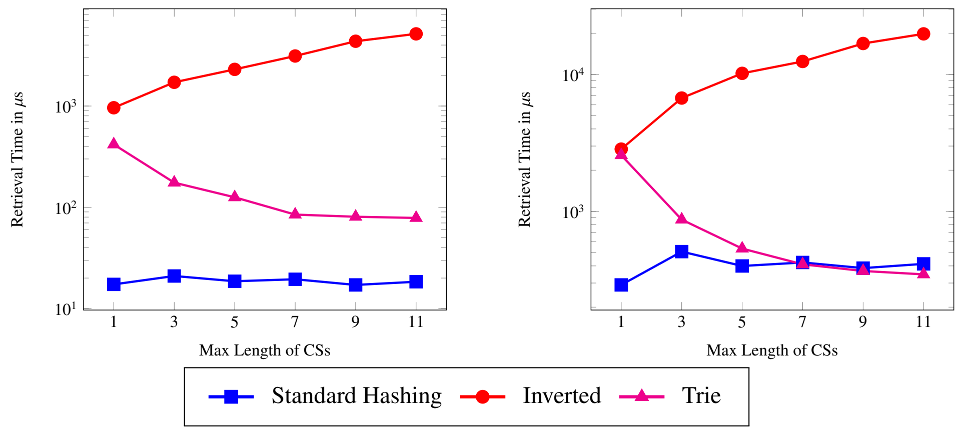


Fig. 3: Increasing maximum length of completeness statements for short (left) and long queries (right)

Influence of the Query Length In this scenario, we vary the query length L_q from 1 to 22. Figure 4 shows the results of this experiment. From the graph, we can see that for all techniques, the retrieval time increases with the query length, though at different rates. For standard hashing, it grows exponentially, whereas for the other techniques, it only grows linearly.[10] In the beginning, the standard hashing technique performs better than the other two. However, from $L_q = 10$ on for the tries and $L_q = 19$ for inverted indexing, the standard hashing technique starts to perform worse. At $L_q = 22$, standard hashing is about 50× slower than inverted indexing, and even about 1,700× slower than the tries. We observe a similarity between the asymptotic growth of inverted indexing and tries, though on an absolute scale the tries technique performs better.

As we expected, standard hashing does not perform well for long queries due to its exponentially many set equality queries. The tries technique, though showing exponential growth in the worst case, performs better than standard hashing. This is most likely due to its pruning in the retrieval, as the tries technique works based on subsequences of the predicates in the query.

From the experiments we conclude that in almost all cases, our baseline approach, the standard hashing, shows the best performance despite its simplicity. However, for long queries, the tries technique is comparable to the baseline, and is even much better for extremely long queries. The baseline approach is infeasible for extremely long queries due to its exponential blow up. The inverted indexes are not suitable for the retrieval task as even though they have the same asymptotic growth as the tries for all the scenarios except the one varying the statement length L_c, the absolute retrieval times are much worse than those of the tries. Moreover, on an absolute scale, the retrieval time of the best technique of each scenario only takes up to 1,000 μs (1 ms). This shows that the retrieval process does not add a significant overhead to completeness reasoning.

[10] Note that the graph is displayed in log-scale on y-axis.

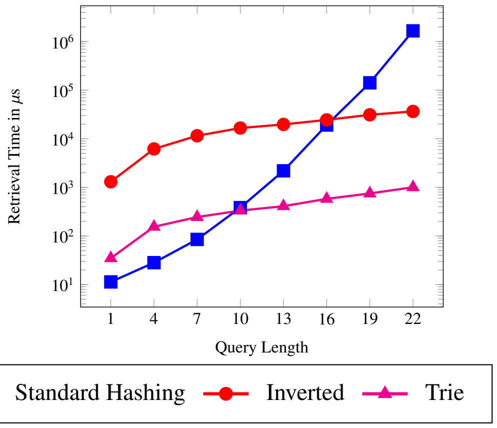


Fig. 4: Increasing the query length

4.3 Overhead of Completeness Reasoning

This scenario differs from the above in that now we compare the cost of completeness reasoning with the cost of query evaluation. We show that applying the predicate-relevance principle can considerably reduce the overhead incurred by completeness reasoning.

To measure this overhead, we perform experiments that compare the running times of plain completeness reasoning, of reasoning based on predicate-relevance, and of query evaluation. For the reasoning based on predicate-relevance, we use the standard hashing retrieval technique. All the parameter values are the default ones: $N_p = 2{,}000$, $N_c = 1{,}000{,}000$, and $L_c = 10$, while we still distinguish between short queries ($L_q = 3$) and long queries ($L_q = 10$). For each kind, we randomly generate 20 queries, so that in total there are 40 randomly generated queries. For query evaluation, we map each predicate of the generated query to a property in the DBpedia Ontology,[11] map the constant in the query to a fixed DBpedia resource, and leave the variables as they are.

In the experiments we measure the reasoning time for plain completeness reasoning, the reasoning plus the retrieval time for the completeness reasoning based on predicate-relevance, and the query evaluation time. We set up a local DBpedia mirror on Virtuoso with a Dual Core Intel Xeon Processor with 2.66 GHz and 16 GB RAM.

Now we discuss the experimental results. Table 2 lists the median of running times of plain completeness reasoning, predicate-relevance based completeness reasoning, and query evaluation. We note that completeness reasoning based on predicate-relevance is considerably faster than the plain one (i.e., milliseconds vs. minutes, respectively), and almost as fast as query evaluation.

Completeness reasoning without the predicate-relevance principle is clearly infeasible, with running times between 30,000 times (for short queries) and 10,000 times (for long queries) slower than with predicate-relevance.This is due to the fact that much

[11] `http://oldwiki.dbpedia.org/Downloads2014#dbpedia-ontology`

Table 2: Comparison of the running time median for plain completeness reasoning, predicate-relevance based (optimized) reasoning, and query evaluation

Query Types	Plain Reasoning	Optimized Reasoning	Query Evaluation
Short	758,001 ms	24 ms	20 ms
Long	781,286 ms	80 ms	36 ms

fewer completeness statements are considered for the reasoning using the predicate-relevance principle. For short queries, there are on average about 160 predicate-relevant completeness statements, whereas for long queries, there are on average about 510 predicate-relevant statements. On the other hand, the original set contains 1 million statements.

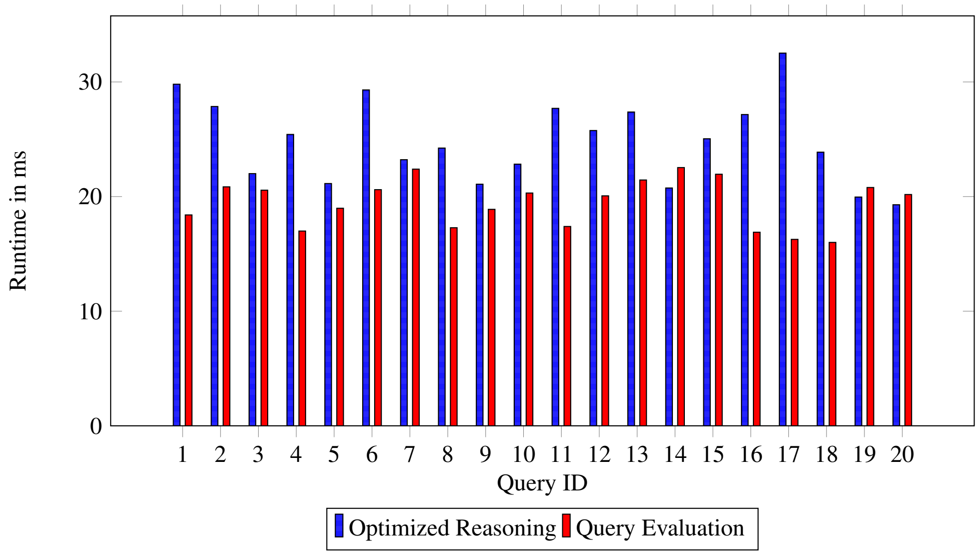


Fig. 5: Individual comparison of predicate-relevance based (optimized) completeness reasoning and query evaluation for *short* queries

For short queries, evaluation is just slightly faster than optimized completeness reasoning (20 ms vs. 24 ms), while for long queries, it is about two times faster (36 ms vs. 80 ms). The median of the overhead for all queries is 1.2× for the short ones and 2.4× for the long ones. In the experiments, query evaluation was fast, since all queries produce empty results over DBpedia, as the queries are generated randomly. Note that this is in fact the best case for query evaluation. Hence, this shows that the predicate-relevance based reasoning does not add much overhead to query evaluation, as real queries are

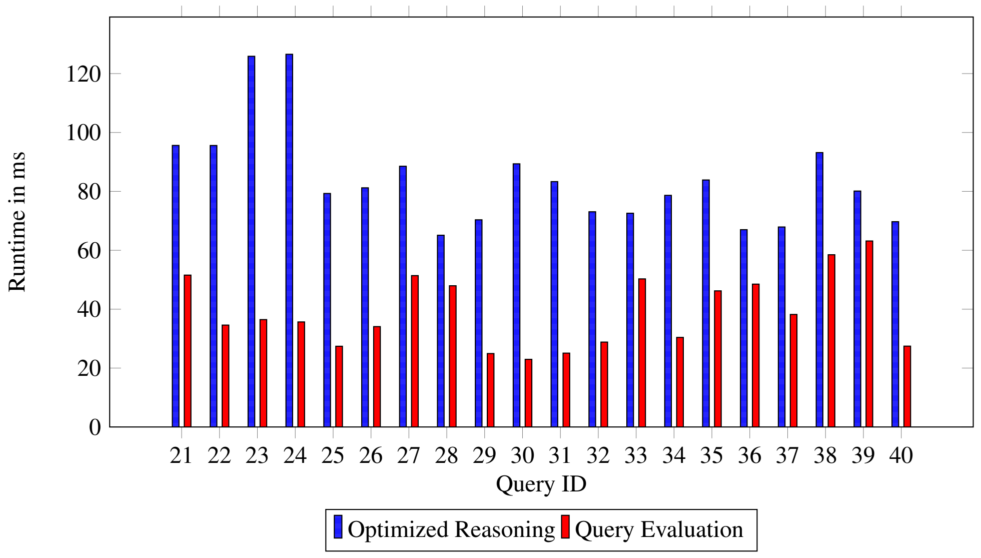


Fig. 6: Individual comparison of predicate-relevance based (optimized) completeness reasoning and query evaluation for *long* queries

likely to return some results, and hence take longer to evaluate. More detailed comparisons can be seen in Figures 5 and 6, which comfront the predicate-relevance based reasoning time and execution time for individual queries. In Figure 5, we see that in most of the cases reasoning is almost as fast as query evaluation, with the exception of three cases where reasoning performs slightly better. In Figure 6, we see that in all cases, query evaluation is faster than completeness reasoning. Note that this is due to long queries generating more predicate-relevant statements, hence increasing the reasoning time. From these experiments, we conclude two things: that the predicate-relevance principle can effectively reduce the completeness reasoning time, and that predicate-relevance based completeness reasoning does not add much overhead to query evaluation.

4.4 Conclusions from the Experiments

In this section, we addressed the problem of performing completeness reasoning over large sets of completeness statements. In general, one can perform reasoning by considering all completeness statements of the data source. However, this is problematic for large sets of statements. In our experiments, we found that reasoning with 1 million statements took up to 13 minutes. Thus, we introduced the predicate-relevance principle as a means to reduce the number of completeness statements considered in completeness reasoning. To realize the predicate-relevance principle, we developed retrieval techniques based on three different index structures, namely standard hash maps, inverted indexes, and tries. As an outcome of our experimental evaluation, both standard hashing and tries allow for fast retrieval of predicate-relevant statements. While standard hashing technique is suitable in the general case, the tries technique is more appropriate for border cases (e.g., extremely long queries). In the end, in our evaluation we found that completeness reasoning using the predicate-relevance principle ran faster than the plain one by a factor of 30,000 for short queries and 10,000 for long queries. Consequently, the overhead of completeness reasoning is considerably reduced and the running time becomes comparable to that of query evaluation. In our experiments, query evaluation was just slightly faster than completeness reasoning for short queries (20 ms vs. 24 ms), and was about twice as fast for long queries (36 ms vs. 80 ms).

As a note, we have dealt with synthetic data only, as real data is not available yet. We assumed a uniform distribution of data, though in reality, data can be skewed. Skewedness appears naturally in a data source with a limited domain. In such a case, one may refine the relevance principle by taking also the subjects and objects positions into account. An open question then is how to handle variables, as we cannot assume that subjects or objects are constants.

5 Conclusions

Real-world RDF data sources like DBpedia may contain a large amount of data. Consequently, to describe their completeness one would need a large number of completeness statements. We presented techniques for efficient completeness reasoning over large sets of statements based on the predicate-relevance principle to rule out a significant number of irrelevant statements in completeness reasoning. We developed several retrieval techniques for predicate-relevant statements based on different index structures. From our experimental evaluations, we concluded that our techniques can considerably reduce the time needed for completeness reasoning, and make it comparable to query evaluation time.

References

1. Keith Alexander, Richard Cyganiak, Michael Hausenblas, and Jun Zhao. *Describing Linked Datasets with the VoID Vocabulary*. W3C Interest Group Note, 3 March 2011. Retrieved Feb 1, 2015 from `http://www.w3.org/TR/2011/NOTE-void-20110303/`.
2. Mario Arias, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. An Empirical Study of Real-World SPARQL Queries. In *Proceedings of the 1st International Workshop on Usage Analysis and the Web of Data (USEWOD'11)*, 2011.
3. Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia – A Crystallization Point for the Web of Data. *Journal of Web Semantics*, 7(3), 2009.
4. Dan Brickley and Ramanathan V. Guha, editors. *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C Recommendation, 10 February 2004. Retrieved Feb 1, 2015 from `http://www.w3.org/TR/2014/REC-rdf-schema-20140225/`.
5. Ashok K. Chandra and Philip M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Proceedings of the 9th ACM Symposium on Theory of Computing (STOC'77)*, 1977.
6. Fariz Darari, Werner Nutt, Giuseppe Pirrò, and Simon Razniewski. Completeness Statements About RDF Data Sources and Their Use for Query Answering. In *Proceedings of the 12th International Semantic Web Conference (ISWC'13)*, 2013.

7. Steve Harris and Andy Seaborne, editors. *SPARQL 1.1 Query Language*. W3C Recommendation, 21 March 2013. Retrieved Feb 1, 2015 from `http://www.w3.org/TR/2013/REC-sparql11-query-20130321/`.
8. Tom Heath and Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Synthesis Lectures on the Semantic Web: Theory and Technology. Morgan & Claypool, 2011.
9. Sven Helmer and Guido Moerkotte. A Performance Study of Four Index Structures for Set-Valued Attributes of Low Cardinality. *VLDB Journal*, 12(3), 2003.
10. Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph, editors. *OWL 2 Web Ontology Language Document Overview (Second Edition)*. W3C Recommendation, 11 December 2012. Retrieved Feb 1, 2015 from `http://www.w3.org/TR/2012/REC-owl2-primer-20121211/`.
11. Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, Edwin Lewis-Kelham, Gerard de Melo, and Gerhard Weikum. YAGO2: Exploring and Querying World Knowledge in Time, Space, Context, and Many Languages. In *Proceedings of the 20th International Conference on World Wide Web (WWW'11)*, 2011.
12. Jörg Hoffmann and Jana Koehler. A New Method to Index and Query Sets. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*, 1999.
13. Graham Klyne and Jeremy J. Carroll, editors. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation, 10 February 2004. Retrieved Feb 1, 2015 from `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`.
14. Sergey Melnik and Hector Garcia-Molina. Adaptive Algorithms for Set Containment Joins. *ACM Trans. Database Syst.*, 28(1), 2003.
15. Amihai Motro. Integrity = Validity + Completeness. *ACM Trans. Database Syst.*, 14(4), 1989.
16. Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.
17. Iztok Savnik. Index Data Structure for Fast Subset and Superset Queries. In *International Cross Domain Conference and Workshop (CD-ARES'13)*, 2013.
18. Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. An Efficient Indexing Technique for Full-Text Databases. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB'92)*, 1992.