Comparing Index Structures for Completeness Reasoning

Fariz Darari
Universitas Indonesia
Depok – Indonesia
fariz@cs.ui.ac.id

Werner Nutt
Free University of Bozen-Bolzano
Bozen-Bolzano – Italy
nutt@inf.unibz.it

Simon Razniewski
Max Planck Institute for Informatics
Saarbrücken – Germany
srazniew@mpi-inf.mpg.de

Abstract—**Data quality is a major issue in the development of knowledge graphs. Data completeness is a key factor in data quality pertaining to how broad and deep is information contained in knowledge graphs. As for large-scale knowledge graphs (e.g., DBpedia, Wikidata), it is conceivable that given the vast amount of information contained in there, they may be complete for a wide range of topics, such as children of Joko Widodo, cantons of Switzerland, and presidents of Indonesia. Previous research has shown how one can augment knowledge graphs with statements about their completeness, stating which parts of data are complete. Such meta-information can be leveraged to check query completeness, that is, whether the answer returned by a query is complete. Yet, it is still unclear how such a check can be done in practice, especially when many completeness statements are involved. We devise implementation techniques to make completeness reasoning in the presence of large sets of completeness statements feasible, and experimentally evaluate their effectiveness in realistic settings based on the characteristics of real-world knowledge graphs.**

I. Introduction

Real-world knowledge graphs may be very large. DBpedia,[1] for instance, contains at least 580 million facts extracted from English Wikipedia alone,[2] whereas Wikidata[3] has over 370 million facts about 42 million entities.[4] Given such a quantity, one may wonder, what quality those knowledge graphs possess?

Data quality plays an important role in the development of knowledge graphs. Data completeness is a key aspect of data quality that deals with how broad and deep is information contained in data sources (or in our context, knowledge graphs) [1]. Generally speaking, data over knowledge graphs is treated in either of the two ways: data is assumed to be complete (i.e., the closed-world assumption), or data is treated to be incomplete (that is, the open-world assumption) [2]. In the real-world, however, it is often necessary to employ a mix between the two assumptions: for some parts of data, they are complete; though for other parts, they are (still) potentially incomplete.[5] Managing data completeness involves providing and making explicit metadata pertaining to which parts of data can be regarded as complete, and which parts cannot.

In practice, there is a substantial amount of Web data sources providing (natural language) metadata about completeness. For example, OpenStreetMap provides around 2,300 pages with completeness status,[6] and Wikipedia contains nearly 15,000 pages having the keywords "list is complete" and "complete list of". While such completeness metadata can be helpful for data editors in order to be better informed as to which data topics are complete, the lack of formal, machine readable completeness metadata hinders the automatic processing of such metadata, which could otherwise enable advanced usages such as completeness analytics, search optimization, and query completeness checking.

Related Work. In previous research, Darari et al. [3] proposed a completeness management framework for (RDF-based [4]) knowledge graphs. They

[1] http://dbpedia.org

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

[3] http://wikidata.org

[4] https://tools.wmflabs.org/wikidata-todo/stats.php

[5] That is, those parts of data upon which the completeness is still unknown.

[6] For example, see https://wiki.openstreetmap.org/wiki/Ahlen

formalized completeness descriptions over knowledge graphs and provided a machine-readable representation for those descriptions. Furthermore, they investigated the problem of query completeness checking: the check whether completeness statements can guarantee the completeness of a query. For instance, having the statements "complete for all children of US presidents" and "complete for all spouses of US presidents" would guarantee the completeness of the query "give all children and spouses of US presidents". Their work, however, concentrated on how such checking can be formalized, without elaborating how it can be done in a scalable manner. In [5], Prasojo et al. developed COOL-WD, a completeness management tool for Wikidata knowledge graph. The tool contains over 10,000 completeness statements about entities in Wikidata. While some simple heuristics has been deployed for the tool, there is still no optimization provided for reasoning with (general) completeness statements. In [6], Darari et al. demonstrated CORNER, a system for checking query completeness based on metadata about completeness, as formalized in [3]. Yet, the system is not able to handle large-scale cases.

Contributions. In this paper we concentrate on the engineering aspect of the problem of completeness and expand upon the work of Darari et al. [3] by optimizing query completeness checking. In particular, our contributions are twofold: (*i*) We propose indexing techniques for completeness statements based on our analysis that the problem of finding completeness statements relevant for query completeness checking can be reduced to the established problem of subset querying (Section III); and (*ii*) based on realistic settings, we conduct experimental evaluations for the problem of query completeness checking (Section IV).

II. Formal Framework

In this work, knowledge graphs are described within the context of RDF (Resource Description Framework) knowledge graphs, which have recently gained increasing attentions [7].

A. Knowledge Graph Modeling and Querying

In this paper, we model knowledge graphs using RDF, and query them using SPARQL. We assume three sets L (*literals*), I (*IRIs*, short for Internationalized Resource Identifiers), and V (*variables*). Literals and IRIs altogether can be referred to as *terms* (or *constants*). An *RDF triple* (or simply a *triple*) is an element $(s, p, o) \in I \times I \times (I \cup L)$. A finite set G of triples is called an RDF graph.

SPARQL (SPARQL Protocol and RDF Query Language) is the standard query language for RDF [8]. It builds upon *triple patterns,* which are like triples, but with the addition of variables. In this paper, we concentrate on conjunctive SPARQL, where sets of triple patterns, called BGPs (Basic Graph Patterns), are used in querying. We call a partial function $\mu\colon L \rightarrow I \cup V$ a *mapping*. Consider a BGP P. The BGP μP denotes variable replacement in P with terms based on μ. Evaluating a BGP P over G, written as $\llbracket P \rrbracket_G$, results in a set of mappings where for every mapping $\mu \in \llbracket P \rrbracket_G$, we have that $\mu P \subseteq G$. For a BGP P, we define the *freeze mapping* $\tilde{id}$ as replacing each variable $?v$ in P to a fresh IRI $\tilde{v}$. From a freeze mapping, the *prototypical graph* $\tilde{P} := \tilde{id}\, P$ can be constructed as a representative for any graph satisfying P. Prototypical graphs will be used later on when characterizing query completeness checking.

SPARQL queries are of three forms: `SELECT`, `ASK`, and `CONSTRUCT` queries. We write a `SELECT` query as (W, P) such that W is a set of variables and P is a BGP. The evaluation of a `SELECT` query is by evaluating $\llbracket P \rrbracket_G$, and then projecting the resulting mappings over W. Such an evaluation is written as $\llbracket Q \rrbracket_G = \pi_W(\llbracket P \rrbracket_G)$. When W is empty, a `SELECT` query can be called an `ASK` query. As for the `CONSTRUCT` query, its form is (P_1, P_2) where P_1 and P_2 are all BGPs. The evaluation of a `CONSTRUCT` query over G results in a graph where the BGP P_1 has been instantiated with all mappings in $\llbracket P_2 \rrbracket_G$. This paper focuses on query evaluation using bag semantics, where duplicates are not removed.

B. Knowledge Graph Completeness

Completeness Statements. Completeness statements capture which topics of a knowledge graph are complete. A *completeness statement* C has the form $Compl(P_C)$ where P_C is a non-empty BGP. Say, we state that a graph is complete for all triple pairs "$?f$ *is a film and* $?f$ *is writ-*

ten by Spielberg" using the statement $C_f = \mathit{Compl}((?f, a, \mathit{Film}), (?f, \mathit{writtenBy}, \mathit{spielberg}))$.

To model the open-world assumption of RDF graphs, we use the notion of available graph G, which we actually have, and ideal graph G', which is a hypothetical, complete graph. We call a pair (G, G') of two graphs, where $G \subseteq G'$, an *extension pair*.

Generally speaking, any graph can be an ideal graph. Yet, the introduction of completeness statements limits the possibilities of ideal graphs: only graphs with no new information wrt. available graphs for the parts covered by the statements can be ideal graphs. For a statement $C = \mathit{Compl}(P_C)$, there is the associated CONSTRUCT query $Q_C = (P_C, P_C)$. A statement C is satisfied by an extension pair (G, G'), written $(G, G') \models C$, whenever $[\![Q_C]\!]_{G'} \subseteq G$. A completeness statement satisfaction basically means that the available graph G is complete for all the information as captured in C wrt. the ideal graph G'. We naturally extend the above definition to a set $\mathbf{C}$ of completeness statements: $(G, G') \models \mathbf{C}$ iff $[\![Q_C]\!]_{G'} \subseteq G$ for every $C \in \mathbf{C}$.

Query Completeness. When querying a knowledge graph, we may be interested to see whether our query is *complete* wrt. the real world. For example, when querying DBpedia for films written by Spielberg, we may want to know if we actually retrieve all such films. A query is complete over an extension pair if the answers returned over the ideal state are also there over the available state. Given a SELECT query Q, we write $\mathit{Compl}(Q)$ to state the completeness of Q. Query completeness $\mathit{Compl}(Q)$ is satisfied by an extension pair (G, G') whenever it is the case that $[\![Q]\!]_{G'} = [\![Q]\!]_G$. This case is written as $(G, G') \models \mathit{Compl}(Q)$.

Completeness Entailment. From the notions of completeness statements and query completeness, we may naturally ask: when can information about the completeness of knowledge graphs guarantee query completeness? We approach the question by 'quantifying' over all extension pairs[7] such that whenever the completeness statements are satisfied by an extension pair, then the extension pair also satisfies the query completeness. We now define completeness entailment. Let $\mathbf{C}$ be a set of completeness statements and Q be a SELECT query. The statements $\mathbf{C}$ *entail the completeness of the query* Q, denoted by $\mathbf{C} \models \mathit{Compl}(Q)$, if for every extension pair satisfying $\mathbf{C}$, it satisfies $\mathit{Compl}(Q)$.

As an illustration, consider C_f as above. Whenever C_f is satisfied by an extension pair (G, G'), then G must contain all triples about films written by Spielberg. Now let us consider the query $Q_f = (\{\,?f\,\}, \{\,(?f, a, \mathit{Film}), (\,?f, \mathit{writtenBy}, \mathit{spielberg}), (?f, \mathit{writtenBy}, \mathit{miller})\,\})$ asking for films written by both Spielberg and Miller. In this case, the statement C_f is not sufficient to guarantee the completeness of Q_f. It might be that Miller wrote a film (that was also written by Spielberg) but this information is missing in the available graph, leading to the non-inclusion of the film in the query result. The query completeness can be guaranteed, for instance, by having an additional statement about the completeness of films written by Miller.

Checking if the completeness of a query $Q = (W, P)$ can be entailed by a set $\mathbf{C}$ of completeness statements can be characterized as follows: first, all the associated CONSTRUCT queries of the completeness statements are evaluated over the prototypical graph $\tilde{P}$ of the query, and second, we check if the evaluation result contains $\tilde{P}$. We construct the transfer operator $T_\mathbf{C}$ for the bulk evaluation of completeness statements: $T_\mathbf{C}(G) = \bigcup_{C \in \mathbf{C}} [\![Q_C]\!]_G$. In the following, we have our theorem of completeness entailment:

Theorem 1: [3] $\mathbf{C} \models \mathit{Compl}(Q)$ iff $\tilde{P} = T_\mathbf{C}(\tilde{P})$.

As query completeness checking corresponds to linearly evaluating CONSTRUCT queries over the frozen BGP of the conjunctive query, its complexity is NP-complete [3].

With respect to the above examples of C_f and Q_f, it is the case that $\tilde{P_f} = \{(\tilde{f}, a, \mathit{Film}), (\tilde{f}, \mathit{writtenBy}, \mathit{spielberg}), (\tilde{f}, \mathit{writtenBy}, \mathit{miller})\}$, while the application of the transfer operator gives us $T_{\{C_f\}}(\tilde{P_f}) = \{(\tilde{f}, a, \mathit{Film}), (\tilde{f}, \mathit{writtenBy}, \mathit{spielberg})\}$. Hence, according to our theorem, it is the case that $\{C_f\} \not\models \mathit{Compl}(Q_f)$.

[7] In this case, not only we abstract over ideal graphs, but also available graphs.

III. Indexing Techniques

A. Relevant Completeness Statements

Let us first provide an estimation of the complexity of completeness reasoning. From Theorem 1, the completeness reasoning task is the checking whether $T_{\mathbf{C}}(\tilde{P})$ equals $\tilde{P}$, where $T_{\mathbf{C}}$ is the transfer operator wrt. $\mathbf{C}$, and $\tilde{P}$ is the prototypical graph of Q. Note that the '$\subseteq$' direction of the equality can be seen immediately. We now focus on the '$\supseteq$' direction, which corresponds to obtaining, for each triple in $\tilde{P}$, a statement $C \in \mathbf{C}$ s.t. the triple is in $[\![Q_C]\!]_{\tilde{P}}$ (note that $T_{\mathbf{C}}(\tilde{P}) = \bigcup_{C \in \mathbf{C}} [\![Q_C]\!]_{\tilde{P}}$). Thus, only statements that *potentially* match such a triple (s, p, o) are required to be processed.

We now analyze the overall runtime of completeness reasoning in realistic settings. Given a set $\mathbf{C}$ of completeness statements and a query $Q = (W, P)$, the following three parameters are involved in the reasoning: the maximum number of triple patterns in a completeness statement, denoted by $\mathit{maxLn}(\mathbf{C})$; the number of completeness statements in $\mathbf{C}$, denoted by $|\mathbf{C}|$; and the number of triples in the prototypical graph, denoted by $|\tilde{P}|$. Note that in the transfer operator $T_{\mathbf{C}}$, we apply for every statement, its associated `CONSTRUCT` query over the prototypical graph of the query. Hence, the overall runtime of completeness reasoning is: $O(|\mathbf{C}||\tilde{P}|^{\mathit{maxLn}(\mathbf{C})})$.

As usual in the data complexity analysis for relational database queries, we are assuming that the query Q is given whereas the set of completeness statements varies. Furthermore, since completeness statements can be treated as queries, we assume $\mathit{maxLn}(\mathbf{C})$ to be bounded by a constant. Under these assumptions, the complexity of completeness reasoning is a function of the number of completeness statements. Using a *plain completeness reasoner*, which considers *all* completeness statements, may lead to non-optimal performance. Therefore, we would like to find an approach to trim down the number of completeness statements in completeness reasoning.

Constant-Relevance Principle. Consider the query asking for "Films written by Spielberg" and the statement "All presidents of Indonesia." It is intuitive that the statement does have any contribution as to whether the query is guaranteed to be complete or not; namely, the statement is *irrelevant to the query*.

We now introduce the *constant-relevance principle* as a direction to differentiate between irrelevant and relevant completeness statements. The principle asserts that whenever a completeness statement C is found to be relevant for the completeness of a query, then it must be that all completeness statement's constants are included in the constants of the query, expressed formally as: $\mathit{const}(C) \subseteq \mathit{const}(Q)$. A statement satisfying this principle is called *constant-relevant*. The following proposition says that whenever a statement is not constant-relevant, then the statement does not contribute to completeness reasoning.

Proposition 1: Let $Q = (W, P)$ be a query and C a completeness statement. It holds that $[\![Q_C]\!]_{\tilde{P}} = \emptyset$ if C is not constant-relevant wrt. Q.

The above proposition opens up the problem of retrieving constant-relevant completeness statements efficiently.

Problem Definition. Given a set $\mathbf{C}$ of completeness statements and a query Q, we would like to retrieve the set of constant-relevant completeness statements wrt. the query. We denote this set as $\mathbf{C}_Q$, that is,

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

This computation of $\mathbf{C}_Q$ is an instance of *subset querying problem*, previously investigated by the database and AI communities [9]–[11]. The problem of subset querying is as follows: Let $\mathbf{S}$ be a set of sets, and S_q query set. Subset querying is obtaining all sets in $\mathbf{S}$ that are a subset of S_q. In our case, the elements of $\mathbf{S}$ are sets of constants in the completeness statements of $\mathbf{C}$, whereas constants in Q would be the query set S_q.

We investigate inverted indexes and tries, as index structures to support subset querying, and standard hashing as a baseline index structure.

Running Example. Throughout the description below, we refer to a set $\mathbf{C} = \{\, C_1, C_2, C_3, C_4 \,\}$ of statements having $\mathit{const}(C_1) = \{\, a, b \,\}$, $\mathit{const}(C_2) = \{\, a, b, c \,\}$, $\mathit{const}(C_3) = \{\, a, b, c \,\}$, $\mathit{const}(C_4) = \{\, d \,\}$, and a query Q having $\mathit{const}(Q) = \{\, a, b \,\}$. Here, we have that $\mathbf{C}_Q = \{\, C_1 \,\}$, as C_1 is the only constant-relevant statement wrt. Q.

B. Standard Hashing-based Retrieval

As for our baseline index structure, we provide a translation of subset querying problem into (an exponential number of) set equality queries. Hashing performs equality queries by retrieving objects using keys. Using a hash map, completeness statements are stored based on the statements' constant sets. There are thus $2^{|\mathit{const}(Q)|} - 1$ set equality queries generated for non-empty subsets of $\mathit{const}(Q)$. In our above example, the generated set equality queries are $\{a\}$, $\{b\}$, and $\{a, b\}$. It is immediate to see that from these queries, only $\{a, b\}$ returns a non-empty set, that is, $\{C_1\}$

As for the implementation, we rely on a standard Java `HashMap`, where the key is constructed by setting an ordering for completeness statement's constants, and the value is the set of all statements with those constants.

C. Inverted Indexing-based Retrieval

Traditionally, inverted indexes are used in the information retrieval domain [12]. In the domain of object-oriented databases, inverted indexing may help in subset querying. Helmer and Moerkotte [9] conducted a comparative experimental evaluation for different index structures for set operation queries involving set-valued attributes. They showed that inverted indexes generally performed best in terms of retrieval costs.

Formalization. Let $\mathbf{C}$ be a set of completeness statements and $\mathbf{P} = \bigcup_{C \in \mathbf{C}} \mathit{const}(C)$ the set of all constants in $\mathbf{C}$. An inverted index M provides a mapping from constants in $\mathbf{P}$ to bags of completeness statements in $\mathbf{C}$, which records the number of occurrences of completeness statements wrt. the statement's constants. With respect to our example, the inverted index M is as follows.

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

Now, we define $B_Q = \biguplus_{p \in \mathit{const}(Q)} M(p)$ as the bag of completeness statements having (at least) one constant appearing in Q, where the multiplicity is based on how many times the constants of the statements occur in the query Q. In our example, we have that $B_Q = M(a) \uplus M(b) = \{\!\!|\, C_1, C_1, C_2, C_2, C_3, C_3 \,|\!\!\}$. From those statements, only the statement C_1 is constant relevant, since it has two constants and occurs also two times in B_Q. Thus, it holds that that $\mathbf{C}_Q = \{\, C_1 \,\}$. We abstract over this example for characterizing the set $\mathbf{C}_Q$. Let us denote the number of appearances of a statement C in B_Q by $\#_C(B_Q)$. Only statements that appear as many times as the number of their constants are constant-relevant. Here, we employ the bag version of $\mathit{const}(C)$. The set $\mathbf{C}_Q$ satisfies the following equation: $\mathbf{C}_Q = \{\, C \in B_Q \mid \#_C(B_Q) = |\mathit{const}(C)| \,\}$.

The crucial operations for using the inverted indexes approach we described above are count and bag union. We use the Google Guava library[8] which provides the class `HashMultiset` with the methods `addAll` (to support bag union) and `count` (to count the number of occurrences of statements in a bag).

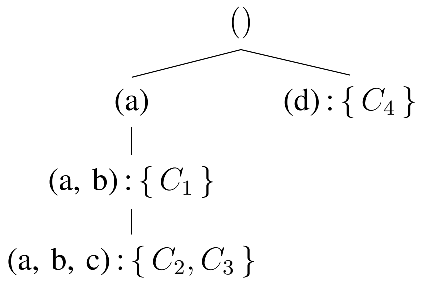
D. Trie-based Retrieval

A trie is an ordered tree that stores sequences with the condition that the tree's nodes are shared between sequences having common prefixes. In the AI community (e.g., [10] and [11]), tries have been used for set-containment queries efficiently.

Formalization. Let $\mathbf{C}$ be a set of completeness statements. The set $\mathbf{S_C}$ is defined as the set consisting of sequences of constants from each statement in $\mathbf{C}$. Over $\mathbf{S_C}$, we define $\mathbf{T_C}$ as the tree where its nodes are prefixes of the set $\mathbf{S_C}$, denoted as $\mathit{Pref}(\mathbf{S_C})$, with the following condition: each node $\bar{s}$ in $\mathit{Pref}(\mathbf{S_C})$ has a child node $\bar{s} \cdot p$ where p is a constant if and only if $\bar{s} \cdot p$ is in $\mathit{Pref}(\mathbf{S_C})$. Furthermore, a trie is augmented with a mapping $M \colon \mathit{Pref}(\mathbf{S_C}) \rightarrow 2^{\mathbf{C}}$, that maps prefixes to statements having the prefix constants.

In our example above, the set $\mathbf{S_C}$ of sequences is $\{\, (a, b), (a, b, c), (d) \,\}$ and the mapping M is $\{\, (a, b) \mapsto \{\, C_1 \,\}, (a, b, c) \mapsto \{\, C_2, C_3 \,\}, (d) \mapsto \{\, C_4 \,\} \,\}$. Mappings to the empty value are left out for the sake of simplicity. A pictorial representation of $\mathbf{T_C}$ is shown as follows.

[8] https://github.com/google/guava



From the trie that we build from completeness statements, we now develop a technique to retrieve the constant-relevant completeness statements wrt. a query. A non-empty sequence $\bar{s} = (p_1, \ldots, p_n)$ can be decomposed into two parts: the head p_1 and the tail $(p_2, \ldots, p_n)$. Given a sequence $\bar{s}$ and a trie $\mathbf{T}$, the subtree of $\mathbf{T}$ rooted at the node $\bar{s}$ is defined as $\mathbf{T}/\bar{s}$. Whenever a subtree does not exist, $\mathbf{T}/\bar{s}$ returns the empty tree $\perp$. The set of statements in $\mathbf{C}$ with their constant sequences being a (not necessarily contiguous) subsequence of $\bar{s}_Q$ is defined as $\mathit{cov}(\bar{s}_Q, \mathbf{T}_\mathbf{C})$. From this definition, it follows that $\mathit{cov}(\bar{s}_Q, \mathbf{T}_\mathbf{C}) = \mathbf{C}_Q$. Our observation is that *cov*, given a subtree $\mathbf{T}$ of $\mathbf{T}_\mathbf{C}$ and a subsequence $\bar{s} = p \cdot \bar{s}'$ of $\bar{s}_Q$, satisfies the following property:

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

In the above recurrence property, we can observe two base cases: the empty set is returned, if the trie is empty; and if the sequence $\bar{s}$ is already empty, *cov* returns the corresponding set of completeness statements wrt. $\mathit{root}(\mathbf{T})$. The recursive case consists of three components, each has simpler forms than the original trie. The *cov* function performs pruning, as also noted in [10]: it trims down all recursive possibilities whenever a subtree does not exist.

As for the trie implementation, we create a class `Trie` where each of its nodes is labeled by some prefix sequence. Prefix sequences are built from constants in completeness statements and are implemented using Java lists. We implement a recursive method based on the *cov* function. For each node, we use a Java `HashMap` for mapping the node's prefix sequence to the associated set of statements. The union of the mapping results will be our set of constant-relevant completeness statements.

IV. Experimental Evaluation

We have presented in the previous section three different indexing schemes that can be used for retrieving constant-relevant completeness statements. In this section, we report on our experimental evaluation investigating: (*i*) "How do the number of statements, the length of statements, and query length impact on the retrieval time under the three indexing schemes?"; and (*ii*) "Which indexing approach performs best in which setting?".

A. Experimental Setup

We randomly generate queries and sets of completeness statements based on three parameters: (*i*) number of statements (N_c), (*ii*) max length of statements (L_c), and (*iii*) query length (L_q).

Based on how we vary the parameter values, we provide three distinct experiment scenarios. We take DBpedia, one of the most prominent RDF knowledge graphs, as our reference for the parameters' default values. We extracted around 580 million RDF triples from the English version of DBpedia.[9] Assuming that $\frac{1}{5}$ of those triples are covered by completeness statements, and that every statement captures 100 RDF triples, DBpedia would own around 1 million completeness statements. Thus, this sets the default value $N_c =$ 1,000,000. The parameter values for query length are also chosen according to real-world statistics of DBpedia SPARQL queries [13]. We therefore choose $L_q = 3$ as the default value for query length wrt. short queries, and $L_q = 6$ for query length wrt. long queries.

We implemented our experiment program using Jena.[10] We ran our experiments on a basic laptop with a 2.5 GHz processor and 8 GB memory. Each observation was taken from the median of 20 runs.

Random Generation of Statements and Queries. To have some degree of freedom in adjusting our parameters, we generated randomly the statements and queries of the experiments based on realistic settings, using (again) DBpedia as our reference point. We set the IRI constant possibilities in the predicate position to be 2500, and those in the subject or object position to be 1,000,000. The

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

completeness statements generated were of the form $Compl(P)$, while the queries were $(var(P), P)$. Our random generation was designed such that no cross-product joins were produced, and that variable and constant repetitions were allowed.

B. Experimental Results

1) Number of Completeness Statements: Here, we vary the number of completeness statements while fixing completeness statement length and query length (recall that still there are two modes of query length). The parameter N_c ranges from 100,000 to 1,000,000. Figure 1 shows the retrieval times of different N_c treatments. We set the y-axis in log-scale. We can clearly observe that inverted indexing performs the worst. It is three times slower than standard hashing for queries that are long, and worse, 53× slower than tries when queries are short. Both standard hashing and tries techniques have their own strengths. While for queries that are short, standard hashing is a little faster, tries perform better for long queries.

Inverted indexing is the slowest here possibly due to its processing of all completeness statements having at least one overlapped constant with the query. This would result in a higher probability of a completeness statement to be 'touched' in the retrieval processing than that with other index structures.

2) Completeness Statement Length: Here the maximum length of statements is varied at $1 - 6$. As shown in Figure 1, we see a contrast between the retrieval time growth of inverted indexing and tries. This is likely due to the higher probability of completeness statements to be included in the reasoning with inverted indexing when they become longer. On the other hand, for tries the probability decreases, since to be processed in tries, all the constants in a statement need to be included in the query. In this scenario, standard hashing performs best for short queries, while tries are best for long queries. Inverted indexing is not suitable here.

3) Query Length: In this case, the query length is varied from 1 to 6. As seen from Figure 1, the retrieval time of constant-relevant statements increases as the query becomes longer, and that standard hashing seems to suffer the most. At first, standard hashing runs faster than tries. Yet, it starts to perform worse from $L_q = 4$. While the time growth of inverted indexing and tries is similar, tries are better on an absolute scale. Standard hashing does not provide good performance when queries are long due to its exponential number of set equality queries. Tries have a pruning ability which could cut down the number of search space in the retrieval process.

4) Reasoning with the Constant-Relevant Filtering: As opposed to the above scenarios, here we provide a runtime comparison of reasoning with- and without the indexing at all. Due to its good performance for retrieving constant-relevant statements, we use standard hashing as a representative index structure. The default values are used here as the experiment parameters: $L_c = 6$, $N_c =$ 1,000,000, and the two query modes of short version ($L_q = 3$) and long version ($L_q = 6$). We see below a table reporting the reasoning time comparison for the plain method and the indexed method (where the retrieval time for constant-relevant statements is already included).

Query Types	Plain Reasoning	Optimized Reasoning
Short	145,773 ms	1.3 ms
Long	146,095 ms	4.1 ms

We observe that our indexing technique can effectively reduce the completeness reasoning time (i.e., from minutes to just milliseconds). Using the constant-relevance principle (with the standard hashing as the index structure) achieves a 110,000× runtime improvement when queries are short, and a 35,000× runtime improvement when queries are long. The main reason is that for the optimized technique, there are much fewer completeness statements considered in the reasoning.

C. Discussion

For short queries, our baseline approach, the standard hashing, shows the best performance, while for long queries, tries perform better. Inverted indexes appear not suitable for the retrieval task. Furthermore, completeness reasoning using the constant-relevant technique takes just a few milliseconds, as opposed to minutes for the unoptimized reasoning.

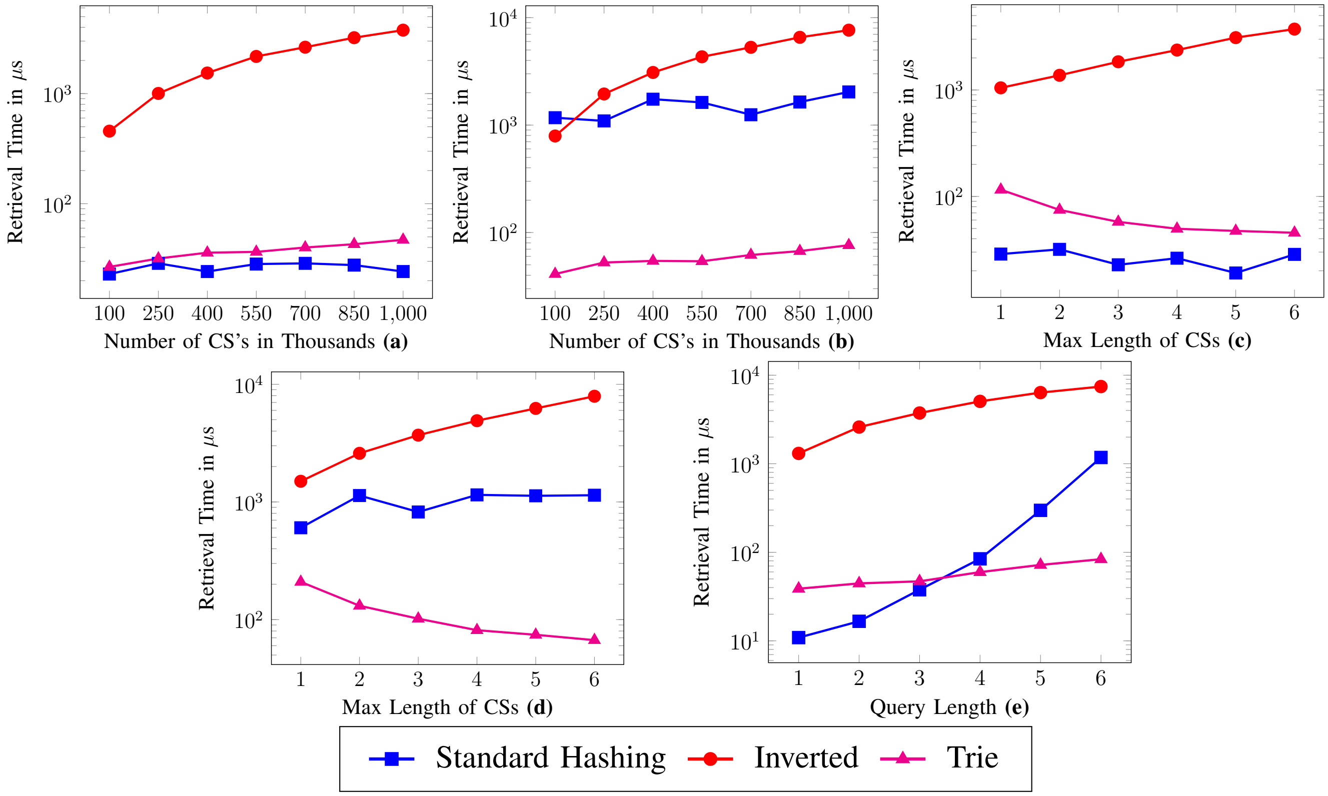


Fig. 1. Increasing N_c for short (a) and long queries (b); Increasing L_c for short (c) and long queries (d); Increasing L_q (e)

V. Conclusions

We presented techniques for efficient completeness reasoning over large sets of statements based on the constant-relevance principle to filter out a large number of irrelevant completeness statements. We developed retrieval techniques for constant-relevant statements based on different index structures: standard hashing, inverted indexes, and tries. Our experiments showed that the proposed techniques enable the deployment of completeness reasoning to large datasets. For future work, we plan to investigate completeness reasoning optimizations with even more number of completeness statements (e.g., hundreds of millions of statements). Exploring other potential index structures for completeness reasoning is also of our interest.

Acknowledgments

This work was partially supported by TaDaQua, funded by the Free University of Bolzano, Italy.

References

[1] R. Y. Wang and D. M. Strong, "Beyond accuracy: What data quality means to data consumers," *J. of Management Information Systems*, vol. 12, no. 4, pp. 5–33, 1996.

[2] W. Fan and F. Geerts, *Foundations of Data Quality Management*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

[3] F. Darari, W. Nutt, G. Pirrò, and S. Razniewski, "Completeness statements about RDF data sources and their use for query answering," in *ISWC*, 2013.

[4] G. Klyne and J. J. Carroll, Eds., *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation, 10 February 2004.

[5] R. E. Prasojo, F. Darari, S. Razniewski, and W. Nutt, "Managing and Consuming Completeness Information for Wikidata Using COOL-WD," in *COLD*, 2016.

[6] F. Darari, R. E. Prasojo, and W. Nutt, "CORNER: A completeness reasoner for SPARQL queries over RDF data sources," in *ESWC Posters & Demos*, 2014.

[7] W. Zheng, L. Zou, W. Peng, X. Yan, S. Song, and D. Zhao, "Semantic SPARQL similarity search over RDF knowledge graphs," *PVLDB*, vol. 9, no. 11, pp. 840–851, 2016.

[8] S. Harris and A. Seaborne, Eds., *SPARQL 1.1 Query Language*. W3C Recommendation, 21 March 2013.

[9] S. Helmer and G. Moerkotte, "A Performance Study of Four Index Structures for Set-Valued Attributes of Low Cardinality," *VLDB Journal*, vol. 12, no. 3, 2003.

[10] J. Hoffmann and J. Koehler, "A New Method to Index and Query Sets," in *IJCAI*, 1999.

[11] I. Savnik, "Index Data Structure for Fast Subset and Superset Queries," in *CD-ARES*, 2013.

[12] J. Zobel, A. Moffat, and R. Sacks-Davis, "An Efficient Indexing Technique for Full-Text Databases," in *VLDB*, 1992.

[13] M. Arias, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente, "An Empirical Study of Real-World SPARQL Queries," in *USEWOD*, 2011.