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 that concerns the breadth, depth, and scope of information contained in knowledge graphs. As for large-scale knowledge graphs (e.g., DBpedia, Wikidata), it is conceivable that given the amount of information contained in there, they may be complete for a wide range of topics, such as children of Donald Trump, 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 a large number of 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 contain a large amount of data. 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 the breadth, depth, and scope of information contained in data sources (or in our context, knowledge graphs) [1]. Generally, 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 assumed to be incomplete (i.e., 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 over 14,000 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 parts of data 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 framework for managing completeness over (RDF-based [4]) knowledge graphs.

[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

They 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 tool to manage completeness over Wikidata knowledge base. 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 focus 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 as follows: (*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*) we conduct experimental evaluations based on realistic settings 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. RDF and SPARQL

We assume three pairwise disjoint infinite sets I (*IRIs*), L (*literals*), and V (*variables*). We collectively refer to IRIs and literals as *terms* (or *constants*). A 3-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* G consists of a finite set of triples [4]. For simplicity, we omit namespaces for the abstract representation of RDF graphs.

The standard query language for RDF is SPARQL (SPARQL Protocol and RDF Query Language) [8]. The basic building blocks of a SPARQL query are *triple patterns,* which resemble triples, except that in each position also variables are allowed. We focus on the conjunctive fragment of SPARQL, which uses sets of triple patterns, called *basic graph patterns* (BGPs). A *mapping* μ is a partial function $\mu \colon V \rightarrow I \cup L$. Given a BGP P, μP denotes the BGP obtained by replacing variables in P with terms according to μ. The evaluation of a BGP P over an RDF graph G, denoted as $[\![P]\!]_G$, results in a set of mappings such that for every mapping $\mu \in [\![P]\!]_G$, it holds $\mu P \subseteq G$. For a BGP P, we define the *freeze mapping* $\tilde{id}$ as mapping each variable $?v$ in P to a fresh IRI $\tilde{v}$. From such a mapping, we construct the *prototypical graph* $\tilde{P} := \tilde{id}\, P$ to represent any possible graph satisfying the BGP P. Prototypical graphs will be used later on when characterizing query completeness checking.

SPARQL queries come as `SELECT`, `ASK`, or `CONSTRUCT` queries. A `SELECT` query has the abstract form (W, P), where P is a BGP and $W \subseteq var(P)$. 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. A `CONSTRUCT` query has the abstract form (P_1, P_2), where both P_1 and P_2 are BGPs, and $var(P_1) \subseteq var(P_2)$. Evaluating a `CONSTRUCT` query over G yields a graph where P_1 is instantiated with all the mappings in $[\![P_2]\!]_G$. In this paper, the semantics considered in query evaluation is the bag semantics, which is the default of SPARQL [8]. In bag semantics, duplicates of query answers are kept.

B. Knowledge Graph Completeness

Completeness Statements. Completeness statements capture which topics of a knowledge graph are complete. A *completeness statement* C has the form $\mathit{Compl}(P_C)$ where P_C is a non-empty BGP. For example, we express that a graph is complete for all pairs of triples that say "*$?m$ is a movie (= Mov) and $?m$ is directed (= dir) by Tarantino*" using the statement $C_{\mathit{dir}} = \mathit{Compl}((?m, a, \mathit{Mov}), (?m, \mathit{dir}, \mathit{tarantino}))$, whose BGP matches all such pairs.

To model the open-world assumption of RDF graphs, we define an *extension pair* as a pair (G, G') of two graphs, where $G \subseteq G'$. We call G the *available graph* and G' the *ideal graph*. Here, an available graph is the graph that we currently store, while an ideal graph is a hypothetical extension over the available graph, representing a version of ideal, complete information.

Without completeness statements, any graph extending the available graph can be an ideal graph. Completeness statements restrict the possibilities of ideal graphs: for the parts captured by completeness statements, they must contain no more information than in the available graph. To a statement $C = \mathit{Compl}(P_C)$, we associate the CONSTRUCT query $Q_C = (P_C, P_C)$. Note that, given a graph G, the query Q_C returns a graph consisting of those instantiations of the pattern P_C present in G. For example, the query $Q_{C_{\mathit{dir}}}$ returns the movies directed by Tarantino in a graph G. An extension pair (G, G') *satisfies* the statement C, written $(G, G') \models C$, if $[\![Q_C]\!]_{G'} \subseteq G$. Intuitively, whenever an extension pair (G, G') satisfies a completeness statement C, then the subgraph of G' captured by C is also present in G. The above definition naturally extends to the satisfaction of a set $\mathbf{C}$ of completeness statements, that is, $(G, G') \models \mathbf{C}$ iff for all $C \in \mathbf{C}$, it is the case that $[\![Q_C]\!]_{G'} \subseteq G$.

An important tool for characterizing completeness entailment is the transfer operator $T_{\mathbf{C}}$, which captures the complete parts of a graph w.r.t. a set of completeness statements. Given a set $\mathbf{C}$ of completeness statements and a graph G, the *transfer operator* is defined as $T_{\mathbf{C}}(G) = \bigcup_{C \in \mathbf{C}} [\![Q_C]\!]_G$.

Query Completeness. When querying a knowledge graph, we may want to know whether the query is *complete* w.r.t. the real world. For instance, when querying DBpedia for movies directed by Tarantino, it would be interesting to know whether we really get *all* such movies. Intuitively, over an extension pair a query is complete whenever all answers we retrieve over the ideal state are also retrieved over the available state. Given a SELECT query Q, to express that Q *is complete*, we write $\mathit{Compl}(Q)$. An extension pair (G, G') *satisfies* $\mathit{Compl}(Q)$, if the result of Q evaluated over G' is the same as Q over G, that is, $[\![Q]\!]_{G'} = [\![Q]\!]_G$. In this case we write $(G, G') \models \mathit{Compl}(Q)$.

Completeness Entailment. From the notions above, a question naturally arises as to when some meta-information about data completeness can provide a guarantee for query completeness. We approach the question by 'quantifying' over all extension pairs[7] such that if an extension pair satisfies the completeness statements, then it must also satisfy the query completeness. We define completeness entailment as follows. 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 $\mathbf{C} \models \mathit{Compl}(Q)$, if any extension pair satisfying $\mathbf{C}$ also satisfies $\mathit{Compl}(Q)$.

As an illustration, consider C_{dir} as above. Whenever an extension pair (G, G') satisfies C_{dir}, then G contains all triples about movies directed by Tarantino. Now let us consider the query $Q_{\mathit{dir}} = (\{\,?m\,\}, \{\ (?m, a, \mathit{Mov}),\ (\,?m, \mathit{dir}, \mathit{tarantino}),\ (?m, \mathit{dir}, \mathit{miller})\,\})$ asking for movies directed by both Tarantino and Miller. In this case, the statement C_{dir} is not sufficient to guarantee the completeness of Q_{dir}. It might be that Miller directed a movie (that was also directed by Tarantino) but this director information is missing in the available graph, leading to the non-inclusion of the movie in the query result. The query completeness can be guaranteed, for instance, by having an additional statement about the completeness of movies directed by Miller.

To check whether the completeness of a query $Q = (W, P)$ is entailed by a set $\mathbf{C}$ of completeness statements, we evaluate all the corresponding CONSTRUCT queries of the statements over the prototypical graph $\tilde{P}$ and check whether in the evaluation result, we have $\tilde{P}$ back. Intuitively, this means that over any possible graph instantiation for answering the query, the completeness statements

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

guarantee that we have the graph instantiation back in our available graph. Formally:

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

As query completeness checking corresponds to evaluating a linear number of CONSTRUCT queries over the (frozen) conjunctive body of the query, its complexity is NP-complete [3].

With respect to our example of C_{dir} and Q_{dir}, it is the case that $\tilde{P}_{dir} = \{(\tilde{m}, a, \mathit{Mov}), (\tilde{m}, \mathit{dir}, \mathit{tarantino}), (\tilde{m}, \mathit{dir}, \mathit{miller})\}$, while the transfer operator gives us $T_{\{C_{dir}\}}(\tilde{P}_{dir}) = \{(\tilde{m}, a, \mathit{Mov}), (\tilde{m}, \mathit{dir}, \mathit{tarantino})\}$. Hence, according to our theorem, it is the case that $\{C_{dir}\} \not\models \mathit{Compl}(Q_{dir})$.

III. Indexing Techniques

A. Relevant Completeness Statements

Before formulating a principle to optimize completeness reasoning, let us first estimate the complexity of the reasoning task. From Theorem 1, the completeness reasoning task is the checking whether $T_{\mathbf{C}}(\tilde{P}) = \tilde{P}$, where $T_{\mathbf{C}}$ is the transfer operator w.r.t. $\mathbf{C}$, and $\tilde{P}$ is the prototypical graph of Q. Note that the '$\subseteq$' direction of the equality can be seen immediately. The interesting part is the '$\supseteq$' direction, which corresponds to finding, for each triple $(s, p, o) \in \tilde{P}$, a completeness statement $C \in \mathbf{C}$ such that $(s, p, o) \in [\![Q_C]\!]_{\tilde{P}}$ (recall 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.

Let $\mathbf{C}$ be a set of completeness statements, *maxLn*$(\mathbf{C})$ the maximum length (i.e., the maximum number of triple patterns) of statements in $\mathbf{C}$, and $Q = (W, P)$ a query. Take any $C \in \mathbf{C}$; to evaluate the query Q_C over $\tilde{P}$, it is necessary to map the triple patterns of Q_C to triples in $\tilde{P}$. Note there are at most $|\tilde{P}|^{|Q_C|}$ possibilities 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. Hence, applying this reasoning to each statement in $\mathbf{C}$ results in the following overall runtime: $O(|\mathbf{C}||\tilde{P}|^{\mathit{maxLn}(\mathbf{C})})$.

As customary in database theory when analyzing the data complexity of query evaluation, 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 the maximum length of statements to be bounded by a constant. Under these assumptions, the complexity of reasoning is a function of the number of completeness statements. Using a *plain completeness reasoner*, which evaluates the CONSTRUCT queries of *all* statements, may lead to poor performance. Thus, we need to find an approach to trim down the number of completeness statements in completeness reasoning.

Constant-Relevance Principle. Consider the query asking for "Movies directed by Tarantino" and the statement "All presidents of Indonesia." It is intuitive that the statement does have any contribution to the completeness of the query; namely, the statement is *irrelevant to the query*.

Let us now introduce the *constant-relevance principle* as a direction to distinguish between irrelevant and relevant completeness statements. The principle states that a completeness statement C can contribute to query completeness only if all constants of the completeness statement are present in the query Q, that is, *const*$(C) \subseteq$ *const*(Q). We say that a statement satisfying this principle is *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 C be a completeness statement and $Q = (W, P)$ be a query. If C is not constant-relevant w.r.t. Q, then $[\![Q_C]\!]_{\tilde{P}} = \emptyset$.

Proposition 1 opens up the problem of how to retrieve constant-relevant statements in an efficient manner. In the following part, we present retrieval techniques for constant-relevant completeness statements.

Problem Definition. For a set $\mathbf{C}$ of completeness statements, we want to know how to retrieve those statements that are constant-relevant w.r.t. a given query Q. We denote this set as $\mathbf{C}_Q$, that is,

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

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 database and AI communities [9]–[11]. 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 constant sets $\mathit{const}(C)$ of the completeness statements C, while the query set S_q consists of the constants in Q, that is, $S_q = \mathit{const}(Q)$.

We study two retrieval techniques based on specialized index structures for subset querying, namely, inverted indexes and tries. Additionally, we develop a baseline technique using standard hashing.

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

B. 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 constant sets using a hash map. For each of the $2^{|\mathit{const}(Q)|} - 1$ non-empty subsets of $\mathit{const}(Q)$, we generate a set equality query using the hash map to retrieve the statements with exactly those constants. In our example, the non-empty subsets of $\mathit{const}(Q)$ 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.

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

Inverted indexes have been originally developed by the information retrieval community for search engine applications [12]. 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. 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 $\mathbf{P} = \bigcup_{C \in \mathbf{C}} \mathit{const}(C)$ be the set of all constants in $\mathbf{C}$. We define a map M that maps from constants in $\mathbf{P}$ to bags of completeness statements in $\mathbf{C}$, where $M(p)$ is a bag that contains as many copies of a statement C as there are occurrences of the constant p in C. We call such a map an *inverted index*. The inverted index M of our example is shown below.

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

Next, we take $B_Q = \biguplus_{p \in \mathit{const}(Q)} M(p)$, which is the bag of all statements that have at least one constant in Q, and where a statement occurs as many times as it has occurrences of constants 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 constant-relevant. The statement C_1 occurs twice in B_Q and has two constants, hence, all its constants appear in the query Q. However, the statements C_2 and C_3 both have three constants, but occur only twice in B_Q. This means that they have other constants that do not appear in the query Q and thus, they are not constant-relevant. Therefore, we conclude that $\mathbf{C}_Q = \{ C_1 \}$.

We can generalize our example to arrive at a characterization of the set $\mathbf{C}_Q$. We denote the occurrence count of a statement C in B_Q by $\#_C(B_Q)$. As seen from the example, those statements whose occurrence count is the same as the number of constants are the constant-relevant ones. In this case, for a statement C, we take the bag version of *const*(C). Then, $\mathbf{C}_Q$ satisfies the equation $\mathbf{C}_Q = \{\, C \in B_Q \mid \#_C(B_Q) = |\mathit{const}(C)| \,\}$.

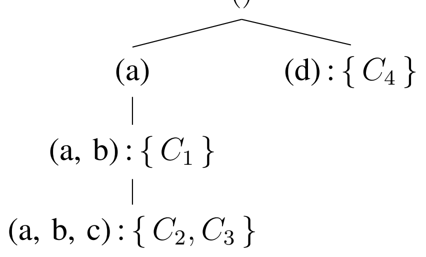
The crucial operations for the implementation of the retrieval technique using inverted indexes are bag union and count. We use the Google Guava library[8] which provides the class `HashMultiset` with the methods `addAll` (to support bag union) and `count` (to count the number of occurrences of statements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

D. Trie-based Retrieval

A trie, 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 [10] and Savnik [11]. 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.

Formalization. We again assume a lexicographical ordering of constants. 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 constants. 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 $\bar{s} \in \mathit{Pref}(\mathbf{S_C})$ has a child $\bar{s} \cdot p$ iff $\bar{s} \cdot p \in \mathit{Pref}(\mathbf{S_C})$, where p is a constant. On top of this trie, we define $M \colon \mathit{Pref}(\mathbf{S_C}) \to 2^{\mathbf{C}}$ as the mapping that maps each prefix to the set of statements whose constants 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 the empty value in M. A graphical representation of the trie $\mathbf{T_C}$ is shown below.



Having built a trie from completeness statements, we now want to retrieve the constant-relevant statements w.r.t. a query. Let us do that for our example. Consider the trie $\mathbf{T_C}$ as before. As *const*$(Q) = \{\, a, b \,\}$, the sequence of *const*(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 w.r.t. M. We start at the root of $\mathbf{T_C}$ with the query sequence (a, b) and an empty set of constant-relevant statements. The root node is trivially a subsequence of $\bar{s}_Q$ and the mapping of the root obviously returns the empty set. Thus, our set of constant-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 constant 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 constant-relevant statements. From this position, we try to visit the subsequences in $\mathbf{T_C}$ that not only contain a, but also one additional constant 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 constant-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 constant 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

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

again the head of the query sequence. As the query sequence is now the empty sequence, we do not go further and finish our whole tree traversal. As a final result, we have our set of constant-relevant statements which contains only C_1.

From our example, we now formalize the retrieval of constant-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 $\mathit{cov}(\bar{s}_Q, \mathbf{T}_\mathbf{C})$ as the set of completeness statements in $\mathbf{C}$ whose sequences of their constants are subsequences of $\bar{s}_Q$, which are not necessarily contiguous. It follows from this definition that $\mathit{cov}(\bar{s}_Q, \mathbf{T}_\mathbf{C}) = \mathbf{C}_Q$. Given a subsequence $\bar{s} = p \cdot \bar{s}'$ of $\bar{s}_Q$ and a subtree $\mathbf{T}$ of $\mathbf{T}_\mathbf{C}$, we observe that the function *cov* satisfies the following recurrence property:

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

The recurrence property has two base cases: when the trie is empty, then simply the empty set is returned; and when there is no element left in the sequence $\bar{s}$ (i.e., the trie traversal stops), the *cov* function returns the set of completeness statements associated with the sequence $\mathit{root}(\mathbf{T})$. Now for the recursive case, there are three components involved. The first one is simply returning the set of completeness statements associated with $\mathit{root}(\mathbf{T})$. The second and third ones correspond to how the trie is traversed: both make the *cov* calls with the tail $\bar{s}'$ of $\bar{s}$ as the call's sequence, but the second case is over the subtree $\mathbf{T}/(\mathit{root}(\mathbf{T}) \cdot p)$ while the third one is over the same trie $\mathbf{T}$.

Note that in the above property, as also observed in [10], the function *cov* performs pruning: when a subtree in the call $\mathit{cov}(\bar{s}, \mathbf{T}/(\mathit{root}(\mathbf{T}) \cdot p))$ 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}.

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. For the retrieval, we implement a recursive method based on the recurrence property of the *cov* function. In the method, for each visited node, we use the `HashMap` of the mapping M to map the label (i.e., prefix sequence) 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 constant-relevant 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 completeness statements, the length of completeness statements, and the length of the query 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 completeness statements (N_c), (*ii*) maximum length of completeness statements (L_c), and (*iii*) length of queries (L_q).

We set up three scenarios, where in each we keep two of the parameters fixed and vary the remaining one. As our reference for setting the default values for the parameters, we take DBpedia [13], one of the most popular and largest RDF knowledge graphs, as an approximation of the realistic parameter values. From the English Wikipedia, DBpedia extracted around 580 million RDF triples.[9] If we assumed that $\frac{1}{5}$ of the triples are captured by completeness statements, and that each statement covers 100 triples, DBpedia would have 1,160,000 completeness statements. Therefore, we

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

set the default value $N_c =$ 1,000,000. The length of queries is chosen based on statistics of SPARQL queries over DBpedia. Arias et al. [14] found that 97% of DBpedia queries are of length less than or equal to 3. Therefore, we choose 3 as the default length for short queries. On the other hand, 99.9% of queries over DBpedia had length less than or equal to 6, so a length of 6 stands for relatively long queries. So, there are two default values for query length: $L_q = 3$ for the short ones, and $L_q = 6$ for the long ones. As for the default value of L_c, we set it to 6, to have a variation of completeness statement length from 1 to 6, which covers the query length.

Our experiment program was implemented in Java using the Apache Jena library.[10] The experiments were run on a standard laptop under Windows 8 with Intel Core i5 2.5 GHz processor and 8 GB RAM, and we report medians over 20 runs.

Random Generation of Statements and Queries. The statements and queries for the experiments have been generated randomly with a uniform distribution of the IRIs for constants. Again, we take DBpedia as our reference. DBpedia has about 2,700 properties and 4.5 million entities, We approximate the number of constant IRIs in the predicate position from the number of properties of DBpedia, that is, 2,500, and the number of constant IRIs in the subject or object position from about $\frac{1}{5}$ of the number of DBpedia entities, that is, 1,000,000. The generated statements were of the form $Compl(P)$, while the generated queries were of the form $(var(P), P)$, that is, all variables in the body were distinguished. Generating the statements and queries is essentially generating triple patterns, which serve as their building blocks.

The triple patterns of a statement are generated as follows. First, we pick a random length between 1 and L_c. Then we randomly choose the predicates of the triple patterns, 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. 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 for that statement. We repeat this process until there are N_c randomly generated statements. We generate triple patterns for queries in a similar way.

[10]http://jena.apache.org/

B. Results

1) Number of Completeness Statements: In this scenario, we vary the parameter N_c within the range of 100,000 – 1,000,000. Figure 1 shows the resulting retrieval times for short queries (1.a) and for long ones (1.b). The y-axis is in log-scale. As can be clearly seen, inverted indexing is generally slower and less scalable than the other techniques. It is on average 53× slower than tries for short queries and 3× slower than standard hashing for long queries. The performance comparison of standard hashing and tries, however, depends on the length of the queries. For short queries, standard hashing is slightly faster. For long queries, the tries technique is faster.

One possible reason why inverted indexing is slow is that at an intermediate step it has to process all statements whose constants overlap with the constants of the query. Hence, with inverted indexing the probability for a completeness statement to be processed in the retrieval is much larger than for other retrieval techniques. The other techniques only process statements whose constants are clearly contained in the query constants. For long queries, the tries perform better than the standard hashing. This is likely due to the subsequence pruning of tries.

2) Length of Completeness Statements: In this scenario, we vary the maximum length L_c of completeness statements from 1 to 6. Figure 1 shows the resulting retrieval times for short queries (1.c) and long queries (1.d). Notably, the retrieval time for inverted indexing increases, while the time for tries even decreases. Basically, the retrieval time for standard hashing remains constant, though showing a little oscillation with no clear pattern. We notice that for short queries, standard hashing performs best, whereas for long queries, tries perform best. Again, inverted indexing performs the worst among all the retrieval techniques.

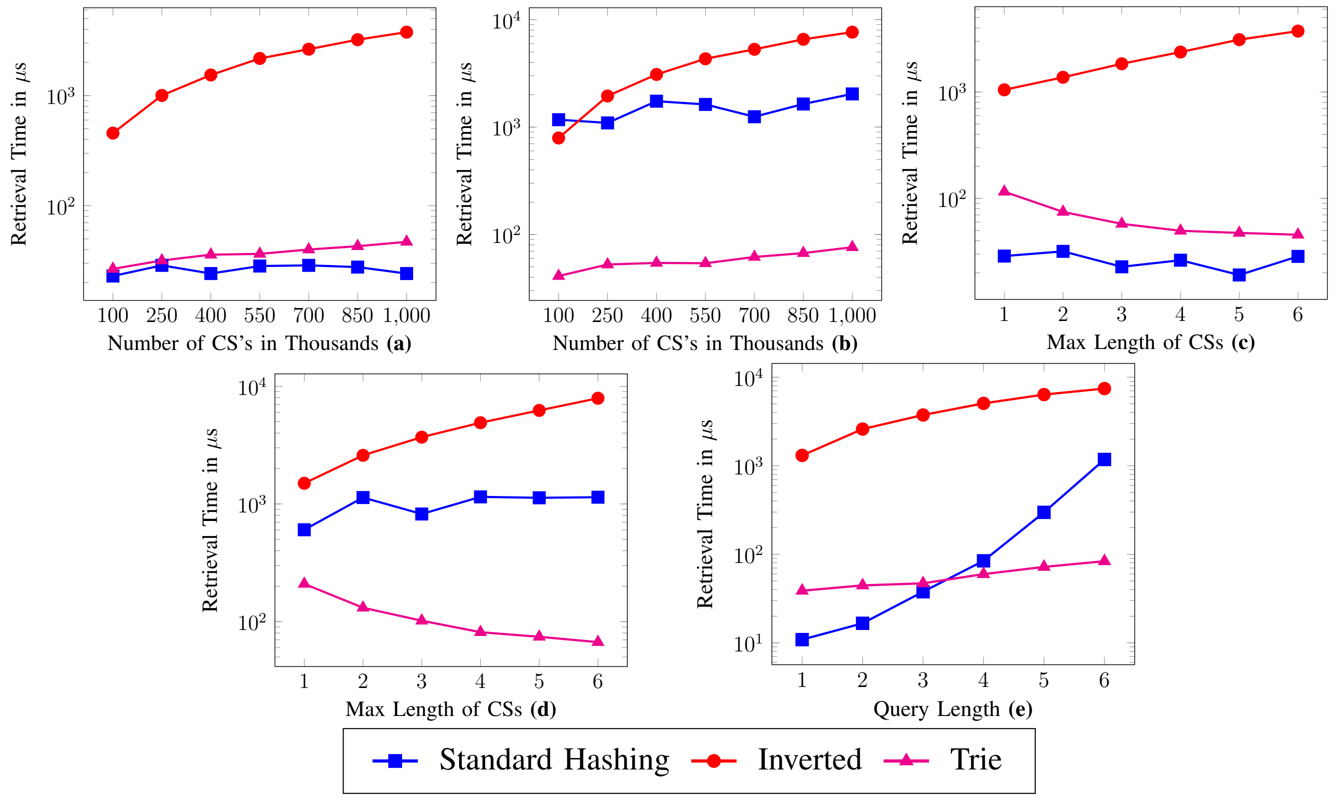


Fig. 1. Increasing the number of completeness statements for short (a) and long queries (b); Increasing maximum length of completeness statements for short (c) and long queries (d); Increasing the query length (e)

These graphs demonstrate the fundamental difference between the inverted indexes and the tries. In the inverted indexes, having a single overlapping constant is enough for a 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 constants 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 nearly constant for standard hashing, is likely due to evaluating always the same set equality queries.

3) Query Length: In this scenario, we vary the query length L_q from 1 to 6. Figure (1.e) shows the results. 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. At the beginning, the standard hashing technique performs better than the tries one. However, from $L_q = 4$ the standard hashing technique starts to perform worse. At $L_q = 6$, standard hashing is about $14\times$ slower than tries. We observe a similarity between the asymptotic growth of inverted indexing and tries, though on an absolute scale the tries technique clearly performs better.

As expected, standard hashing does not perform well for long queries due to its exponentially many set equality queries. The tries technique, though potentially having exponential growth in the worst case, performs better than standard hashing. This is most likely due to its pruning ability over subsequences of query constants.

4) Reasoning with the Constant-Relevant Filtering: This scenario differs from the above in that now we compare the cost of completeness reasoning without and with the optimization technique. We show that applying the constant-relevance principle can considerably reduce the overhead incurred by completeness reasoning.

To measure this overhead, we perform experiments that compare the runtimes of plain completeness reasoning and of reasoning based on constant-relevance. For the reasoning based on constant-relevance, we use the standard hashing retrieval technique as it shows relatively good performance in our previous experiments. All the parameter values are the default ones: $N_c = 1{,}000{,}000$, and $L_c = 6$, while we still distinguish between short queries ($L_q = 3$) and long queries ($L_q = 6$). The table below reports the reasoning time for plain completeness reasoning and the reasoning plus the retrieval time for the completeness reasoning based on constant-relevance.

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

We note that reasoning based on constant-relevance is considerably faster than the plain one (i.e., milliseconds vs. minutes, respectively). Completeness reasoning with the constant-relevance principle is fast, with runtimes between 110,000× (for short queries) and 35,000× (for long queries) faster than that without constant-relevance. This is due to the fact that much fewer completeness statements are considered for the reasoning using the constant-relevance principle. For short queries, there are on average about 49 constant-relevant completeness statements, whereas for long queries, there are on average about 105 constant-relevant statements. On the other hand, the original set contains 1 million completeness statements.

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 for both short and long queries. Moreover, on an absolute scale, completeness reasoning with the constant-relevant retrieval techniques only takes up to about a few milliseconds.

V. Conclusions

We presented techniques for efficient completeness reasoning over large sets of statements based on the constant-relevance principle to rule out a significant number of irrelevant completeness statements. We developed retrieval techniques for constant-relevant statements based on different index structures. 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 the project TaDaQua, funded by the Free University of Bozen-Bolzano.

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] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann, "Dbpedia - A crystallization point for the web of data," *J. Web Sem.*, vol. 7, 2009.

[14] 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.