Efficient Identification of Implicit Facts in Incomplete OWL Knowledge Bases

John Liagouris
National Technical University of Athens
liagos@dblab.ece.ntua.gr

Manolis Terrovitis
Institute for the Management of Information Systems
Research Center "Athena"
mter@imis.athena-innovation.gr

28 Feb 2014

Abstract

Integrating incomplete and possibly inconsistent data from various sources is a challenge that arises in several application areas, especially in the management of scientific data. A rising trend for data integration is to model the data as axioms in the Web Ontology Language (OWL) and use inference rules to identify new facts. Whereas there are several approaches that employ OWL for data integration, there is little work on scalable algorithms able to handle large datasets that do not fit in main memory.

The main contribution of the paper is an algorithm that allows using OWL rules for integrating data in an environment with limited memory. We propose a technique that exhaustively applies a set of inferences rules on large disk-resident datasets. To the best of our knowledge, this is the first work that proposes an I/O-aware method for such an expressive subset of OWL. Previous approaches considered either simpler models (e.g. RDF) or main memory algorithms. In the paper we detail the proposed algorithm, prove its correctness, and experimentally evaluate it on real and synthetic data.

1 Introduction and Motivation

In many application areas there is a need to integrate or curate incomplete data using expressive rules whose evaluation cannot be easily accommodated in relational databases. Representative examples often arise in the field of scientific data management. The state-of-the-art practice for addressing such problems is to model the data with the Web Ontology Language (OWL) [9], a standard of W3C. OWL extends the Resource Description Framework Schema (RDFS) [8] and allows the definition of assertions, constraints, classifications and taxonomies in the form of axioms which are amenable to automated reasoning procedures. Through the latter, one can extract new facts and dependencies or even identify inconsistencies. Over the last few years, OWL has been the basis for a multitude of scientific ontologies like SNOMED CT [14], GALEN [12], FMA [11], NCI Thesaurus [13], etc., most of which are actively maintained and widely used by practitioners and researchers in the respective fields.

Our work focuses on the efficient evaluation of complex inference rules on large sets of ontological axioms that cannot fit in main memory. Axioms in this setting describe the data and their schema, whereas the inference rules define recursive procedures that infer additional facts. For example, assume that the information "John is infected with virus A" is stored in one data source, and in another source there is the information that (i) "John is infected with virus B" and (ii) "Those that are infected by both virus A and B become ill". If we integrate data from both sources, we should also be able to infer, and add to the final dataset, that "John is ill". OWL enables users to specify such axioms, and facilitates data integration by offering reasoning mechanisms for extracting implicit facts. Given an initial set of axioms and a set of inference rules, the identification of all possible axioms with respect to the rules is known as the computation of the *logical closure* [44].

Reasoning with expressive rules on large and complex OWL ontologies is attracting significant interest in data management. OWL is already being used as a tool for integrating data of any type [23], including relational data [42]. The adoption of OWL as a mechanism for data integration has been further motivated by the spread of *Linked Data* and the support of OWL entailments in the SPARQL language [10]. This trend implies that OWL reasoning is not only needed for scientific ontologies which may be small in size (and relatively static), but it has to be performed also on huge volumes of operational data from various sources. The previous need is clearly reflected in the increasing support of OWL features by commercial RDBMSs like Oracle [45, 26, 32, 21] and IBM [26]. In this context, the inference tasks must be performed in an I/O-aware environment where main memory is not infinite. Unfortunately, the state-of-the-art systems with adequate inference capabilities from the areas of logic programming, deductive databases and semantic web, e.g., YAP [7], DLV [3], LogicBlox [4], OWLIM [21], Jena [1], etc., are either memory-oriented (and, hence, they cannot scale to large collections of axioms) or they focus on the evaluation of (partially) bounded queries, i.e., queries that retrieve information associated with a specific entity in the data, e.g., "Find all medical conditions associated with John".

The contributions of the paper are summarized in the following:

1. We model the logical closure computation as a reachability problem on a graph that represents the axioms of the ontology, and we propose a semantically oblivious storage scheme that facilitates the in-bulk application of different inferences within the same I/O operations. The core idea behind our approach is to establish a rule-independent pattern for accessing the data and decide on the fly which rule to perform.

2. We develop a novel method which operates efficiently under a limited memory budget and computes the logical closure within a series of simple database

operations like sort, merge and join. We also show how logical and physical optimizations can be incorporated in our evaluation scheme.

3. We provide a proof of correctness for the proposed algorithm and its optimizations.

4. We evaluate our techniques using real and synthetic datasets.

2 Problem Definition

OWL, in its complete form, is very expressive and many important reasoning tasks have exponential complexity with respect to the size of the data. For this reason, most existing datasets are expressed in tractable subsets of the language, namely OWL2-EL, OWL2-RL and OWL2-QL, that have "good" computational properties. In this work, we focus on a large subset of OWL2-EL, the *nominal-safe* $\mathcal{SROEL}$ [34]. $\mathcal{SROEL}$ supports all features of OWL2-EL except *admissible range restrictions*, *keys* and *datatypes*. We do not address the previous features here since they are not popular in practice.

For the ease of presentation, the syntax we use here is that of Description Logic (DL) [18], i.e., the logical formalism behind OWL, and not the syntax of OWL itself. The notation employed in the paper is depicted in Table 1. *Individuals* represent the actual data (instances) and *classes* have the usual semantics of collections of individuals. *Roles* (also known as *properties*) are relations that associate individuals. The existential restriction on a class C with a role R (denoted with $\exists R.C$) is a class itself; $\exists R.C$ stands for the class of all individuals which are associated with role R to at least one individual of class C. We refer to $C_1 \sqcap C_2$ and $\exists R.C$ as complex classes. The $\exists R.Self$ denotes the class of all individuals which are related through role R with themselves and it is used to express reflexivity.

$\mathbf{1}.C_1 \sqsubseteq C_2$	$\mathbf{2}.C_1 \sqcap C_2 \sqsubseteq C_3$	$\mathbf{3}.C_1 \sqsubseteq \exists R.C_2$	$\mathbf{4}.\exists R.C_1 \sqsubseteq C_2$
$\mathbf{5}.\{a\} \sqsubseteq \{b\}$	$\mathbf{6}.\{a\} \sqsubseteq C$	$\mathbf{7}.C \sqsubseteq \exists R.Self$	$\mathbf{8}.\exists R.Self \sqsubseteq C$
$\mathbf{9}.\top \sqsubseteq C$	$\mathbf{10}.C \sqsubseteq \bot$	$\mathbf{11}.R_1 \sqsubseteq R_2$	$\mathbf{12}.R_1 \circ R_2 \sqsubseteq R_3$

Figure 1: Normal Forms of Axioms

Ontologies in $\mathcal{SROEL}$ are finite collections of inclusion axioms depicted in Fig. 1. Axioms of types 1-4 model the inclusions between simple and complex classes. An axiom of type 5 expresses *equality* between individuals, i.e., the nominal class $\{b\}$ is a subclass of $\{a\}$ (and the reverse) iff $a = b$. An axiom of type 6 is used to model class assertions, i.e., a class assertion is expressed as an inclusion of the form $\{a\} \sqsubseteq$ C. Similarly, a property assertion that associates a with b through role R is equivalent to the axiom $\{a\} \sqsubseteq \exists R.\{b\}$ (type 3). Axioms of types 7 and 8 model inclusions involving the reflexive class of individuals, whereas axioms of types 9 and 10 model inclusions involving the top and bottom (empty) class. The bottom class is used to declare disjoint classes as $C_1 \sqcap C_2 \sqsubseteq \bot$ and inconsistencies as $\{a\} \sqsubseteq \bot$ (which often arise through

Name	Symbol	Meaning
Top	$\top$	The class that contains all individuals of the ontology
Bottom	$\bot$	The empty class
Class Conjuction	$C_1 \sqcap C_2$	The class that contains all individuals that belong to both C_1 and C_2
Existential Restriction	$\exists R.C$	The class that contains all individuals related with an individual of class C through property R
Reflexivity	$\exists R.Self$	The class that contains all individuals related with themselves through property R
Nominal	$\{a\}$	The class that contains only the individual a
Class Inclusion (SubClassOf)	$C_1 \sqsubseteq C_2$	If an individual is of type C_1 then it is also of type C_2
Property Inclusion	$R_1 \sqsubseteq R_2$	If two individuals are related through property R_1 then they are also related through property R_2
Complex Property Inclusion	$R_1 \circ R_2 \sqsubseteq R_3$	If an individual a is related with b through R_1 and b is related with c through R_2 then a is related with c through R_3

Table 1: Symbols and Terminology

the inference). Finally, axioms of types 11 and 12 denote the inclusions between roles. Note that a class C can be a nominal class in the axioms of Fig. 1 but not when it appears on the right side of an axiom; the right part of an inclusion can be a nominal $\{a\}$ only when the left part is also a nominal (type 5). Ontologies with this restriction are found in the literature as *nominal-safe* [30]. The reason we distinguish axioms of types 5 and 6 from those of type 1 is because there are inference rules that apply specifically on them (we explain this below).

Let us consider the small ontology of Fig. 2. *NotVaccinated, InfectedWithVirusA, Ill, VaccineTypeX* and *Vaccinated1994* are simple classes, *InfectedWithVirusA* $\sqcap$ *Not-Vaccinated* and $\exists$*Vaccinated. VaccineTypeX* are complex classes, *Vaccinated* is a role, and *john* and *va* are individuals (expressed as nominal classes). Assume that axioms 1, 2, and 3 come from a data source A which contains a series of scientific facts. Axiom 1 states that patients who are infected with Virus A (*InfectedWithVirusA*) and are not vaccinated (*NotVaccinated*) are ill (*Ill*). Axiom 2 states that all those who have been vaccinated with an inefficient vaccine of type *VaccineTypeX* ($\exists$*Vaccinated.VaccineTypeX*) should be treated as not vaccinated. Finally, axiom 3 denotes that the vaccine *va* is of type *VaccineTypeX*. Now consider that axioms 4 and 5 come from a data source B that has the medical history of patients and states that *john* belongs to a group of people who have been vaccinated in 1994 (*Vaccinated1994*) and that he is infected with virus A. Axiom 6 is added by an expert and states that all people who had been vaccinated

Source A	1. $\mathit{InfectedWithVirusA} \sqcap \mathit{NotVaccinated} \sqsubseteq \mathit{Ill}$
Source A	2. $\exists \mathit{Vaccinated}.\mathit{VaccineTypeX} \sqsubseteq \mathit{NotVaccinated}$
Source A	3. $\{\mathit{va}\} \sqsubseteq \mathit{VaccineTypeX}$
Source B	4. $\{\mathit{john}\} \sqsubseteq \mathit{Vaccinated1994}$
Source B	5. $\{\mathit{john}\} \sqsubseteq \mathit{InfectedWithVirusA}$
Expert	6. $\mathit{Vaccinated1994} \sqsubseteq \exists \mathit{Vaccinated}.\{\mathit{va}\}$

Figure 2: Integration example

in 1994 where vaccinated with vaccine va. From the previous example it is easy to see that additional data can be inferred and added into the integrated dataset; from the data of the two sources we can infer that John is ill, since he has been infected with virus A and he has been vaccinated in 1994 when everyone was vaccinated with the inefficient vaccine *va* of type *VaccineTypeX*. The detection of all such additional data or the existence of possible inconsistencies and discrepancies between data from different sources is achieved through the iterative application of a set of inference rules.

For $\mathcal{SROEL}$, these rules are depicted in Fig. 3. When the axioms that match the body of a rule (antecedent) are found in a collection of axioms, let D, then the axiom in the head of the rule (consequent) is true and can be added to D. The exhaustive application of the rules until no additional axioms can be produced, i.e., until no axioms that are not already in D are inferred (fix-point), is known as the computation of the logical closure. The result of this process is a dataset D' such that $D \subseteq D'$.

IR1 $C_1 \sqsubseteq C_3 \leftarrow C_1 \sqsubseteq C_2 \ \wedge \ C_2 \sqsubseteq C_3$
IR2 $C_1 \sqsubseteq C_4 \leftarrow C_1 \sqsubseteq C_2 \ \wedge \ C_1 \sqsubseteq C_3 \ \wedge \ C_2 \sqcap C_3 \sqsubseteq C_4$
IR3 $C_1 \sqsubseteq \exists R.C_3 \leftarrow C_1 \sqsubseteq C_2 \ \wedge \ C_2 \sqsubseteq \exists R.C_3$
IR4 $C_1 \sqsubseteq C_4 \leftarrow C_1 \sqsubseteq \exists R.C_2 \ \wedge \ C_2 \sqsubseteq C_3 \ \wedge \ \exists R.C_3 \sqsubseteq C_4$
IR5 $C_1 \sqsubseteq \exists R_2.C_2 \leftarrow C_1 \sqsubseteq \exists R_1.C_2 \ \wedge \ R_1 \sqsubseteq R_2$
IR6 $C_1 \sqsubseteq \exists R_3.C_3 \leftarrow C_1 \sqsubseteq \exists R_1.C_2 \ \wedge \ C_2 \sqsubseteq \exists R_2.C_3 \ \wedge \ R_1 \circ R_2 \sqsubseteq R_3$
IR7 $C_1 \sqsubseteq \bot \leftarrow C_1 \sqsubseteq \exists R.C_2 \ \wedge \ C_2 \sqsubseteq \bot$
IR8 $C_1 \sqsubseteq \exists R_3.C_2 \leftarrow C_1 \sqsubseteq \exists R_1.Self \ \wedge \ C_1 \sqsubseteq \exists R_2.C_2 \ \wedge \ R_1 \circ R_2 \sqsubseteq R_3$
IR9 $C_1 \sqsubseteq \exists R_3.C_2 \leftarrow C_1 \sqsubseteq \exists R_1.C_2 \ \wedge \ C_2 \sqsubseteq \exists R_2.Self \ \wedge \ R_1 \circ R_2 \sqsubseteq R_3$
IR10 $C \sqsubseteq \exists R_3.C \leftarrow C \sqsubseteq \exists R_1.Self \ \wedge \ C \sqsubseteq \exists R_2.Self \ \wedge \ R_1 \circ R_2 \sqsubseteq R_3$
IR11 $C_1 \sqsubseteq \exists R.Self \leftarrow C_1 \sqsubseteq C_2 \ \wedge \ C_2 \sqsubseteq \exists R.Self$
IR12 $C_1 \sqsubseteq \exists R_2.Self \ \leftarrow C_1 \sqsubseteq \exists R_1.Self \ \wedge \ R_1 \sqsubseteq R_2$
IR13 $C_1 \sqsubseteq C_2 \leftarrow C_1 \sqsubseteq \exists R.Self \ \wedge \ \exists R.Self \sqsubseteq C_2$
IR14 $C_1 \sqsubseteq C_3 \ \leftarrow C_1 \sqsubseteq \exists R.Self \ \wedge \ C_1 \sqsubseteq C_2 \ \wedge \ \exists S.C_2 \sqsubseteq C_3$
IR15 $\{a\} \sqsubseteq \exists R.Self \leftarrow \{a\} \sqsubseteq \exists R.\{a\}$
IR16 $\{b\} \sqsubseteq \{c\} \leftarrow \{a\} \sqsubseteq \{b\} \ \wedge \ \{a\} \sqsubseteq \{c\}$

Figure 3: Inference Rules ($\wedge$ stands for logical AND)

In the following we describe each inference rule of Fig. 3:

- Rule **IR1** states that if a class C_1 is a subclass of C_2 and C_2 is a subclass of C_3, then C_1 is also a subclass of C_3.

- Rule **IR2** states that if a class C_1 is a subclass of C_2 and C_3, and the intersection of C_2 and C_3 is a subclass of C_4, then C_1 is also a subclass of C_4 (since it belongs to both C_2 and C_3).

- Rule **IR3** states that if a class C_1 is a subclass of C_2 and all individuals in C_2 are related through role R with at least one invidual of C_3, then all individuals of C_1 are also related through role R with at least one individual of C_3 ($C_1 \sqsubseteq \exists R.C_3$).

- Rule **IR4** states that if (i) all individuals of C_1 are related through role R with at least one individual of C_2 which is a subclass of C_3, and (ii) all individuals which are related through R with at least one individual of C_3 are of type C_4, then we can infer that C_1 is a subclass of C_4 (all of its individuals are of type C_4).

- Rule **IR5** states that if all individuals of a class C_1 are related through role R with at least one individual of C_2, then they will be related with at least one individual of C_2 through all superroles of R. This inference is based on the definition of subroles (see Table 1).

- Rule **IR6** states that if (i) all individuals of a class C_1 are related through role R_1 with at least one individual of C_2, (ii) all idividuals of C_2 are related through R_2 with at least one individual of C_3, and (iii) the role chain $R_1 \circ R_2 \sqsubseteq R_3$ exists in the ontology, then we can infer that all individuals of C_1 are related through R_3 with at least one individual of C_3. This inference comes from the definition of role chains (see Table 1).

- Rule **IR7** states that if all individuals of a class C_1 are related through R with at least one individual of C_2, and C_2 is a subclass of $\bot$, i.e., it cannot contain any individuals, then C_1 must also be empty. This inference comes from the definition of the $\bot$ class and it helps identifying contradictions in the ontology.

- Rule **IR8** states that if (i) each one of the individuals of C_1 is related with itself through R_1, (ii) all individuals of C_1 are related through R_2 with at least one individual of C_2, and (iii) the role chain $R_1 \circ R_2 \sqsubseteq R_3$ exists in the ontology, then all individuals of C_1 are also related through R_3 with at least one individual of C_2.

- Rule **IR9** states that if (i) all individuals of C_1 are related through R_1 with at least one individual of C_2, (ii) each one of the individuals of C_2 is related with itself through R_2, and (iii) the role chain $R_1 \circ R_2 \sqsubseteq R_3$ exists in the ontology, then all individuals of C_1 are also related through R_3 with at least one individual of C_2.

- Rule **IR10** states that if (i) each one of the individuals of C is related with itself through R_1 and also through R_2, and (ii) the role chain $R_1 \circ R_2 \sqsubseteq R_3$ exists in the ontology, then all individuals of C are also related through R_3 with at least one individual of C.

- Rule **IR11** states that if a class C_1 is a subclass of C_2 and each individual in C_2 is related through role R with itself, then each individual of C_1 is also related through role R with itself.

- Rule **IR12** states that if each individual of class C_1 is related through role R with itself, then it is also related with itself through all superroles of R.

- Rule **IR13** states that if (i) each individual of C_1 is related through role R with itself, and (ii) all individuals which are related through R with themselves are of type C_2, then we can infer that C_1 is a subclass of C_2 (all of its individuals are of type C_2).

- Rule **IR14** states that if (i) each individual of C_1 is related through role R with itself, (ii) C_1 is a subclass of C_2, and (iii) all individuals which are related through R with at least one individual of C_2 are of type C_3, then we can infer that C_1 is a subclass of C_3 (all of its individuals are of type C_3).

- Rule **IR15** states that if an individual is related with itself through role R, then it belongs to the class that contains all individuals which are related with themselves through R.

- Rule **IR16** states that if a nominal $\{a\}$ is defined as a subclass of a nominal $\{b\}$ and also of a nominal $\{c\}$, then $\{b\}$ is also a subclass of $\{c\}$ (and the reverse). This rule is used to capture the equivalence relation between individuals ($a = b$) which may be defined in the ontology as $\{a\} \sqsubseteq \{b\}$ or $\{b\} \sqsubseteq \{a\}$ or even with both these axioms (redundancy).

Rules like **IR1**, **IR3** and **IR11** define (generalized) transitive closures [28], and if the logical inferences were limited to them, existing evaluation methods from database research would be sufficient to address the problem [44]. The complexity of inferences is increased by the fact that the majority of the rules are mutually recursive [20], either directly (when a rule creates axioms that participate in the body of another rule and vice versa, e.g, **IR3** and **IR4**), or indirectly (when a rule affects another through a third rule and vice versa, e.g., rules **IR1** and **IR6** through **IR3** and **IR4**). In addition, rules like **IR2** introduce more complex dependencies. Although simple transitive closure can be evaluated in a single pass over the data [15], such complex reasoning operations incur multiple passes; hence, for large-scale ontologies with millions of axioms (implicit or explicit), the problem becomes I/O-bounded.

Discussion. This work focuses on the logical closure of the rules in Fig. 3, and in the rest of the paper we only discuss its efficient evaluation in an environment with limited memory. Still, the rules we present here are tightly associated with a very important reasoning task in OWL which is known as *classification* [18, 17]. Classification amounts to the identification of all direct inclusions between the named classes of the ontology, e.g., *NotVaccinated, InfectedWithVirusA, Ill, VaccineTypeX* and *Vaccinated1994* in the example of Fig. 2. Besides the closure computation, this process requires two more steps in order to be performed correctly: a) the *normalization* of the input axioms that is performed before applying the rules, and b) the final *reduction* phase that extracts the direct class inclusions from the closure. The most expensive part of classification is by far the computation of the logical closure, hence, the results of our work can be also exploited in this setting. To this end and because the rules of Fig. 3 are complete with respect to classification only when the dataset is normalized, we bring the input axioms in normal form before applying the rules. In fact, Fig. 1 depicts the normalized axioms, that is, each symbol C_i refers to a class name (and not a complex class) like in the example of Fig. 2. For more details about the normalization and the transitive reduction phases, see [33] and [31].

3 Data modeling

Our approach adopts a semantically oblivious representation of the ontology based on a graph model, i.e., we model D as a graph with a small collection of metadata, and then store its edges in a relational table. Specifically, the majority of axioms in D, that is, all axioms except those of type 2, 11 and 12, are modeled as a directed labeled multigraph $G(V, E, l_V, l_E)$, where:

- V is the set of nodes. There is one node for each class in D, including $\top$ and $\bot$. We also consider a special node having the label Self.
- E is the set of edges. There is one edge for each (different) axiom in D except for axioms of type 2, 11 and 12.
- $l_V = N_C \cup\ N_I \cup\ \{\mathsf{Self}\} \cup \{\mathsf{Top}\} \cup \{\mathsf{Bottom}\}$ is the set of node labels. N_C and N_I are the sets of class and individual names in D. Top and Bottom are the labels of the nodes referring to the classes $\top$ and $\bot$ respectively.
- $l_E = N_{R^-} \cup N_{R^+} \cup \{\mathsf{subClassOf}\}$ is the set of edge labels. N_R^+ is a set of labels produced by the concatenation of a property name with the symbol $^+$. Such labels are created for each property R appearing in an axiom of type 3 or 7. N_{R^-} is produced similarly for the properties appearing in axioms of type 4 or 8. The use of subClassOf label is explained below.

Intuitively, for each class that appears in D we create a node in G having as label the name of the class. Then, for each axiom we add an edge to G, depending on the axiom type (T_x), as follows:

T_1 $C_1 \sqsubseteq C_2$ is represented by a subClassOf edge from node C_1 to node C_2.

T_3 $C_1 \sqsubseteq \exists R.C_2$ is represented by an edge from from node C_1 to node C_2, marked with label R^+.

T_4 $\exists R.C_1 \sqsubseteq C_2$ is represented by an edge from node C_1 to node C_2, marked with label R^-.

T_5 $\{a\} \sqsubseteq \{b\}$ is represented by a subClassOf edge from node a to node b.

T_6 $\{a\} \sqsubseteq C$ is represented by a subClassOf edge from node a to node C.

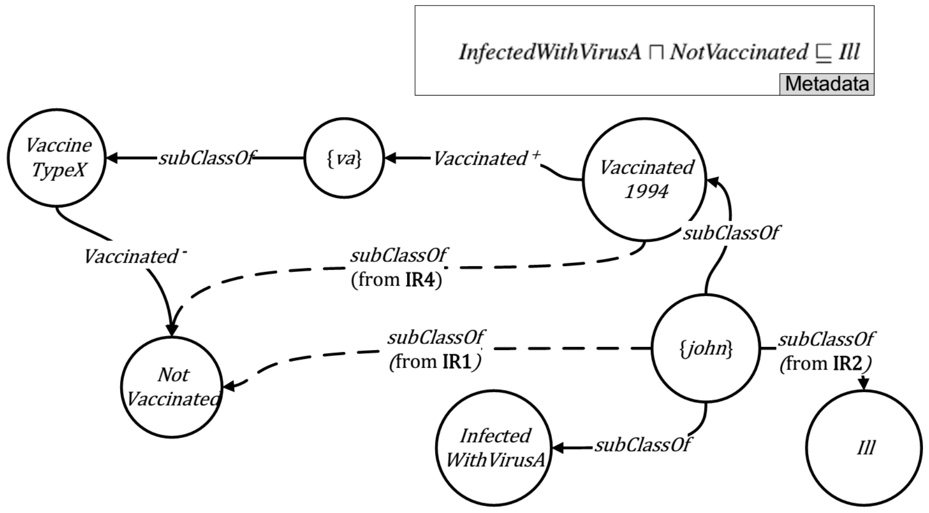


Figure 4: Graph of Running Example

T_7 C $\sqsubseteq \exists R.Self$ is represented by an edge from node C_1 to the unique node Self, marked with label R^+.

T_8 $\exists R.Self \sqsubseteq C$ is represented by an edge from the unique node Self to C, marked with label R^-.

T_9 $\top \sqsubseteq C$ is represented by a subClassOf edge from the unique node Top to node C.

T_{10} $C \sqsubseteq \bot$ is represented by a subClassOf edge from node C to the unique node Bottom.

Axioms of type 2, 11 and 12 are not modeled directly in the graph; they are kept separately as metadata since their semantics are very different from those of the rest of the axioms and they cannot be represented in an intuitive way on the graph.

The graph for our running example is provided in Fig. 4. The metadata (axioms of type 2, 11 and 12) are depicted on the upper right corner. Each non-dashed edge corresponds to an explicit axiom, i.e., an axiom that exists in D from the beginning. The edges (axioms) added by the inference are shown with dashed lines. Next to each such edge we also denote the inference rule that created it. Each axiom, except axiom 1 which is represented in the metadata, introduces an edge between the nodes of the graph. Axioms 3,4 and 5 introduce subClassOf edges, while axiom 2 introduces a *Vaccinated*$^-$ edge (since *Vaccinated* appears in the left side of the inclusion in axiom 2). Axiom 6 introduces a *Vaccinated*$^+$ edge. Using **IR4** on axioms 6, 3 and 2 we get a new axiom *Vaccinated1994* $\sqsubseteq$ *NotVaccinated*, which is represented by a dashed edge on the graph. This axiom, together with axiom 4 satisfy **IR1**, so an additional axiom $\{john\} \sqsubseteq$ *NotVaccinated* is added. This last axiom, along with axioms 1 and 5, satisfies **IR2**, so a final axiom $\{john\} \sqsubseteq$ *Ill* is added to the original data. We point out that each inference rule can only add edges to the graph G, so the set V of nodes remains the same and no additional axioms are added to the metadata.

Within our model, most inference rules of Fig. 3 can be evaluated by considering only: (i) the incoming and outgoing edges of each node, (ii) the source and end nodes of these edges, and (iii) the metadata of G (depicted on the up-right part of Fig. 4). In other words, rule application is reduced into examining at most 2-hop paths in G. For instance, **IR6** requires two hops: from C_1 to C_2 through R_1 and from C_2 to C_3 through R_2 (the axiom $R_1 \circ R_2 \sqsubseteq R_3$ of type 12 belongs to the metadata). The only exception is rule **IR4** which requires 3 hops: from C_1 to C_2 through R, from C_2 to C_3 through subClassOf, and from C_3 to C_4 through R^-. The aforementioned property is crucial for the design of an I/O-aware evaluation algorithm; it implies that, if we traverse the graph node by node (i.e., retrieve all edges associated with a specific node), we can correctly produce all axioms implied by the rules of Fig. 3 in this part of the graph. In other words, a single application of all inference rules on the dataset can be consistently performed within the same scan, provided that this scan proceeds node by node. To compute the complete logical closure, the new edges have to be added to the graph and the process must be repeated until no new edges are created.

3.1 Storage Scheme

The graph G and its metadata are stored in five relations:

R_G Relation R_G contains all edges that comprise the graph G except those that correspond to axioms of type 4 and 8. R_G has four fields: (i) R which contains the edge label, (ii) C_1 which stands for the source node, (iii) C_2 which stands for the target node, and (iv) a 4-bit field T which contains additional information about the edge. The three less important bits (2^3=8) in T are used for denoting the type of the axiom the edge corresponds to, whereas the more important bit is used by the algorithm as we explain in the following sections[1]. In practice, relation R_G contains the largest part of the data. It is also the only relation that is expanded with new tuples during the logical closure computation.

R_2 Relation R_2 contains all axioms of type 2 ($C_1 \sqcap C_2 \sqsubseteq C_3$). It has three fields, one for each class that appears in the axiom: (i) C_1 for the first conjunction class, (ii) C_2 for the second conjunction class, and (iii) C_3 for the subsumer class.

R_{11} Relation R_{11} stores all axioms of type 11 ($R_1 \sqsubseteq R_2$) and has two fields: (i) R_1 for the subsumee, and (ii) R_2 for the subsumer property.

R_{12} Relation R_{12} stores all axioms of type 12 ($R_1 \circ R_2 \sqsubseteq R_3$) and has three fields: (i) R_1 for the first property in the chain, (ii) R_2 for the second property, and (iii) R_3 for the subsumer property.

R_{48} Relation R_{48} stores all edges corresponding to axioms of type 4 and 8, i.e., $\exists R.C_1 \sqsubseteq C_2$ and $\exists R.Self \sqsubseteq C$. It has three fields: (i) R which is the edge

[1]Note that we only need 3 bits for the axiom types since four types of axioms in Fig. 1 are stored in seperate relations.

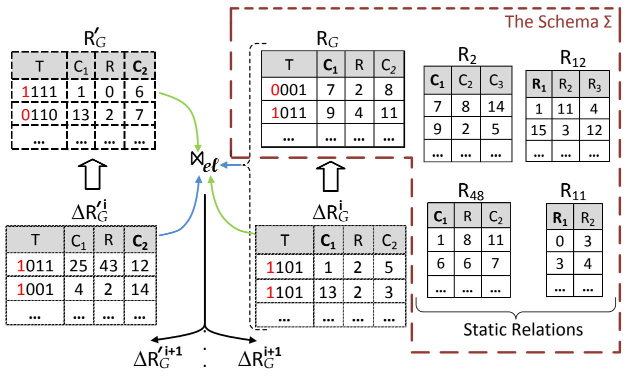


Figure 5: Data model for D

label, (ii) C_1 which is the source node (for axioms of type 8 this is the Self node), and (iii) C_2 which is the target node.

The overall schema Σ is illustrated in Fig. 5. Note that classes in all tuples are represented by integer IDs. The R'_G relation is temporary and it is created by the algorithm we present in Section 4. The schema of Fig. 5 is a 1-1 representation of the axioms of Fig. 1. This is straightforward for axioms of types 2, 11 and 12 which get separate tables. For axioms 4 and 8, the homogenous representation in table R_{48} leads to a 1-1 mapping since they are syntactically the same. For the rest of the axioms stored in R_G, the T attribute (which encodes the type of the axiom) guarantees that the mapping is 1-1.

The main motivation behind our algorithm and the proposed storage scheme is to reduce the I/Os and, especially, the random I/Os. Our goal is to access the data on disk as few times as possible and to perform these accesses mainly with sequential scans of the underlying relations. In Section 3, we noted that the inference rules can be completely evaluated by "looking" only in a neighborhood of the graph. We further observed that if we cluster all axioms that contain a certain class (i.e., all incoming and outgoing edges of a node in the graph), then we can evaluate most inference rules of Fig. 3 consistently. To this end, we propose storing most of the axioms represented by the graph in a single relation R_G which is sorted on the class IDs (R_G is sorted on C_1 whereas its copy R'_G is sorted on C_2 - cf. Fig. 5). This enables us to efficiently retrieve all edges that are associated with a class (i.e., a node) and perform on them all applicable rules in bulk. The previous property is exactly what we exploit in our approach; the algorith we present in Section 4 retrieves tuples from the sorted R_G (and its copy R'_G) in contiguous blocks (as many as the input buffers allow), and minimizes the random I/Os because all these blocks (probably except the first one) are fetched in memory with sequential accesses on disk. Moreover, note that the axioms we store in R_G (and R'_G) are those appearing in the head of at least one rule (the remaining axioms are kept in separate relations). Intuitively, this means that the algorithm will update only these two relations at each step of the iteration (with the use of deltas), minimizing in this way also the random writes to disk.

Still, some rules are not completely evaluated by accessing only R_G and R'_G. The rest of the relations in the schema of Fig. 5 (R_2, R_{48}, R_{11} and R_{12}) have to be accessed to retrieve axioms of types 2, 4, 8, 11 and 12. The reasons these axioms are kept in separate relations are the following:

- Rules **IR2**, **IR4**, **IR6**, **IR**8, **IR9**, **IR10** and **IR14** have three predicates in their bodies and their evaluation requires a 3-way join. Thanks to the additional static relations, the 3-way joins required by the previous rules are evaluated by performing only 1 join between the large relations R_G and R'_G; the additional join is always with one of the other small relations.
- Axioms stored in R_2 ($C_1 \sqcap C_2 \sqsubseteq C_3$), R_{11} ($R_1 \sqsubseteq R_2$) and R_{12} ($R_1 \circ R_2 \sqsubseteq R_3$) are not represented as edges in the graph, hence, they participate in joins which have very different join conditions from the joins between the rest of the axioms. All other axioms contain two classes (one on each side of $\sqsubseteq$) on which the relations R'_G and R_G are sorted. Axioms of R_2, R_{11} and R_{12} cannot benefit from such sorting, so inserting them in R_G and its copy R'_G would only make their retrieval more expensive.
- The axioms of type 2, 4, 8, 11 and 12 do not appear in the head of any rule. This means that keeping them separately from the rest of the axioms will not introduce additional random disk writes because the corresponding relations are static, i.e., they are not updated by the inference.

Note that traditional algorithms which evaluate each rule of Fig. 3 independently, access the underlying database on a per-rule basis and cannot benefit from the previous schema. At each iteration step, these algorithms examine only axioms that are related to a specific rule. In case they encounter other types of axioms in the relation they scan, they simply omit them. Hence, storing different types of axioms in the same relation (like in R_G) will only result in redundant I/Os for them.

4 Overview of the Algorithm

The salient feature of our approach is that the rules of Fig. 3 are evaluated in bulk over the schema Σ of Fig. 5. The proposed *Batch Rule Application* (BRA) algorithm is equivalent to a semi-naive evaluation [44] of each rule in isolation, i.e., the combinations of tuples which have been considered in a previous step of the iteration are never checked again under the same rule.

As mentioned in Section 3, applying the rules of Fig. 3 on G requires the examination of at most 2-hop paths (3-hop for **IR4**) and a lookup in the metadata. On the schema Σ, this can be done by performing (i) a self-join of the form $R_G \bowtie_{C_2=C_1} R_G$ (for identifying the 2-hop paths), and (ii) a subsequent join of the intermediate tuples with one of the static relations R_2, R_{11}, R_{12}, and R_{48} (depending on the particular rule). To avoid the increased random disk page accesses of the self-join, we adopt the following strategy. First, we ensure the semi-naive evaluation by keeping the new tuples in a temporary relation ΔR_G. Instead of inserting ΔR_G into R_G and then perform a self-join on R_G as a naive algorithm would do, we join ΔR_G with R_G and insert the

```
Algorithm: $BRA$
Input    : The schema $\Sigma$={$R_G, R_2, R_{11}, R_{12}, R_{48}$} populated with the initial tuples;
Output   : The logical closure of the rules in Fig. 3;

1  initialize relations;
2  let $\Delta R'^i_G$ be the left delta relation and $\Delta R^i_G$ be the right delta relation at step $i$ (see also
   Fig. 5);
3  $i = 0$;
4  $R'^i_G \bowtie_{el} R^i_G$ ;                         //output sent to $\Delta R^{i+1}_G$ and $\Delta R'^{i+1}_G$
5  while $|\Delta R'^{i+1}_G|\,!=0$ or $|\Delta R^{i+1}_G|\,!=0$ do
6      $i$++;
7      sort the right $\Delta R^i_G$ on $C_1$ and remove duplicates;
8      sort the left $\Delta R'^i_G$ on $C_2$ and remove duplicates;
       //remove old tuples from the right delta
9      $\Delta R^i_G \longleftarrow \Delta R^i_G \setminus R^{i-1}_G$;
       //remove old tuples from the left delta
10     $\Delta R'^i_G \longleftarrow \Delta R'^i_G \setminus R'^{i-1}_G$;
11     $R'^{i-1}_G \bowtie_{el} \Delta R^i_G$;                 //output sent to $\Delta R^{i+1}_G$and $\Delta R'^{i+1}_G$
12     merge $\Delta R^i_G$ with $R^{i-1}_G$ into $R^i_G$ so that the relation remains sorted on $C_1$;
13     $\Delta R'^i_G \bowtie_{el} R^i_G$ ;                 //output sent to $\Delta R^{i+1}_G$ and $\Delta R'^{i+1}_G$
14     merge $\Delta R'^i_G$ with $R'^{i-1}_G$ into $R'^i_G$ so that the relation remains sorted on $C_2$;
15 return $R^i_G$
```

first into the latter afterwards. Second, the join between ΔR_G and R_G is designed so that all rules are checked when a node neighborhood of G is fetched from disk. This means that the rules are applied in bulk within each scan of the dataset and we do not have to perform different scans for applying different rules. We provide details on this in Section 4.1. The overall procedure is depicted in Algorithm BRA. In the following we describe its high-level steps.

Initialization. The algorithm assumes a limited space of size M in main memory. First, it creates a copy R'_G of R_G on disk. This allows to avoid the expensive self-join on R_G and instead of it to perform an initial join between R'_G and R_G. As we show later on, the same replication is followed for the delta relation, that is, at each step of the iteration we have a copy of ΔR_G denoted as $\Delta R'_G$. In order to enable efficient merge-joins, R'_G is sorted on the attribute C_2 whereas R_G is sorted on C_1. Finally, a hash index is created for each static relation: R_2 is hashed on C_1, R_{11} on R$_1$, R_{12} on R$_1$, and R_{48} on C_1 (these attributes are highlighted in Fig. 5). Each one of the previous operations utilizes the entire available memory M. When the initialization is completed, M is used by the algorithm for fetching tuples of R_G, R'_G, ΔR_G, and $\Delta R'_G$ (all of which are disk resident) and for the output buffers.

Evaluation. The first step of the algorithm is to join the relations R_G and R'_G (**line 4**) according to the operator $\bowtie_{el}$ that applies all inference rules together. We term this join *el*-join and describe its details in Section 4.1. The new tuples created by the application of the rules are stored in ΔR_G (which is sorted on C$_1$) and $\Delta R'_G$ (which is sorted on C_2). In the baseline version of our algorithm, ΔR_G and $\Delta R'_G$ contain exactly the same tuples (sorted in a different order), but in Section 5 we show how we can prune tuples from the deltas. After the initial creation of ΔR_G and $\Delta R'_G$ in **line 4**, the algorithm starts its main loop (**lines 5-14**) which terminates when no new tuples are produced (condition in **line 5**). Since the main relations are updated at each iteration, we use R^i_G and R'^i_G to denote the relations R_G and R'_G at the i-th step of the algorithm. Similarly, ΔR^i_G and $\Delta R'^i_G$ are used for ΔR_G and $\Delta R'_G$.

Each step of the algorithm breaks into a sequence of simple database operations: (i) *sorting*, (ii) *set difference*, (iii) *join*, and (iv) *merging* as in a typical semi-naive evaluation. At step $i \geq 1$, R^{i-1}_G and R'^{i-1}_G are already sorted in the previous step $(i-1)$, so we only need to sort ΔR^i_G and $\Delta R'^i_G$. The latter are sorted on the attributes C_1 and C_2 respectively[2], and their duplicates (with respect to all attributes) are removed during this process (**lines 7-8**). Then, two set-difference operations are performed in order to remove the "old" tuples existing in the deltas: one between ΔR^i_G and R^{i-1}_G (**line 9**), and another one between $\Delta R'^i_G$ and R'^{i-1}_G (**line 10**). After that, ΔR^i_G is joined with R'^{i-1}_G (**line 11**), and then it is merged with R^{i-1}_G into the relation R^i_G (**line 12**) so that the latter remains sorted on C_1. Analogously, $\Delta R'^i_G$ is joined with R^i_G (**line 13**), and then it is merged with R'^{i-1}_G into the relation R'^i_G (**line 14**) so that the latter remains sorted on C_2. Note that $\Delta R'^i_G$ is joined with R^i_G, not R^{i-1}_G, which implies a join of $\Delta R'^i_G$ with both R^{i-1}_G and ΔR^i_G (see also Fig. 5). All tuples produced by the two *el*-joins are stored in the relations $\Delta R'^{i+1}_G$ and $\Delta R'^{i+1}_G$ which are used in the next step. When the loop is over, the logical closure of the rules is contained in R_G.

4.1 The *el*-join operator

The el-join is the core of the algorithm we propose. It is a complex operator which breaks the evaluation of each rule into smaller operations shared with other rules. Following a multi-query optimization paradigm [41], the common join predicates in the bodies of the rules are batched and evaluated all together in groups, so that a significant number of redundant I/Os is avoided. Conceptually, the application of a rule in this setting may be postponed till other rules are evaluated, and continue again later on as we scan the dataset. This aspect is very similar to the notion of eddies [16], however, the latter focus on join re-ordering whereas in our case the dynamic scheduling of the rules is simply determined by the type of the tuples fetched from disk.

The basic operations in the evaluation of the rules are the joins between the different relations. The most expensive joins are those involving R_G and R'_G. We have two types of such joins. The first is $R'_G \bowtie_{R'_G.C_2=R_G.C_1 \wedge R'_G.T=f(R_G.T)} R_G$ and the second is the self join $R_G \bowtie_{R_G.C_1=R_G.C_1 \wedge R_G.T=f(R_G.T)} R_G$, where f is a matching function that decodes the attribute T in order to determine the applicable rule. Intuitively, the first type of join creates *pairs of incoming and outgoing edges* of a node C_1, whereas

[2]In fact, the attributes C_1 and C_2 are only the primary attributes of sorting. Since we need to remove the duplicates as well, the remaining attributes of the relations are also taken into account in sorting but we omit them here to simplify the presentation.

the second type combines all *outgoing* edges of a node C_1. Based on the common operations in different rules, we partition the rules into the following classes:

- **Class 1** contains the rules **IR5**, **IR12** and **IR13**. To evaluate these rules we need to examine every edge labeled with a role R on the graph, and the complex roles from the metadata. This is translated into a join between R_G and the static relations R_{48} and R_{11}.

- **Class 2** contains the rule **IR15** that only needs to examine each node separately, so we only have to scan R_G.

- **Class 3** contains rules **IR2**, **IR8**, **IR10** and **IR14**. These rules require examining all the pairs of *outgoing* edges for each graph node C_1 and also the metadata. This is reflected to a self-join of the second type and joins with the static relations.

- **Class 4** contains rule **IR16** that requires examining all pairs of outgoing edges for each node, but not the metadata.

- **Class 5** contains rules **IR1**, **IR3**, **IR7** and **IR11**, which require examining every pair of incoming and outgoing edges for each node C_1. This is translated into a join of the first type between R_G and R'_G.

- **Class 6** contains the rules **IR4**, **IR6** and **IR9**. **IR4** requires examining chains of three edges in the graph where the last edge is of type 4 ($\exists R_G.C_1 \sqsubseteq C_2$) that is stored in R_{48}. **IR6** and **IR9** require the examination of all pairs of incoming and outgoing edges for a node and also the examination of the metadata for complex roles. In all cases, the evaluation requires a join of the first type between R_G and R'_G, and a join of the results with the static relations; with R_{48} for **IR4**, and with R_{12} for **IR6** and **IR9**.

The way the operator works is depicted in the Algorithm *el*-JOIN. The input relations S and P refer to the left and right relations in lines **4**, **11** and **13** of the algorithm BRA. Hence, S stands for one of R'^0_G, R'^{i-1}_G and $\Delta R'^i_G$, whereas P for one of R^0_G, ΔR^i_G and R^i_G. The output of the operator are the relations $\Delta R'^{i+1}_G$ and ΔR^{i+1}_G. Let U and Q be the in-memory buffers for the relations S and P respectively. In sum, the el-join applies the rules within a carefully designed merge-join between S and P (**lines 2-16**), and completes the evaluation with interleaving joins between streams of tuples and the static relations of Σ. Recall that the relations S and P are already sorted on the join attributes C_2 and C_1. The term *trigger* is used in the pseudocode for all rules that include an "external" join with the static relations (R_2, R_{11}, R_{12}, R_{48}). Triggering a rule means that the respective tuples are pushed to the sub-operators that perform the external joins. This is done directly as we scan relation P (for rules of Class 1) or after partially applying a rule of Class 3 and 5. In the former case, the tuples sent to the sub-operators come from P, whereas in the latter case they are intermediate tuples produced by the partial application of the rules. A rule is applied partially when some (and not all) of the axioms in its body are checked. For example, we say that **IR6** is partially applied when we perform only its first join ($C_1 \sqsubseteq \exists R_1.C_2 \ \wedge \ C_2 \sqsubseteq \exists R_2.C_3$)

as shown in Fig. 3. The remaining rules (Classes 2 and 4) are evaluated as a whole within the main merge-join.

```
Algorithm: el-JOIN
Input  : relations $S, P, R_2, R_{11}, R_{12}, R_{48}$;
Output : relations $\Delta R_G^{i+1}$ and $\Delta R_G^{\prime i+1}$;
vars   : $U$, $Q$: buffers;

let $U$ and $Q$ be the memory buffers for $S$ and $P$ respectively;
while there are tuples to join on $S.C_2 = P.C_1$ do
    identify the minimum $C_1$ value in $Q$, let $min_Q$;
    for each unconsidered tuple $t \in Q$:$t.C_1 = min_Q$ do
        trigger active rules of Class 1;                  //IR5, IR12, IR13
        apply active rules of Class 2;                                //IR15
    //self join on $P$
    for each unconsidered pair of tuples $(t_1, t_2)$:$t_1, t_2 \in Q$ and $t_1.C_1 = t_2.C_1 = min_Q$
    do
        trigger active rules of Class 3;             //IR2,IR8,IR10,IR14
        apply rules of Class 4;                                       //IR16
    //join $S$ and $P$
    for each unconsidered pair of tuples $(t_1, t_2)$:$t_1 \in U, t_2 \in Q$ and
    $t_1.C_2 = t_2.C_1 = min_Q$ do
        //read as many blocks of $S$ and $P$ needed
        apply rules of Class 5 ;                     //IR1, IR3, IR7, IR11
        trigger rules of Class 6;                          //IR4, IR6, IR9
    if no unconsidered tuples exist in $U$ and $Q$ then
        if $Q$ has to be reloaded and there are active rules in Class 3 then
            shift tuple(s) in $Q$ with the maximum $C_1$ value to the beginning of the
            buffer;
        reload $U$ and/or $Q$ with the next block(s) of tuples;
if there are active rules in Classes 1, 2, 3 and 4 then
    while $P$ is not exhausted do
        repeat lines 3-8;
        remove from $Q$ the tuples checked under all active rules in Classes 1, 2, 3 and 4;
        if there are active rules in Class 3 then
            shift tuple(s) in $Q$ with the maximum $C_1$ value to the beginning of the
            buffer;
        reload $Q$ with the next block(s) of tuples;
```

The operator starts by filling the buffers U and Q with tuples from S and P. It works on groups of tuples having a common value for the $Q.C_1$ attribute (min_Q) and proceeds as follows. First, it checks whether a rule of Class 1 should be triggered or applies rules of Class 2 (**lines 4-6**). Then, it partially applies rules of Class 3 by performing a self join on $Q.C_1$ (**lines 7-9**). Joins of the second type $P \bowtie_{P.C_1=P.C_1 \wedge P.T=f(P.T)} P$ are evaluated here and the intermediate tuples are pushed to the sub-operators as we mentioned previously. In the next step, it checks for rules of Class 5 and 6 (**lines 10-12**), so it evaluates the join patterns of the first type $S \bowtie_{S.C_2=P.C_1 \wedge S.T=f(P.T)} P$. Rules of Class 5 are applied as a whole, whereas those of Class 6 are applied partially (like the rules of Class 3) and the intermediate tuples are sent to the sub-operators. Similarly to a typical merge join algorithm, in **lines 10-12** we may need to read additional blocks of S and P in order to ensure that there are no more pairs of tuples (t_1, t_2) for which the condition $t_1.C_2 = t_2.C_1$ holds. The only difference with a typical merge join algorithm lies in the way the blocks of the two relations S and P are fetched from disk (**lines 14-15**); some tuples of P remain in the buffer (shifted to the beginning) even after a subsequent load operation is performed in Q (**line 14**). This happens only for the tuples with the maximum $Q.C_1$ value, and only when there are *active* rules that require a natural join on $Q.C_1$ (Class 3), so that the operator can consistently apply this join as a self join on the "sliding window" over P while the main merge-join proceeds as usual (no additional I/Os occur). We clarify this in the example provided below. To avoid naive evaluation, i.e., checking previously checked axiom combinations, we need to evaluate the rules of Classes 1 and 2 on the static relations of Σ only once. To this end, rules of these classes are *active* only in the initial step of BRA and they are deactivated in the following steps. A similar optimization is used for the rules of Class 3 to avoid the unnecessary join between $R'^{i-1}_G \bowtie_{el} \Delta R^i_G$. For the ease of presentation, we assume that the tuples in P (resp. S) with the same C_1 (resp. C_2) value fit always in Q (resp. U). Special cases where the large number of tuples requires a nested-loop join of the inner blocks are not addressed here.

When the main loop of the merge join is completed, unread tuples from P that need to be checked under the rules of Classes 1, 2, 3 or 4 are handled in **lines 17-23**. There, the remaining blocks of P are fetched from disk with sequential scans and the procedure in **lines 3-8** is repeated till the relation is exhausted. We emphasize that despite the predefined order for checking the different classes of rules in BRA, the rules are actually applied according to the types of the tuples in the buffers U and Q. Thus, the order of rules, as we scan S and P, is dynamic and driven by the underlying data.

Since external joins are needed in the rules of Classes 1, 3 and 6, the interleaving joins with the static relations of Σ occur in **lines 5, 8 and 12**. Each such join is performed between a stream of tuples and one of the relations R_2, R_{11}, R_{12}, and R_{48}. Streams for rules of Class 1 are populated with tuples from P in **line 5**. The respective streams for the rules of Class 3 and 6 are produced by the self join on $P.C_1$ in **line 8** and the join S $\bowtie_{S.C_2=P.C_1}$ P in **line 12**. Obviously, after a join is performed in the previous two cases, attributes which are no longer needed for producing the final tuples are discarded. A static relation is associated with exactly one stream, so the latter may contain tuples belonging to the workload of different rules. This amounts to a conceptual re-grouping of the rules at a second level, based on their common join patterns with the static relations. When an external join is performed, the final tuples are sent to the deltas.

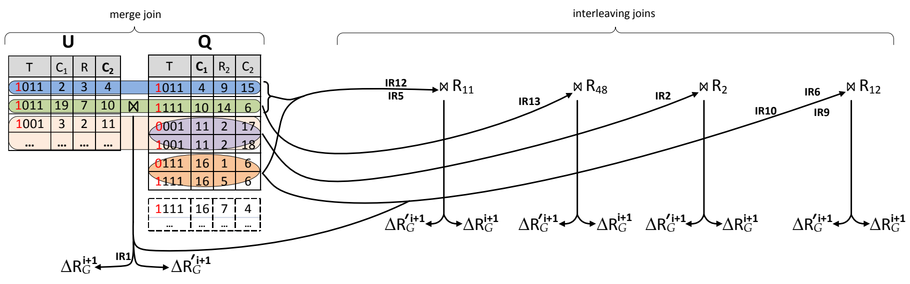


Figure 6: The el-join operator (U and Q are the "sliding windows" over S and P resp.)

Example. The way the rules are applied is highlighted in the following example. Recall that the external relations are kept in memory and that the last 3 bits of the T attribute in each tuple denote the type of the axiom. Assume a specific point of the evaluation where the buffers U and Q contain the tuples shown in Fig. 4.1. As implied by their types, these tuples must be checked under the rules **IR1**, **IR2**, **IR5**, **IR6**, **IR9**, **IR10**, **IR12** and **IR13**. Note that the previous rules involve all join patterns we have mentioned. The operator starts by identifying the tuple with the minimum value for $Q.C_1$ (min_Q = 4). It triggers rule **IR5**, that is, it pushes the tuple ⟨(T=3), 4, 9, 15⟩ to the stream of R_{11}. Then, it partially applies rule **IR6** between the tuples ⟨(T=3), 2, 3, 4⟩ and ⟨(T=3), 4, 9, 15⟩ ($C_1 \sqsubseteq \exists R_1.C_2 \wedge C_2 \sqsubseteq \exists R_2.C_3$), and pushes the intermediate tuple to the stream of R_{12}. Proceeding with the next min_Q value, it pushes ⟨(T=7), 10, 14, 6⟩ to the streams of R_{11} and R_{48} so that **IR12** and **IR13** are applied, and partially applies **IR9** between the tuples ⟨(T=3), 19, 7, 10⟩ and ⟨(T=7), 10, 14, 6⟩ ($C_1 \sqsubseteq \exists R_1.C_2 \wedge C_2 \sqsubseteq \exists R_2.Self$). The intermediate tuple is sent to the stream of R_{12} just like in the case of **IR6** before. Now, the next min_Q value is 11 and there are two tuples with this value in Q (both are of type 1). el-join partially applies **IR2** on these two tuples (i.e., the join $C_1 \sqsubseteq C_2 \wedge C_1 \sqsubseteq C_3$) and sends the intermediate tuple to the stream of R_2. It also applies **IR1** between these tuples and the tuple ⟨(T=1), 3, 2, 11⟩ of U. The output of **IR1** is directly sent to the deltas. Finally, let the last tuples of Q are ⟨(T=7), 16, 1, 6⟩ and ⟨(T=7), 16, 5, 6⟩, both of type 7 and with a common value for the C_1 attribute. These tuples belong to the workload of **IR10** and **IR12**. After the rules are applied similarly to the previous rules, there are no more unconsidered tuples in Q so the buffer must be loaded with the next tuples from P. Note that the next tuple from P in the example is ⟨(T=7), 16, 7, 14⟩ which is also of type 7 and must be checked under **IR10** with both ⟨(T=7), 16, 1, 6⟩ and ⟨(T=7), 16, 5, 6⟩ (for the join $C \sqsubseteq \exists R_1.Self \wedge C \sqsubseteq \exists R_2.Self$). For this reason, the last two tuples are not discarded from Q during the load operation; they are kept in the beginning of the buffer to ensure that **IR10** is evaluated correctly in the next step and with no additional I/Os.

5 Algorithm Optimizations

This section presents two important optimizations on the baseline algorithm BRA. The effectiveness of these techniques are highlighted in the experimental section.

5.1 Pruning tuples from the Deltas

In the general case, to guarantee that a rule has considered every combination of old and new axioms, the latter have to be inserted into both ΔR_G^i and $\Delta R_G'^i$. Intuitively, new axioms are additional edges to the graph of Section 3 and by adding them to both deltas, we ensure that all valid hops will be explored. Still, not every node and edge in the problem graph is the same; by closely examining the rules we can decide that some of the new edges do not need to be included in both deltas. This applies to edges corresponding to axioms of type 7 ($C \sqsubseteq \exists R.Self$) which are only considered together with other axioms that have a common C value. Based on this observation we can state the following Lemma (proof is given in Section 6):

Lemma 1 *The axioms produced by rules* **IR11***,* **IR12** *and* **IR15** *can be omitted from* R'_G *and* $\Delta R_G'^i$ *without altering the final result of the algorithm.*

Moreover, the rule **IR1** describes a classical transitive property, thus, it can be evaluated in a left- or right-linear fashion [44]. In our setting where all rules are evaluated together, this optimization requires special treatment because the tuples produced by **IR1** are actually updates in the workload of other rules; still, we prove in Section 6 that the following Lemma holds:

Lemma 2 *The axioms produced by rule* **IR1** *can be omitted from* $\Delta R_G'^i$ *without altering the final result of the algorithm.*

In other words, axioms of type 1 ($C_1 \sqsubseteq C_2$), are initially replicated to R_G and R'_G but in the rest of the algorithm they are only inserted in the right delta (ΔR_G^i).

5.2 Rule Application while Merging

Recall that the rules of Classes 1, 2, 3 and 4 require (i) a self-join on $R_G.C_1$, (ii) a join between R_G and a static relation of Σ or (iii) no join at all. Thus, they are applied while traversing the tuples of R_G in the merge join between $\Delta R_G'^i$ and R_G (**lines 1-9** and **17-23** in algorithm *el*-join). The evaluation of these rules within the el-join operator implies a certain inefficiency; although the merge join of $\Delta R_G'^i$ and R_G does not need to continue scanning the rest of R_G when $\Delta R_G'^i$ has run out of tuples, the operator exhaustively scans R_G (**lines 17-23**) because of the aforementioned rules. Based on this, we can improve the baseline algorithm by simply moving the evaluation of all rules belonging to Classes 1, 2, 3 and 4 to the *merge* phase of the algorithm BRA (**line 12**). There, no additional I/Os are required for the evaluation of these rules because all tuples from $\Delta R_G'^i$ and R_G are read and written back to disk anyway in order to update R_G.

When applying this optimization, **lines 4-9**, **14-15** and **17-23** in the algorithm *el-join* are omitted since they are essentially transferred to the merge phase. The new merging procedure is depicted in the algorithm APPLY_MERGE. For the ease of presentation, we assume that the tuples with common C_1 value fit always in the merge buffer(s). Note that the application of a rule in the algorithm APPLY_MERGE is performed in exactly the same way we described in Section 4.1; if a rule is applied as a whole, then the new tuples are sent directly to the output buffers, otherwise they are pushed into the streams of the static relations. Whenever we use the term "new tuple" in the pseudocode, we refer to a tuple that is produced in the previous step of the iteration. An "old" tuple is one produced in an older step. Recall that this information is denoted by the first bit of the T attribute of each tuple. This bit is updated when flushing the tuples to disk after merging.

Algorithm: APPLY_MERGE
Input : The relations ΔR_G and R_G that need to be merged;
Output : The order-preserving relation R_m

```
1  let $B_m$ be the merge buffer(s);
2  while there are tuples to merge in $\Delta R_G$ and $R_G$ do
3      mark tuples in R as "old" using the corresponding bit in the T attribute;
4      move as many tuples as possible from $\Delta R_G$ and $R_G$ into $B_m$ so that the total order
       with respect to the sort attributes is preserved.
5      identify the minimum $C_1$ value in $B_m$, let $min_B$;
6      for each unconsidered tuple $t$ : $t$ is "new" and $t.C_1 = min_B$ do
7          apply active rules of Class 1;
8          trigger active rules of Class 2;
9      for each unconsidered pair of tuples $(t_1, t_2) \in B_m$ : $t_1$ is a "new" tuple and $t_1.C_1$ =
       $t_2.C_1 = min_B$ do
10         trigger active rules of Class 3;
11         apply active rules of Class 4;
12     if there are shifted tuples from a previous step then
13         remove shifted tuple(s) from $B_m$;
14     if $B_m$ is full then
15         flush tuples to disk;                    //mark tuples as ``old''
16         shift the tuple(s) with the maximum $C_1$ value in the beginning of $B_m$;
```

6 Theoretical properties

In this section we provide a proof sketch for two important properties of our algorithm: (i) its correctness and (ii) its efficiency. In the former case we show that it produces all valid tuples with respect to the rules of Fig. 3 and in the latter we prove that it follows a semi-naive evaluation strategy, i.e., it does not produce the redundant intermediate results of a naive approach.

6.1 Correctness

We prove that the proposed algorithm and its optimizations provide the correct result with respect to the inference rules. To show this we have to show that (i) a complete evaluation of the el-join is equivalent to the application of every rule in isolation, and (ii) the pruning optimization of Section 5 does not affect the final result.

6.1.1 el-join operator

In Section 4.1 we showed that the el-join operator evaluates every single rule of Figure 3. The evaluation takes into account all self joins on R_G (that is, the joins between R_G and its copy R'_G as well as the self joins on $R_G.C_1$), and also the joins with the static relations wherever needed (completeness). Moreover, the initial axioms have a 1-1 mapping to their relational representation, as explained in Section 3. This guarantees that each rule will only consider the correct tuples and, hence, no incorrect results are produced (soundness). Intuitively, the output of the el-join operator is exactly the same with the output of an algorithm that tests each rule sequentially; the only difference is that our operator avoids scanning R'_G and R_G again and again for each one of the 16 rules of Fig. 3. What remains to be shown is that the iterative application of el-join on the schema Σ of Section 3 still produces the correct result.

6.1.2 Fix-point evaluation

Lemma 3 *The algorithm BRA will terminate after producing exactly the same result with an algorithm that applies each rule of Fig. 3 sequentially.*

Proof (Sketch) Within our graph-based modeling, it is easy to see that the our algorithm will terminate. Since the inference rules add only new edges to the graph of Section 3 but no new nodes, in the worst case the fix-point is reached when all possible types of edges appear between every single pair of nodes in the graph. The types of edges are finite and their number is equal to the size of L_E (see Section 3). We now show that the baseline algorithm of Section 4 (without any optimizations) produces the same result with a fix-point algorithm that applies one rule after the other.
No invalid results are created. We have shown that the el-join creates only the tuples that a single application of every rule would create. Thus, the first time that el-join is applied to the whole dataset, it will produce only tuples that will eventually be found by the naive recursive application of the rules. By induction, in the i-th iteration of BRA all axioms in R^i_G and R'^i_G are correct (i.e., they will be found by the sequential method too) because el-join will have produced only valid tuples in the previous steps. Thus no invalid tuples can be found by BRA.
No results will be omitted. Assume that the algorithm BRA is completed and it has not found at least one valid tuple t. This means that, at its last step, it applied el-join

and the latter did not produce any new tuples. Since we showed that BRA does not produce any wrong tuples, then at the last step all axioms should be correct, i.e., they will be detected by the sequential algorithm. Moreover, since BRA does not delete any tuple, the produced result will be a superset of the initial tuples. This means that if we apply the sequential method to the result of BRA it should discover the missing tuples. To do so, at least one inference rule that is applied once (and in isolation) to the result of BRA should produce at least one tuple that is not producible by el-join. Since el-join is equivalent to the sequential application of all inference rules, this cannot happen, thus, BRA cannot miss any result.

6.1.3 Pruning

Proof LEMMA 1. (Sketch) The reason a tuple t of type 7 (C $\sqsubseteq$ $\exists$R.Self) is not needed in R'_G and $\Delta R'_G$ lies in the way the *el*-join operator works and the fact that such tuples are joined with a tuple from R_G only on the C_1 attribute. This is done in rules **IR8**, **IR9**, **IR10**, **IR11**, and **IR14**. As shown in the algorithm *el*-join of Section 4.1, the only type of join that requires a tuple to be included in the left relation S, i.e., in R'_G and $\Delta R'_G$, is the join $R'_G.C_2 \bowtie R_G.C_1$ (**lines 10-12**). All other joins are performed with tuples of the right relation P, i.e., with tuples of R_G and ΔR_G. Since tuples of type 7 do not participate in such a join pattern, they can be included only in R_G and ΔR_G.

Proof LEMMA 2. (Sketch) Tuples of Type 1 are needed by the rules **IR1**, **IR3** and **IR11** on the left side, i.e., on R'_G and $\Delta R'_G$. When considering each rule in isolation, it is known that **IR1** can be evaluated correctly in a right-linear fashion, so the new tuples can be appended only to ΔR_G [44]. What remains is to prove that the same holds for rules **IR3** and **IR11** when all rules are evaluated together. Let E = $\{t_1$, t_2, t_3, ..., $t_n\}$, $n \geq 2$, be a set of tuples of type 1 such that $t_i.C_2$ = $t_{i+1}.C_1$, $0 < i < n$. Intuitively, these tuples define a "path" between the nodes $t_1.C_1$ and $t_n.C_2$. Let also t_p be a tuple that is created by the recursive application of **IR1** on E such that $t_p.C_1$=$t_1.C_1$ and $t_p.C_2$=$t_n.C_2$. Now, consider a tuple t of type 3 such that $t_n.C_2$ = $t.C_1$. According to **IR3**, tuples t_p and t must be joined in order to produce a new tuple t' such that $t'.C_1$ = $t_1.C_1$, $t'.R$ = $t.R$, and $t'.C_2$ = $t.C_2$. In case t_p is appended only to ΔR_G, the Algorithm BRA is not going to perform this join. However, the same tuple t' can be created by the recursive application of **IR3** on the tuples of E as follows. First, we apply **IR3** on the pair of tuples (t_n, t) and produce the tuple u_1, then we do the same on the pair (t_{n-1}, u_1) and produce the tuple u_2, then on the pair (t_{n-2}, u_2) and so on till the tuple t' is produced. Algorithm BRA will correctly perform the previous recursive application of **IR3** if and only if (a) each tuple of type 3 exists in the right side, and (b) the set E of tuples exists in the left side. The former is true since every tuple is appended to the right side. Regarding the latter, we have to distinguish three cases:

1. All tuples in E exist in R_G from the beginning. In this case, the tuples exist in both sides since they are copied from R_G to R'_G in initialization.

2. A tuple in E is produced by a rule other than **IR1**. In this case the tuple exist in the left side because all tuples of type 1 which are produced by a rule other than **IR1** are appended to both deltas.

3. A tuple in E is produced by **IR1**. In this case, the tuple was produced (and, hence, it can be substituted in E) by a chain of tuples like the one we considered before. By inductively applying the same process to each tuple produced by **IR1** in this chain, we result with a set of tuples that define the needed path and exist in the left side due to 1 or 2.

6.2 Semi-naive Evaluation

Lemma 4 *The algorithm BRA is equivalent to a semi-naive evaluation of each rule of Fig. 3 in isolation.*

Proof The approach we described is equivalent to a semi-naive evaluation of each rule in isolation and it does not materialize any redundant facts. For the rules of Classes 4 and 5 that include a join of the form $R'_G \bowtie_{R'_G.C_2=R_G.C_1 \wedge R'_G.T=f(R_G.T)} R_G$, this is guaranteed by the overall design of the algorithm and the use of the two deltas. Specifically, each pair of tuples joined in **lines 11 and 13** of the algorithm BRA includes *at least* one "new" tuple, i.e., a tuple belonging to the relation $\Delta R'^i_G$ and/or ΔR^i_G. These deltas are produced in the $(i-1)$-th step of the iteration and participate in the *el*-join only after removing the "old" tuples (**lines 9 and 10** in BRA). Note that in the join $\Delta R'^i_G \bowtie_{el} R^i_G$ (**line 13** in BRA) both tuples may be "new" since ΔR^i_G is included in R^i_G.

Regarding the rules of Class 3 and 4 that include a self join of the second form, i.e., $R_G \bowtie_{R_G.C_1=R_G.C_1 \wedge R_G.T=f(R_G.T)} R_G$, the semi-naive evaluation is achieved in a different way. The complete application of these rules needs all tuples produced till the i-th step of the procedure, thus, they are applied only in the join $\Delta R'^i_G \bowtie_{el} R^i_G$ (**line 13** in the algorithm BRA), as a self join on the "sliding window" over R^i_G (**lines 7-9** in the algorithm *el*-JOIN). In order to guarantee that the "old" tuples will not be checked one another under the same rules at that point, we need to distinguish these tuples from those produced in the previous step $(i-1)$. This information is encoded in the T attribute of each tuple just before the join $\Delta R'^i_G \bowtie_{el} R^i_G$, i.e., when merging the relations R^{i-1}_G and ΔR^i_G in the i-th step (**line 12** of the algorithm BRA). There, all tuples coming from R^{i-1}_G are marked as "old", whereas the tuples from ΔR^i_G are marked as "new" (default value).

The remaining rules of Classes 1 and 2 are evaluated in a semi-naive fashion using a third technique. Since these rules involve either a join with the static relations of Σ or no join at all, they should be applied only to each "new" tuple of P (**lines 5-6** in the algorithm *el*-JOIN), i.e., only to the tuples of R^0_G in the first iteration and the tuples of ΔR^i_G at each subsequent step. Instead of using an additional condition on the T attribute there, we can simply *activate* these rules only in the initial join $R'^0_G \bowtie_{el} R^0_G$ and the join $R'^{i-1}_G \bowtie_{el} \Delta R^i_G$ (**lines 4 and 11** in the algorithm BRA). As implied in the previous paragraph, this technique is also used in the rules of Class 3. The latter are deactivated in the join $\mathrm{R}'^{i-1}_G \bowtie_{el} \Delta\mathrm{R}^i_G$ because $\Delta\mathrm{R}^i_G$ does not include all tuples produced in the previous steps.

7 Performance Evaluation

In this section we present an experimental evaluation of BRA on real and synthetic ontologies. Our algorithm is compared with an alternative bottom-up algorithm, termed ORT, that applies the rules independently, and also with the most mature implementations of the state-of-the-art inference algorithms from the related work.

Experimental Setting. We compared BRA with the state-of-the-art systems in the three areas of related work: Prolog-based systems (YAP [7] and XSB [6]), Deductive databases (DLV [3] and LogicBlox [4]). These tools are the most actively maintained and mature implementations in the respective fields [36]. The Prolog-based systems follow a top-down evaluation strategy whereas DLV and LogicBLox are bottom-up Datalog engines. All these systems operate only in main-memory and they were allowed to use the entire memory of the machine. Since none of these systems offer built-in support for the fragment of OWL2-EL, all inferences rules of Fig. 3 were defined manually.

To provide a better understanding on the benefits of BRA and since the implementation details of the aforementioned systems are not always transparent, we implemented an ORT (One Rule at a Time) algorithm which applies each rule independently as described in [24]. In the experiments below, ORT operates on the most favorable storage scheme for its evaluation strategy, that is, a scheme with one separate relation per type of axiom as explained in Section 3.1. The ORT algorithm applies the rules as follows. First, **IR1** and **IR2** are interchanged: **IR1** is applied exhaustively, then followed by **IR2** which is applied once, and this procedure is repeated until no new axioms are created. After this, it applies all other rules in a round-robin fashion, i.e, each one of the remaining rules is applied once, and the algorithm returns to the first step. The whole process repeats until no more axioms are produced. This version of ORT (first **IR1** and **IR2** exhaustively, and then the rest) outperformed the completely naive approaches where all rules are applied only once at each step of the iteration. Just like BRA, ORT applies each rule in a semi-naive fashion; previously examined combination of axioms are ignored and only new ones are considered at each step of the procedure.

BRA operates on 5 base relations (schema of Fig. 5), which are stored in 5 files. To demonstrate the effectiveness of the optimization heuristics we also implemented a simplified version of BRA, namely SN (from semi-naive), which is basically BRA without any optimization heuristic of those discussed in Section 5.1. Moreover, to highlight how BRA can exploit modern hardware with a large amount of memory, we created a main-memory version of BRA, denoted as BRA-M. BRA-M operates like BRA but, given enough memory, it caches the base relations, creates a hash index on $R'_G.C_2$ and another one on $R_G.C_1$, and performs hash joins instead of sort-merge joins. We emphasize that BRA-M is not proposed as a state-of-the art main-memory algorithm; we only use it to demonstrate how BRA can exploit a very large cache. Finally, to clearly assess the impact of the storage scheme on BRA, the latter is applied on the schema used for ORT (one relation per type of axiom) and report the results for this version as BRA-A.

For a fair comparison, we always give BRA, BRA-A, ORT and SN the same total amount of memory buffers (default setting is 40MBs). In BRA, BRA-A and SN, the given memory is used for performing the steps described in Section 4. Note that, even with this small amount of memory, the static relations of Fig. 5 can be completely cached (after the first step of the iteration), still, leaving enough memory for BRA to operate. A small amount of memory is also kept for the output buffers. In the case of ORT, the static relations can again be completely cached (after the first step of the iteration), and the rest of the cache is used for (i) the joins between the different relations, (ii) the external sortings during the iterations, (iii) the set difference operations, and (iv) the output buffers.

Datasets. We used two large datasets from the biomedical domain, namely SNOMED CT [14] and GALEN8 [12] which are used in various clinical studies. SNOMED CT has 379692 classes, 61 roles, 623999 class inclusions and 11 role inclusions. GALEN8 has 125391 classes, 995 roles, 280693 class inclusions and 1387 role inclusions. Both ontologies have many class inclusions with complex dependencies and, thus, they have become the "standard" ontologies in all published benchmarks for OWL reasoners. Their complexity is reflected in the large number of new (implicit) axioms produced by the inference (more than 11M for SNOMED CT and 30M for GALEN8). Using these real-world data, we also created synthetic data of various sizes in order to evaluate the scalability of our approach. The synthetic data are multiplications of the original in two ways. First, we kept the same forms of axioms but added copies of each axiom by replacing the IDs of the classes appearing in it. In this case, the resulting dataset represents multiple graphs which are isomorphic to the original. Second, we kept the number of classes (nodes) constant and multiplied the number of properties (labeled edges in the graph of Section 3). This way, we essentially multiply the number of axioms of type 3, 4, 7 and 8 as well as the related property axioms (i.e., the axioms of type 11 and 12). Intuitively, the second method results in a graph with larger node degree; it is applied only to GALEN8 because SNOMED CT contains no axioms of type 12 and very few of type 11.

Implementation details. All our algorithms were implemented in C++ (g++ 4.6.3). The experiments were conducted on a machine running Linux Ubuntu (3.5.0-40) with a CPU at 3.60GHz, 64Gb of RAM, and a 750Gb SATA hard disk. In order to present accurate results about memory utilization and the exact performance of each algorithm under a limited memory budget, all disk-based implementations (SN, BRA, BRA-A and ORT) bypass the kernel's caching mechanism by performing the I/O operations with the Linux O_DIRECT flag. Note that the reported results do not include the cost of normalization which is very small (11 seconds for SNOMED CT and 29 seconds for GALEN8) and needed by both of our algorithms and all third party competitors.

Comparison with other methods. We compare all methods in terms of running time, split as CPU time and I/O time (we estimate it as total time minus CPU time), and in terms of I/O disk accesses for disk based methods. In all experiments, time is given in seconds, The results are depicted in Fig. 7.a (SNOMED CT) and in Fig. 7.b (GALEN8). Amongst all the systems we tested, only XSB managed to terminate on GALEN8; the rest either crashed or did not terminate after running for hours (more than three hours). We observe that BRA outperforms all competitor systems (except YAP in the SNOMED CT dataset) even if they do only main-memory evalu-

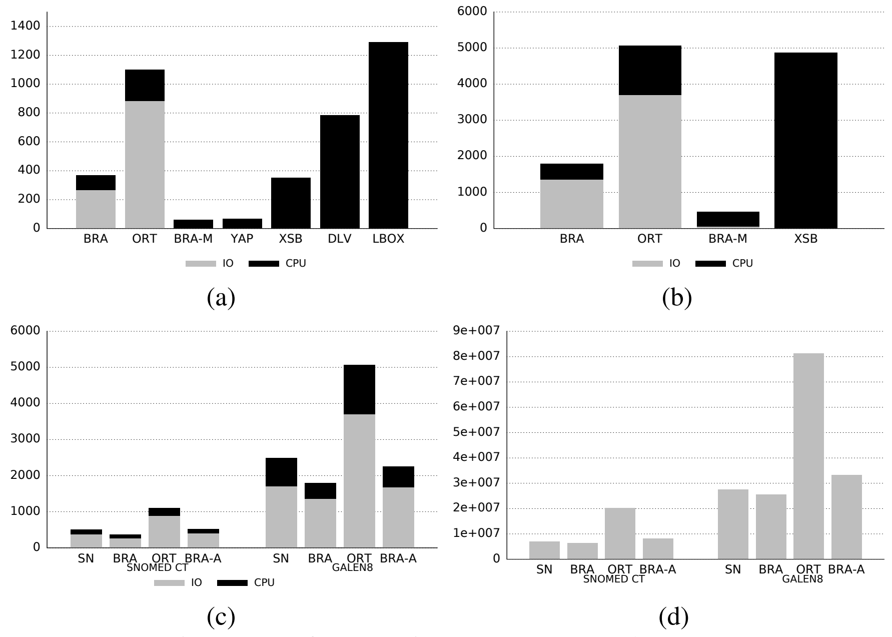


Figure 7: Performance in SNOMED CT and GALEN8

ation. The BRA-M algorithm slightly outperforms YAP even for SNOMED CT (60 secs for BRA-M vs 68 for YAP). The memory requirements of all systems are significant: for SNOMED CT, YAP needs 2.2G, XSB 1.8G, DLV 3.6G and LogicBlox 18G of memory. This amounts to an increase of x14, x12, x24, x121 compared to the size of the input dataset (148MB in OWL/XML format). For GALEN8, XSB needs 5.3G (x19 increase - the initial dataset size is 274MB in OWL/XML format). Note that the previous results correspond to the maximum memory allocated by the process that performed the inference. Moreover, we tried to compare BRA to OWLIM [21] and Jena [1], but they failed to compute the logical closure in both datasets (they both operated for more than 24 hours without any result). Finally, an important observation from the results of BRA and ORT is that computing logical closure on the disk is an I/O-bounded problem, since the evaluation time of both these algorithms is dominated by I/O time.

Understanding BRA's performance factors. We compared BRA to the basic seminaive strategy SN to understand the impact of our optimizations, and to BRA-A to understand the impact of the storage scheme in the overall performance of BRA. The results are depicted in In Fig. 7.c and Fig. 7.d in terms of seconds and I/O accesses respectively. The optimizations we propose give a 28$\%$ speedup to BRA with respect to SN and the storage scheme provides 30$\%$ faster evaluation for BRA with respect to BRA-A for SNOMED CT and 20$\%$ for GALEN8. Note that SN is worse than BRA due to the lack of logical optimizations but, still, it is much better than ORT (in terms of both time and I/Os). The reason BRA-A performs worse than BRA is that is causes an increased number of random I/Os. Since it has to scan more than two

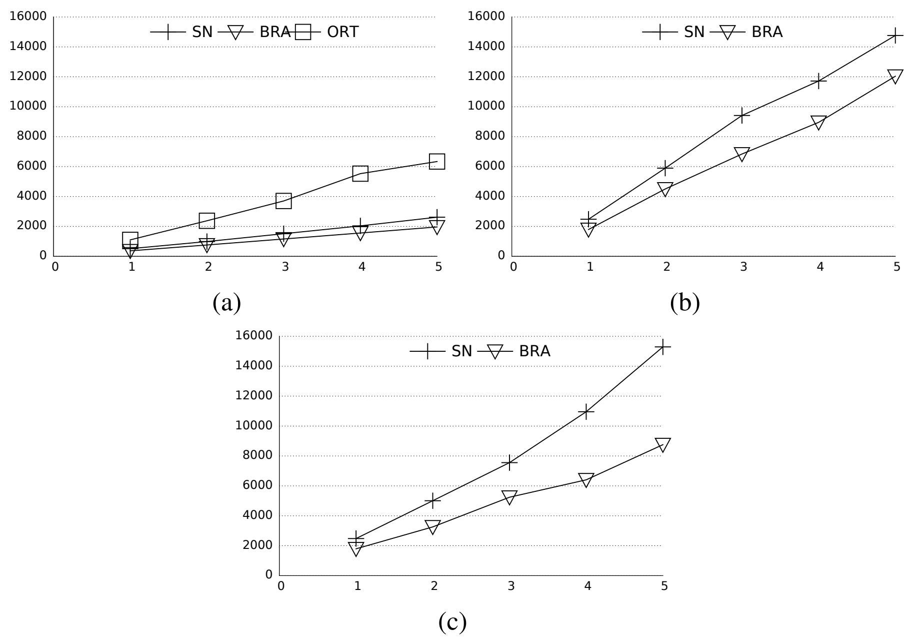
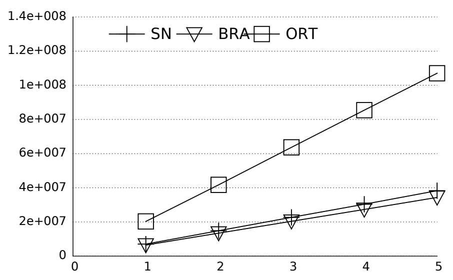


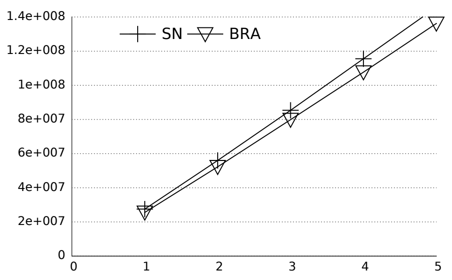
Figure 8: Scaling with dataset size (time in seconds)

relations at each step of the iteration (not only R'_G and R_G), it cannot fetch as large blocks with sequential scans as BRA; the input buffers must be split according to the base relations and, hence, each relation is assigned with a smaller buffer. Finally, BRA takes less than 30% of the ORT time, in both datasets, and this performance gain is reflected in the total number of I/Os each algorithm performs.

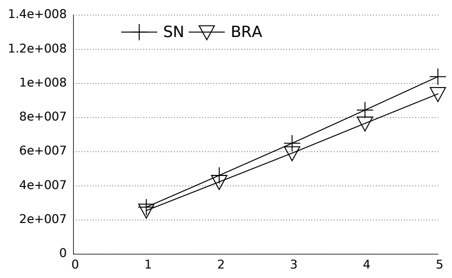
Scaling with the dataset size. Fig. 8 and 9 demonstrate how BRA and ORT scale with the dataset size. ORT did not terminate within a reasonable time for the GALEN8 datasets due to the increased number of I/Os, so it is omitted from the respective figures (Fig. 8.b, 8.c, 9.b and 9.c). In Fig. 8.a and 8.b we see results when the dataset size increases. The datasets in these two experiments have been created by multiplying SNOMED CT and GALEN8 respectively according to the first method we described previously. The x-axis traces the multiplication factor (e.g., in Fig. 8.a for $x = 1$ we have the original SNOMED CT, for $x = 2$ we have a dataset that is double the original SNOMED CT and so on). The y-axis traces the performance in seconds. We can see that BRA scales linearly as ORT but with a significantly smaller slope. We also note that the optimization heuristics present the same behaviour, hence, by improving the performance of the algorithm in each iteration, they reduce the scaling slope even more. Finally, in Fig. 8.c we see how BRA behaves as the number of edges grow according to the second multiplication method. Again the behavior is linear, but we notice that the effect of the pruning heuristic is more substantial. The reason behind this lies in the number of duplicated tuples produced by **IR3**. In GALEN8, **IR6** produces too many tuples of type 3 which, in combination with tuples of type 1 and rule **IR3**, produce even more tuples of type 3. By pruning tuples of type 1 from the left delta (Section 5.1), **IR3**



(a)



(b)



(c)

Figure 9: Scaling with dataset size (disk page accesses)

produces significantly less duplicated tuples at each step of the iteration and, hence, both the CPU time and the total I/Os are reduced. The I/Os performed in each one of the experiments of Fig. 8.a, 8.b, and 8.c are given in Fig. 9.a, 9.b, and 9.c respectively. Note that the impact of the heuristics is more profound in terms of execution time, since pruning significantly reduces CPU time as well.

In summary, there are no I/O-aware methods for evaluating the logical closure and even those that can evaluate the closure in main memory cannot outperform BRA in most cases. Moreover, we showed that the proposed storage scheme and the optimization heuristics have a significant impact on BRA.

8 Related Work

Artificial Intelligence. Regarding the fragment of OWL2-EL, the rule-based approach we highlighted in Section 2 was introduced in [17, 19]. All widely-used reasoners (also known as EL classifiers) [25] work in main memory and use a variation of the algorithm presented in [19]. Highly-optimized versions of this algorithm can be found in [31] and [2]. Despite their particular differences, all approaches rely on creating dynamic lists for each class of the dataset (in main memory) and keep track of the axioms that contain it. Inference rules use the lists to detect axioms that contain a class and update them with the new axioms they infer. The design of the in-memory algorithms is based on the assumption that the cost of lookups and updates in the lists is negligible (which is true for main-memory systems where the lists are implemented as a hash map). Using these methods for disk-based evaluation is clearly inefficient since they would perform a huge number of random I/Os, hence, they are unsuitable for our problem setting.

Deductive Databases and Prolog-based Systems. Research in deductive databases and Prolog-based systems focuses mainly on the logical optimization of queries with at least one of the variables bounded. In our particular problem, such a query would be the query "subClassOf(c,X)?" where c is the ID of a specific class in the ontology and X is a variable. Intuitively, this query asks for all superclasses of c. The problem we address here requires the answers to queries of the form "subClassOf(X,Y)?" which have no bounded variables. Regardless the existence of free or bounded variables in a query, Prolog-based systems will always evaluate it following a *top-down* strategy. A top-down algorithm starts from the given query (goal), it substitutes this goal with its subgoals (i.e., with the predicates found in the body of a rule where the initial goal appears as head), then it recursively substitutes all these subgoals with their respective subgoals and so forth. When the query is bounded, the approach we described can limit the search space and avoid infering axioms that are irrelevant to the query [20]. However, in the logical closure computation all facts are relevant, hence, the evaluation cannot benefit from the pruning power of the top-down strategy. At the same time, the substitution of goals with subgoals adds significant overhead (both in time and space) and it has also the problem of entering in infinite loops (although the latter is avoided with the use of Tabling [43]). In our particular setting, note that the inference rules of Fig. 3 have many common predicates in the head of the rules; this means that a top down algorithm would have to examine many different paths of substitutions for evaluating a predicate. On the other hand, a bottom-up strategy for a query "subClassOf(X,Y)?" will exhaustively evaluate each rule of Fig. 3 (in a semi-naive fashion), and terminate when there are no new tuples to produce. This simple logic proves more efficient for logical closure, i.e., when the algorithm has to compute "everything". As a final comment, the well-known Magic Sets optimization [38, 39] has been proposed for reducing the irrelevant facts in a bottom-up evaluation of a (partially) bounded query, hence, the logical closure computation cannot benefit from it.

An interesting db-oriented approach for ontology management is recently presented in [22, 27]. This work extends Datalog in order to express axioms like those of Fig. 1 as rules. Note that some axioms in OWL are not interpretable into safe Datalog rules without this extension. Representative examples are the axioms of the form $\mathrm{C}_1 \sqsubseteq \exists\mathrm{R}.\mathrm{C}_2$ which define Tuple-Generating Dependencies (TGDs) and whose entailments are traditionally checked using the Chase algorithm. The theoretical framework presented in these papers is targeted to query answering and it is not optimized for the logical closure computation.

Relational databases. Recursive queries in relational databases can be expressed up to some extent with the Common Table Expressions (an SQL standard) or the proprietary features of some systems like Oracle's CONNECT_BY. The main drawback of such approaches is that the inference rules of Fig. 3 cannot be expressed in a single SQL query and, thus, they can only be evaluated sequentially, i.e., one rule (query) after the other in a predefined order. As a result, the disk-resident relations are accessed on a per-rule basis that is extremely inefficient in terms of both CPU and I/O cost. ORT algorithm follows exactly this approach and its performance drawbacks with respect to

BRA are highlighted in Section 7.

Work in [26] performs the inference tasks with a main-memory OWL reasoner and uses the database mainly as a backend. On the other hand, [35] and [24] take advantage of the RDBMS's built-in features and employ User Defined Functions (UDFs) to express the recursive rules. Still, they apply each rule independently from the others like ORT. A recent approach in the field is Oracle's Semantic Graph [5], a disk-oriented platform for reasoning with various OWL fragments. In this system, the application of the inference rules is provided as a single database operator.

Multiple-query Optimization. The multiple-query optimization paradigm we adopt has been extensively studied in the database community [29, 40, 41]. These works differ from ours in the following. First, they are general in that they can handle queries arbitrarily given by the user. Here the recursive rules are predefined which allows for custom optimizations. Second, their main focus is on reusing the intermediate query results. Here we use the common parts of the rules as a pattern to access the data and save I/Os.

Semantic Web Systems. Recently, there is an increasing interest in the management of incomplete RDF knowledge bases. To this end, the new specification of SPARQL [10] incorporates similar reasoning tasks to those we presented here so that the evaluation of queries on top of RDF graphs can capture the "hidden" relationships (edges) implied by the semantics of RDFS and OWL. This is a rather uninvestigated and intriguing setting where the results of our work could be also exploited. [37] and [46] are motivated by the same principle, however, from a different perspective. They support general user-defined rules and they do not address any of the multi-query optimizations we consider here. Other triple stores with inference support are Jena [1] and OWLIM [21]. These tools are designed for storing and querying RDF data, hence, every OWL axiom is internally represented as an RDF triple or a set of RDF triples depending on how complex it is. This design leads to a significantly more verbose representation of the axioms compared to the one of Section 3 and, hence, more complicated (n-way) joins are required for applying the rules. None of these systems supports built-in reasoning algorithms for OWL2-EL but they offer generic rule engines where the rules can be defined manually. Jena supports top-down and bottom-up evaluation whereas OWLIM has a pure bottom-up engine. Again, rules in these systems are applied sequentially like in ORT.

9 Conclusions

In this paper we proposed an I/O-aware algorithm that efficiently computes the logical closure of a set of inference rules on large OWL2-EL ontologies. The salient feature of our approach is that we model and store the ontology axioms in a homogeneous way so that different inference rules are evaluated in bulk and within the same I/O operations. We demonstrated experimentally that our algorithm outperforms existing strategies and scales very well to large datasets. A future research direction is to extend our work in other fragments of OWL, e.g., OWL2-RL, where rule-based inference is already a common practice.

References

[1] Apache Jena. *https://jena.apache.org/*.

[2] CB reasoner. *http://www.cs.ox.ac.uk/isg/tools/CB/*.

[3] DLV System. *http://www.dlvsystem.com/*.

[4] LogicBlox. *http://www.logicblox.com/*.

[5] Oracle Spatial and Graph 12c - RDF Semantic Graph (white papers). *http://www.oracle.com/technetwork/database/options/spatialandgraph/documentation/rdfsem-techinfo-1916685.html*.

[6] XSB Prolog. *http://www.xsb.com/what-we-do/emerging-technologies/xsb-prolog*.

[7] Yet Another Prolog. *http://www.dcc.fc.up.pt/ vsc/Yap/index.html*.

[8] RDF Vocabulary Description Language 1.0: RDF Schema. *http://www.w3.org/TR/rdf-schema/*, February 2004.

[9] OWL 2 Web Ontology language Structural Specification and Functional-style Syntax. *http://www.w3.org/TR/owl2-syntax/*, 2009.

[10] SPARQL 1.1 Query Language. *http://www.w3.org/TR/sparql11-query/*, November 2012.

[11] Foundational Model of Anatomy. *http://sig.biostr.washington.edu/projects/fm/AboutFM.html*, 2013.

[12] Generalized Architecture for Languages, Encyclopedias and Nomenclatures in Medicine. *http://www.openclinical.org/prj_galen.html*, 2013.

[13] NCI Thesaurus. *http://ncit.nci.nih.gov/*, 2013.

[14] Systematized Nomenclature of Medicine - Clinical Terms. *http://www.ihtsdo.org/snomed-ct/*, 2013.

[15] R. Agrawal, S. Dar, and H. V. Jagadish. Direct transitive closure algorithms: Design and performance evaluation. *TODS*, pages 427–458, 1990.

[16] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.

[17] F. Baader, S. Brandt, and C. Lutz. Pushing the $\mathcal{EL}$ envelope. In *IJCAI*, 2005.

[18] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[19] F. Baader, C. Lutz, and B. Suntisrivaraporn. Efficient reasoning in $\mathcal{EL}+$. In *DL*, 2006.

[20] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, 1986.

[21] B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov. OWLIM: A family of scalable semantic repositories. In *Semantic Web J*, 2011.

[22] A. Cali, G. Gottlob, and A. Pieris. Advanced processing for ontological queries. *PVLDB*, 3(1):554–565, 2010.

[23] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, R. Rosati, and M. Ruzzi. Using OWL in data integration, 2009.

[24] V. Delaitre and Y. Kazakov. Classifying $\mathcal{ELH}$ ontologies in SQL databases. In *OWLED*, 2009.

[25] K. Dentler, R. Cornet, A. ten Teije, and N. de Keizer. Comparison of reasoners for large ontologies in the OWL 2 EL profile. *Semantic Web*, 2(2), 2011.

[26] J. Dolby, A. Fokoue, A. Kalyanpur, E. Schonberg, and K. Srinivas. Scalable Highly Expressive Reasoner (SHER). *J. Web. Sem.*, 7(4), 2010.

[27] G. Gottlob, G. Orsi, and A. Pieris. Ontological queries: Rewriting and optimization. In *ICDE*, 2011.

[28] H. V. Jagadish, R. Agrawal, and L. Ness. A study of transitive closure as a recursion mechanism. In *SIGMOD*, 1987.

[29] M. Jarke. Common subexpression isolation in multiquery optimization. *Query Processing in Database Sys.*, 1985.

[30] Y. Kazakov, M. Kroetzsch, and F. Simancik. Practical reasoning with nominals in the $\mathcal{EL}$ family of description logics. In *KR*, 2012.

[31] Y. Kazakov, M. Krötzsch, and F. Simančík. ELK Reasoner: Architecture and Evaluation. 2012.

[32] V. Kolovski, Z. Wu, and G. Eadon. Optimizing enterprise-scale OWL 2 RL reasoning in a relational database system. In *ISWC*, 2010.

[33] M. Kroetzsch. Efficient inferencing for the description logic underlying OWL EL. Technical Report 3005, Institute AIFB, Karlsruhe Instistute of Technology (KIT), 2010.

[34] M. Kroetzsch. Efficient rule-based inferencing for OWL EL. In *IJCAI*, 2011.

[35] M. Kroetzsch, A. Mehdi, and S. Rudolph. Orel: Database-driven reasoning for OWL 2 profiles. In *DL*, 2010.

[36] S. Liang, P. Fodor, H. Wan, and M. Kifer. OpenRuleBench: An analysis of the performance of rule engines. In *WWW*, 2009.

[37] N. Nakashole, M. Sozio, F. M. Suchanek, and M. Theobald. Query-time reasoning in uncertain RDF knowledge bases with soft and hard rules. In *VLDS*, 2012.

[38] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *J. Log. Program.*, 11(3-4):189–216, 1991.

[39] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. *TKDE*, 6(4), 1994.

[40] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi-query optimization. In *SIGMOD*, 2000.

[41] T. Sellis. Multi-query optimization. *TODS*, 13(1):23–52, 1988.

[42] S. Suwanmanee, D. Benslimane, P.-A. Champin, and P. Thiran. Wrapping and integrating heterogeneous databases with OWL. In *ICIES*, 2005.

[43] K. T. Tekle and Y. A. Liu. More efficient Datalog queries: Subsumptive Tabling beats Magic Sets. In *SIGMOD*, 2011.

[44] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. II.* Computer Science Press, 1989.

[45] Z. Wu, G. Eadon, S. Das, E. I. Chong, V. Kolovski, M. Annamalai, and J. Srinivasan. Implementing an inference engine for RDFS/OWL constructs and user-defined rules in oracle. In *ICDE*, 2008.

[46] M. Yahya and M. Theobald. D2R2: Disk-oriented deductive reasoning in a RISC-style RDF engine. In *RuleML*, 2011.