Efficient Identification of Implicit Facts in Incomplete OWL2-EL Knowledge Bases

John Liagouris
Department of Electrical and Computer Engineering
National Technical University of Athens
Greece
liagos@dblab.ece.ntua.gr

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

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. Although 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 this paper is an algorithm that allows the effective use of OWL for integrating data in an environment with limited memory. The core idea is to exhaustively apply a set of complex inference rules on large disk-resident datasets. To the best of our knowledge, this is the first work that proposes an I/O-aware algorithm for tackling with such an expressive subset of OWL like the one we address here. Previous approaches considered either simpler models (e.g. RDFS) 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 OWL 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 with 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 valid axioms with respect to the rules is known as completion or saturation of the ontology.

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 [41]. 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 [5, 44, 31] 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 saturation problem as a reachability problem on a graph that represents the axioms of the ontology. Based on this model, 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
2. We develop a novel method that operates efficiently under a limited memory budget and performs the saturation of the ontology 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 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 SROEL [33]. 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 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 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 \( C_1 \cap C_2 \) 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.

| Class Inclusion (SubClassOf) | \( C_1 \subseteq C_2 \) |
| Property Inclusion | \( R_1 \subseteq R_2 \) |
| Complex Property Inclusion | \( R_1 \circ R_2 \subseteq R_3 \) |

#### Table 1: Symbols and Terminology

| Source A | 1. \( \text{InfectedWithVirusA} \cap \text{NotVaccinated} \subseteq \text{Ill} \) |
| Source A | 2. \( \exists \text{Vaccinated}.\text{VaccineTypeX} \subseteq \text{NotVaccinated} \) |
| Source A | 3. \( \{\text{va}\} \subseteq \text{VaccineTypeX} \) |
| Source B | 4. \( \{\text{john}\} \subseteq \text{Vaccinated1994} \) |
| Source B | 5. \( \{\text{john}\} \subseteq \text{InfectedWithVirusA} \) |
| Expert | 6. \( \text{Vaccinated1994} \subseteq \exists \text{Vaccinated}.\{\text{va}\} \) |

#### Figure 2: Integration example

With this restriction are found in the literature as nominal-safe [29]. 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 \( \cap \) NotVaccinated, Vaccinated, \{\text{va}\} and \( \exists \text{Vaccinated}.\text{VaccineTypeX} \) are complex classes. Vaccinated is a role, and \( \text{john} \) and \( \text{va} \) are individuals (expressed as nominals). 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 insufficient vaccine of type VaccineTypeX (\( \exists \text{Vaccinated}.\text{VaccineTypeX} \)) should be treated as not vaccinated. Finally, axiom 3 denotes that the vaccine \( \text{va} \) 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 \( \text{john} \) 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 in 1994 where vaccinated with vaccine \( \text{va} \). 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 \( \text{john} \) is ill, since he has been infected with virus A and he has been vaccinated in 1994 when everyone was vaccinated with the insufficient vaccine \( \text{va} \) 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.

### Figure 1: Normal Forms of Axioms

Ontologies in 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 \( \{a\} \) is a subclass of \( \{b\} \) (and the reverse) if \( a = b \). An axiom of type 6 is used to model class assertions, i.e., a class assertion is expressed as an inclusion of the form \( \{a\} \subseteq C \). Similarly, a property assertion that associates \( a \) with \( b \) through role \( R \) is equivalent to the axiom \( \{a\} \subseteq \exists R.\{b\} \) (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 \( C_1 \cap C_2 \subseteq \bot \) and inconsistencies as \( \{a\} \subseteq \bot \) (which often arise through 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
For 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 completion or saturation. The result of this process is a dataset D′ such that D ⊆ D′.

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 [43]. The complexity of the task we described 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. IR2 states that if a class C1 is a subclass of C2 and C3, and the intersection of C2 and C3 is a subclass of C4, then C1 should also be a subclass of C4. Due to space restrictions, we refer the reader to [35] for a detailed description of each rule. 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-bound.

Discussion. This work focuses on the saturation of a SROEL ontology under the rules of 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 actual saturation, classification 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 saturated axioms. The most expensive part of classification is by far the saturation, 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, i.e., each symbol C1 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 [32] and [30].

3. DATA MODELING

Our approach adopts a semantically oblivious representation of the ontology based on a graph model. A collection D of axioms in the normal forms of Fig. 1 is modeled as a graph with a small collection of metadata, and then the edges of this graph are stored in a relational table. Specifically, the majority of axioms in D, i.e., all axioms except those of type 2, 11 and 12, are modeled as a directed labeled multigraph G(V,E,lv,le), where:

- V is the set of nodes. There is one node for each class in D, including ⊤ and ⊥. 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.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR1</td>
<td>C₁ ⊆ C₂ → C₁ ⊆ C₂ ∧ C₂ ⊆ C₃</td>
</tr>
<tr>
<td>IR2</td>
<td>C₁ ⊆ ∃R.C₂ → C₁ ⊆ C₂ ∧ C₂ ⊆ ∃R.C₃</td>
</tr>
<tr>
<td>IR3</td>
<td>C₁ ⊆ ∃R₁C₂ → C₁ ⊆ ∃R₂C₂ ∧ R₁ o R₂ ⊆ C₃</td>
</tr>
<tr>
<td>IR4</td>
<td>C₁ ⊆ ∃R₁C₂ → C₁ ⊆ C₂ ∧ C₂ ⊆ ∃R₂C₃</td>
</tr>
<tr>
<td>IR5</td>
<td>C₁ ⊆ ∃R₁C₂ → C₁ ⊆ C₂ ∧ C₂ ⊆ ∃R₂C₃</td>
</tr>
<tr>
<td>IR6</td>
<td>C₁ ⊆ ∃R₁C₂ → C₁ ⊆ C₂ ∧ C₂ ⊆ ∃R₂C₃</td>
</tr>
<tr>
<td>IR7</td>
<td>C₁ ⊆ ∃R₁C₂ → C₁ ⊆ C₂ ∧ C₂ ⊆ ∃R₂C₃</td>
</tr>
<tr>
<td>IR8</td>
<td>C₁ ⊆ ∃R₁C₂ → C₁ ⊆ C₂ ∧ C₂ ⊆ ∃R₂C₃</td>
</tr>
<tr>
<td>IR9</td>
<td>C₁ ⊆ ∃R₁C₂ → C₁ ⊆ C₂ ∧ C₂ ⊆ ∃R₂C₃</td>
</tr>
<tr>
<td>IR10</td>
<td>C₁ ⊆ ∃R₁C₂ → C₁ ⊆ C₂ ∧ C₂ ⊆ ∃R₂C₃</td>
</tr>
<tr>
<td>IR11</td>
<td>C₁ ⊆ ∃R₁C₂ → C₁ ⊆ C₂ ∧ C₂ ⊆ ∃R₂C₃</td>
</tr>
<tr>
<td>IR12</td>
<td>C₁ ⊆ ∃R₁C₂ → C₁ ⊆ C₂ ∧ C₂ ⊆ ∃R₂C₃</td>
</tr>
<tr>
<td>IR13</td>
<td>C₁ ⊆ ∃R₁C₂ → C₁ ⊆ C₂ ∧ C₂ ⊆ ∃R₂C₃</td>
</tr>
<tr>
<td>IR14</td>
<td>C₁ ⊆ ∃R₁C₂ → C₁ ⊆ C₂ ∧ C₂ ⊆ ∃R₂C₃</td>
</tr>
</tbody>
</table>

Figure 3: Inference Rules (∧ stands for logical AND)

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

- \[ T₁ \] C₁ ⊆ C₂ is represented by a subclassesOf edge from node C₁ to node C₂.
- \[ T₃ ] C₁ ⊆ ∃R.C₂ is represented by an edge from node C₁ to node C₂, marked with label R⁺.
- \[ T₄ ] C₁ ⊆ ∃R.C₂ is represented by an edge from node C₁ to node C₂, marked with label R⁻.
- \[ T₅ ] {a} ⊆ {b} is represented by a subClassOf edge from node a to node b.
- \[ T₆ ] {a} ⊆ C is represented by a subClassOf edge from node a to node C.
- \[ T₇ ] ⊤ C ⊆ ∃R.Self is represented by an edge from node C to the unique node Self, marked with label R⁺.
- \[ T₈ ] ∃R.Self ⊆ C is represented by an edge from the unique node Self to C, marked with label R⁻.
- \[ T₉ ] ⊤ ⊆ C is represented by a subClassOf edge from the unique node Top to node C.
- \[ T₁₀ ] ⊥ ⊆ C 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 of the example, except axiom 1 which is represented in the metadata, introduces an edge between the nodes of the graph. Axioms 3, 4 and 5 introduce \texttt{subClassOf} edges, while axiom 2 introduces a \texttt{Vaccinated+} edge (since \texttt{Vaccinated} appears on the left side of the inclusion in axiom 2). Axiom 6 introduces a \texttt{Vaccinated+} edge. Using \texttt{IR4} on axioms 3, 6 and 2 we get a new axiom \texttt{Vaccinated1994} \texttt{\notin NotVaccinated}, which is represented by a dashed edge on the graph. This axiom, together with axiom 4, satisfies \texttt{IR1}, so an additional axiom \{\textit{john}\} \texttt{\subseteq NotVaccinated} is added. This last axiom, along with axioms 1 and 5, satisfies \texttt{IR2}, so a final axiom \{\textit{john}\} \texttt{\subseteq III} is added to the original data. We point out that each inference rule can only add edges to the graph \textit{G}, so the set \textit{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 \textit{G} (depicted on the upper-right part of Fig. 4). In other words, rule application is reduced into examining at most 2-hop paths in \textit{G}. For instance, \texttt{IR6} requires two hops: from \textit{C1} to \textit{C2} through \textit{R} and from \textit{C2} to \textit{C3} through \textit{R} (the axiom \texttt{R1} \texttt{\circ R2} \texttt{\subseteq R3} of type 12 belongs to the metadata). The only exception is rule \texttt{IR4} which requires 3 hops: from \textit{C1} to \textit{C2} through \textit{R}, from \textit{C2} to \textit{C3} through \texttt{subClassOf}, and from \textit{C3} to \textit{C4} through \textit{R}'. The aforementioned property is crucial for the design of an I/O-aware algorithm; it implies that, if we traverse the graph \textit{G} 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 of \textit{G}, provided that this scan proceeds node by node. To perform the saturation, the new edges have to be added to \textit{G} and the process must be repeated until no new edges are created.

### 3.1 Storage Scheme

The graph \textit{G} and its metadata are stored in five relations:

- \texttt{RC}: Relation \texttt{RC} contains all edges that comprise the graph \textit{G} except those that correspond to axioms of type 4 and 8. \texttt{RC} has four fields: (i) \textit{R} which contains the edge label, (ii) \textit{C1} which stands for the source node, (iii) \textit{C2} which stands for the target node, and (iv) a 4-bit field \textit{T} which contains additional information about the edge. The three less important bits (2\(^3\) through 8) in \textit{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.

- In practice, relation \texttt{RG} contains the largest part of the data. It is also the only relation that is expanded with new tuples during the saturation.

- \texttt{R2}: Relation \texttt{R2} contains all axioms of type 2 (\texttt{C1} \texttt{\cap C2} \texttt{\subseteq C3}). It has three fields, one for each class that appears in the axiom: (i) \textit{C1} for the first conjunction class, (ii) \textit{C2} for the second conjunction class, and (iii) \textit{C3} for the subsumer class.

- \texttt{R11}: Relation \texttt{R11} stores all axioms of type 11 (\texttt{R1} \texttt{\subseteq R2}) and has two fields: (i) \textit{R1} for the subsumer, and (ii) \textit{R2} for the subsumer property.

- \texttt{R12}: Relation \texttt{R12} stores all axioms of type 12 (\texttt{R1} \texttt{\circ R2} \texttt{\subseteq R3}) and has three fields: (i) \textit{R1} for the first property in the chain, (ii) \textit{R2} for the second property, and (iii) \textit{R3} for the subsumer property.

- \texttt{R18}: Relation \texttt{R18} stores all edges corresponding to axioms of type 4 and 8, i.e., \texttt{\exists R.C1 \subseteq C2} and \texttt{\exists R.Self \subseteq C}. It has three fields: (i) \textit{R} which is the edge label, (ii) \textit{C1} which is the source node (for axioms of type 8 this is the Self node), and (iii) \textit{C2} which is the target node.

The overall schema \texttt{Σ} is illustrated in Fig. 5. Note that classes in all tuples are represented by integer IDs. The relation \texttt{RG} is a temporary copy of \texttt{RC} 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 in Fig. 1. This is straightforward for axioms of types 2, 11 and 12 which get separate tables. For axioms of type 4 and 8, the homogenous representation in table \texttt{R48} leads to a 1-1 mapping since all these axioms are syntactically the same. For the rest of the axioms stored in \texttt{RG}, the \textit{T} attribute (which encodes the type of the axiom) guarantees that the mapping is 1-1.

The main motivation behind the algorithm we present in Section 4 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 \textit{G}. Specifically, we 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 \texttt{RG} which is sorted on the class IDs (\texttt{RG} is sorted…

\textsuperscript{1}Note that we only need 3 bits for the axiom types since four types of axioms in Fig. 1 are stored in separate relations.
on C1 whereas its copy R′G2 is sorted on C2 - 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 algorithm 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 (R2, R48, R11, and R12) 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, IR8, 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 R2 (C1 ∩ C2 ⊆ C3), R11 (R1 ⊆ R2) and R12 (R1 ∩ R2 ⊆ R3) 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 ⊆) on which the relations R(G) and R′G are sorted. Axioms of R2, R11 and R12 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 on the schema Σ of Fig. 5. The proposed *Batch Rule Application (BRA)* algorithm is equivalent to a semi-naive evaluation [43] 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) ⊆ G2 = C1, R2 (for identifying the 2-hop paths), and (ii) a subsequent join of the intermediate tuples with one of the static relations R2, R11, R12, and R48 (depending on the particular rule). To reduce the number of disk page accesses during the evaluation, we adopt the following strategy. First, we ensure the semi-naive evaluation by keeping the new tuples in a temporary relation ∆R(G). Instead of inserting ∆R(G) into R(G) and then perform a self join on R(G) as a naive algorithm would do at each step of the procedure, we join ∆R(G) with R(G) and merge these two relations afterwards. Second, the join between ∆R(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 described 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 ∆R(G) denoted as ∆R′G. In order to enable efficient merge-joins, R′G is sorted on the attribute C2 whereas R(G) is sorted on C1. Finally, a hash index is created for each static relation: R2 is hashed on C1, R11 on R1, R12 on R1, and R48 on C1 (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 the relations in Fig. 5 (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 ⊆ which 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 ∆R(G) (which is sorted on C1) and ∆R′G (which is sorted on C2). In the baseline version of our algorithm, ∆R(G) and ∆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 \( \Delta R_G \) 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 \( R_G \) and \( R'_G \) to denote the relations \( R_G \) and \( R'_G \) at the \( i \)-th step of the algorithm. Similarly, \( \Delta R_G \) and \( \Delta R'_G \) are used for \( \Delta R_G \) 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 \( i \geq 1 \), \( R_G^{i-1} \) and \( R'_G^{i-1} \) are already sorted in the previous step \((i-1)\), so we only need to sort \( \Delta R_G \) and \( \Delta R'_G \). The latter are sorted on the attributes \( C_1 \) and \( C_2 \) respectively, 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 \( \Delta R_G \) and \( R'_G^{i-1} \) (line 9), and another one between \( \Delta R'_G \) and \( R_G^{i-1} \) (line 10). After that, \( \Delta R_G \) is joined with \( R'_G^{i-1} \) (line 11), and then it is merged with \( R_G^{i-1} \) into the relation \( R_G^{i} \) (line 12) so that the latter remains sorted on \( C_1 \). Analogously, \( \Delta R'_G \) is joined with \( R_G^{i-1} \) (line 13), and then it is merged with \( R'_G^{i-1} \) into the relation \( R'_G^{i} \) (line 14) so that the latter remains sorted on \( C_2 \). Note that \( \Delta R_G \) is joined with \( R'_G \), not \( R'_G^{i-1} \), which implies a join of \( \Delta R'_G \) with both \( R'_G^{i-1} \) and \( R_G^{i-1} \) (see also Fig. 5). All tuples produced by the two \( el \)-joins are stored in the relations \( \Delta R_G^{i+1} \) and \( \Delta R'_G^{i+1} \) which are used in the next step. When the loop is over, the saturated collection of axioms 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 [40], 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 \( R_G \bowtie R'_G \) \( C_2=R_G \sqcap R'_G \ T=f(R_G, T) \ R_G \) and the second is the self join \( R_G \bowtie R'_G \) \( C_1=R_G \sqcap R'_G \ T=f(R'_G, T) \ R'_G \), 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 the second type combines all \( outgoing\ edges \) of a node \( C_1 \). Based on the common operations in different rules, the \( el \)-join partitions the rules into the following classes:

- **Class 1** contains the rules IR5, IR12 and IR13. To evaluate these rules, we need to examine (i) every edge labeled with a role \( R \) on the graph, and (ii) the metadata of \( G \). This is translated into a join between \( R_G \) and the static relations \( R_{18} \) 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 pairs of \( outgoing \) edges for each graph node

---

\(^3\)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.

---

**Algorithm: el-JOIN**

**Input**: relations \( S, P, R_2, R_{11}, R_{12}, R_{48}; \)

**Output**: relations \( \Delta R_G^{i+1} \) and \( \Delta R'_G^{i+1} \);

**vars**: \( U, Q \); buffers;

1. let \( U \) and \( Q \) be the memory buffers for \( S \) and \( P \) respectively;
2. while there are tuples to join on \( S, C_2 = P.C_1 \) do
3. identify the minimum \( C_1 \) value in \( Q \), let \( min_Q \);
4. for each undetermined tuple \( t \in Q: C_1 = min_Q \) do
5. trigger active rules of Class 1; // IR5, IR12, IR13
6. apply active rules of Class 2; // IR15
7. // self join on \( P \)
8. for each undetermined 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
9. trigger active rules of Class 3; // IR2, IR8, IR10, IR14
10. apply active rules of Class 4; // IR16
11. // join \( S \) and \( P \)
12. for each undetermined 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
13. // read as many blocks of \( S \) and \( P \) needed
14. apply rules of Class 5; // IR1, IR3, IR7, IR11
15. trigger rules of Class 6; // IR4, IR6, IR9
16. if no undetermined tuples exist in \( U \) and \( Q \) then
17. if \( Q \) has to be reloaded and there are active rules in Classes 3 or 4 then
18. shift tuples in \( Q \) with the maximum \( C_1 \) value to the beginning of the buffer;
19. remove from \( U \) and \( Q \) all considered tuples except the shifted ones (if any);
20. reload \( U \) and/or \( Q \) with the next blocks of tuples;
21. while \( P \) is not exhausted do
22. repeat lines 3-9.
23. if there are active rules in Classes 3 or 4 then
24. shift tuples in \( Q \) with the maximum \( C_1 \) value to the beginning of the buffer;
25. remove from \( Q \) all tuples except the shifted ones (if any);
26. reload \( Q \) with the next blocks of tuples;

\( C_1 \) and also the metadata. This is reflected in 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 paths of three edges in the graph where the last edge is of type 4 (\( \exists R.C_1 \subseteq C_2 \)) that is stored in \( R_{18} \). 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 role chains (\( R_1 \circ R_2 \subseteq R_3 \)). 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 \( R_{18} \) 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 \( R_G^{i+1}, R_G^{i-1} \) and \( \Delta R'_G \), whereas \( P \) for one of \( R_G, \Delta R'_G \).
and \( R_{\Delta_i} \). The output of the operator are the relations \( \Delta R_{i+1} \) and \( \Delta P_{i+1} \). 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 2-17), and completes the evaluation with interleaving joins between streams of tuples and the static relations of the schema \( \Sigma \). 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 of el-Join 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 or 6. 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 (\( C_1 \subseteq \exists R_1.C_2 \land C_2 \subseteq \exists R_2.C_3 \)) as shown in Fig. 3. The remaining rules (Classes 2, 4 and 5) are evaluated as a whole within the main merge-join.

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 and applies the rules of Class 2 (lines 4-6). Then, it applies the rules of Class 3 (partially) and also the rules of Class 4 by performing a self join on \( Q.C_1 \) (lines 7-9). Joins of the form \( P \models_{P.C_1 = P.C_1, P.T = 1(P.T)} P \) are evaluated here and the intermediate tuples (for the rules of Class 3) are pushed to the sub-operators as we mentioned before. In the next step, it checks for rules of Class 5 and 6 (lines 10-12), so it evaluates the join patterns of the form \( S \models_{S.C_2 = P.C_1, S.T = f(P.T)} P \). 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 from 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 \( Q \). 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 \( Q.C_1 \) (Classes 3 and 4), 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 the shifting operation in the example provided below. The activation/deactivation of the rules is needed to avoid naive evaluation, i.e., checking previously checked axiom combinations. For instance, rules of Class 1 must be evaluated only on the new tuples in \( P \). To this end, they are active only in the initial join of the Algorithm BRA (line 4) and also in the join \( R_{\Delta_i} \) (line 11). A similar optimization is used for the rules of Class 2, 3 and 4.

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 18-24. There, the remaining blocks of \( P \) are fetched from disk with sequential scans and the procedure in lines 3-9 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 applying the rules, as we scan \( S \) and \( P \), is dynamic and driven by the underlying data.

External joins with the static relations of \( \Sigma \) are only needed in the rules of Classes 1, 3 and 6, hence, these joins occur in lines 5, 8 and 12 of el-Join. 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 \( P.C_1 \) (line 8) and the join \( S \models_{S.C_2 = P.C_1} P \) (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 the common join patterns that involve a static relation. When an external join is performed, the final tuples are sent to the deltas.

**Example.** The way the rules are applied is highlighted in the following example. Recall 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. 6. 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 so far. 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 \((T=3), 4, 9, 15\) \((C_1 \sqsubseteq \exists R_1C_2 \land C_2 \sqsubseteq \exists R_2C_3)\), and pushes the intermediate tuple to the stream of \(R_{12}\). Proceeding with the next \(\text{min}Q\) 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 \((T=7), 10, 14, 6\) \((C_1 \sqsubseteq \exists R_1C_2 \land C_2 \sqsubseteq \exists R_2\text{Sel}(f))\). The intermediate tuple is sent to the stream of \(R_{12}\) just like in the case of IR6 before. Now, the next \(\text{min}Q\) 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 \(C_1 \sqsubseteq C_2 \land C_1 \sqsubseteq C_3\) 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 delta. 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. This pair of tuples belongs to the workload of IR10. After the rule is applied, there are no more unconsidered tuples in \(Q\) so the buffer must be loaded with the next tuples from \(P\). Note that the next (disk-resident) tuple from \(P\) in the example is \((T=7), 16, 7, 6\) 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 \(C \sqsubseteq \exists R_1\text{Sel}(f) \land C \sqsubseteq \exists R_2\text{Sel}(f)\)). For this reason, the last two tuples are not discarded from \(Q\) during the load operation; they are shifted to 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 the logical optimizations in the algorithm BRA. The effectiveness of these techniques is highlighted in the experimental section.

The basic optimization we perform in BRA focuses on pruning tuples from the deltas. This optimization works as follows. In the general case, to guarantee that a rule has considered every combination of old and new axioms, the axioms have to be inserted into both \(\Delta R_G\) and \(\Delta R_C\). 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 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 \((C \sqsubseteq \exists R\text{Sel}(f))\). These axioms are only considered with other axioms that share a common \(C\) value. Based on this observation we can state the following Lemma (proof is given in [35]):

**Lemma 1.** The axioms produced by rules IR11, IR12 and IR15 can be omitted from \(R^\prime_G\) and \(\Delta R^\prime_G\) without altering the final result of the algorithm.

Moreover, the rule IR1 describes a typical transitive property, thus, it can be evaluated in a left- or right-linear fashion [43]. In our setting where all rules are evaluated together, this optimization requires special treatment because the tuples produced by IR1 are actually updated 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 \(\Delta R_C\) without altering the final result of the algorithm.

In other words, axioms of type 1 \((C_1 \sqsubseteq C_2)\) are initially replicated to \(R_2\) and \(R_C\) but, in the subsequent iterations, the axioms of type 1 that are produced by IR1 are only inserted in the right delta \(\Delta R_C\).

Finally, we manage an additional minor optimization by applying the rules of Classes 1, 2, 3 and 4 in the merge phase of BRA. We refer the interested reader to [35] for further details.

6. Theoretical Properties

In this section we present the theoretical properties of our algorithm and provide the respective proofs. The two basic properties we discuss are: (i) its correctness and (ii) its efficiency. In the former case we show that BRA 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 naïve evaluation.

**Correctness.** The proposed algorithm and its optimizations produce 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 optimizations of Section 5 do not affect the final result.

To prove point (i), we have to show that el-join applies each rule once on the input data, and also that the exhaustive application of each rule leads to the same result with an alternative algorithm that applies the rules sequentially. We have the following Lemma:

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

The above Lemma is proved by showing that BRA produces (i) all correct axioms and (ii) no additional axiom. The detailed proof can be found in [35]. The proof of correctness for the pruning optimizations is actually the proof of Lemmas 1 and 2. The proof of Lemma 1 is straightforward and we omit it for the interest of space (it can be found in [35]). The proof for Lemma 2 follows:

**Proof.**Lemma 2. (Sketch) Tuples of type 1 \((C_1 \sqsubseteq C_2)\) 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 \(\Delta R_G\) [43]. 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 \(t_p.C_2 = t_n.C_2\). Then, consider a tuple of type 3 such that \(t_{< i}C_2 = t_{> i}.C_1\). According to IR3, tuples \(t_p\) and \(t\) must be joined in order to produce a new tuple \(t'\) such that \(t'.C_1 = t_1.C_1\) and \(t'.C_3 = t_n.C_3\). In case \(t_p\) is appended only to \(\Delta R_G\), 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 \((t_1, t_1)\) 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 (i) each tuple of type 3 exists on the right side, and (ii) the set \(E\) of tuples exists on the left side. The former is true since every tuple is appended by el-Join to the right delta. 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 \(R_C\) in IR1.
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 (transitive closure). In this case, the tuple was produced (and, hence, it can be substituted) by a chain of tuples that form a path in \(G\), like the path represented by \(E\). By inductively applying the same substitution process to each
tuple produced by IR1 in this chain, we result with a set of tuples that define the needed path and exist on the left side (R′_C) due to 1 or 2. The proof for IR3 can be easily adapted for IR11.

**Semi-naive Evaluation.** The performance of BRA heavily depends on the semi-naive evaluation of the inference rules. We have the following Lemma:

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

A detailed proof for Lemma 4 can be found in [35]. Here we only highlight the basic idea. For the rules of Classes 5 and 6 that include a join between the left (R′_C) and the right (R_C) relation of Fig. 3, the semi-naive evaluation is ensured with the use of the deltas. However, for the rules of Class 3 and 4, e.g., IR2, that require a self join on R_C, the deltas are not adequate for semi-naive evaluation. The join on R_C or R′_C is applied by scanning R_C when it has already been merged with ∆R_C. In this case, the unconsidered pairs of tuples in c-l-join are distinguished with the use of the leftmost-bit of the T attribute of each tuple (denoted with red colour in Fig. 5 and 6). If this bit is set, it means that the tuple is produced in the previous step of the iteration, thus, it is a “new” one; otherwise, it is an “old” one. Tuples having these bits unset are never re-joined one another under the same rule. A similar idea is also employed for the semi-naive evaluation of the remaining rules (Classes 1 and 2).

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 sequentially, and also with the state-of-the-art inference algorithms from the related work.

**Experimental setting.** We compared BRA with the state-of-the-art systems in two areas of related work: Prolog-based systems (YAP [7] and XSB [6]), and 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 that 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. 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′_C, C2 and another one on R_C, C1, 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, nor even as a contribution of this work; 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 also 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 ontologies 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 ker-
nel’s caching mechanism by performing the I/O operations with the Linux O_DIRECT flag. Note that the reported results do not include the time of normalization (cf. Section 2) which is very small (11 seconds for SNOMED CT and 29 seconds for GALEN8) and is needed by all algorithms, including the competitors.

**Comparison with other methods.** We compare all algorithms in terms of running time, and also in terms of disk page accesses for disk-based methods (each page is 4KB in all experiments). The running time for disk-based methods is split into the CPU time (denoted with black colour) and the I/O time (denoted with light grey - it is estimated as the total time minus the CPU time). 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 more than three hours. We observe that BRA outperforms all competitor systems (except YAP and XSB on SNOMED CT) even if they perform in main memory. BRA-M slightly outperforms YAP even for SNOMED CT (60 secs for BRA-M vs 68 secs for YAP). The inferior performance of BRA on SNOMED CT compared to YAP and XSB (although slightly worse than XSB) is a result of the trade-off it provides between the memory requirements and the evaluation time. YAP and XSB are main-memory systems and the only I/Os they perform (as all main-memory competitors) are those needed for loading the input axioms from disk (these I/Os are negligible). On the other hand, BRA is a disk-based algorithm and its performance is dominated by the I/O time; whenever the buffers are full, it has to write the data back to disk (and read them again in a subsequent step). The memory requirements of all competitors 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. Finally, we tried to compare BRA with OWLIM [21] and Jena [1], but these systems failed to operate on both datasets (they operated for more than 24 hours without any result).

**Understanding BRA’s performance factors.** To understand the impact of (i) the logical optimizations and (ii) the storage scheme in the overall performance of the algorithm, BRA was compared with the basic semi-naive strategy SN and also with BRA-A. The results are depicted in Fig. 7.c and Fig. 7.d in terms of time and I/Os respectively. The optimizations we propose give a 28% speedup to BRA with respect to SN (on both datasets) whereas the storage scheme depicted in Fig. 5 results in a significantly faster evaluation for BRA, that is, 30% faster on SNOMED CT and 20% faster on GALEN8 with respect to BRA-A. Note that the performance time of SN is worse than that of BRA (due to the lack of optimizations - cf. Section 5), still, it is much better than that of ORT (in terms of both time and I/Os). On the other hand, BRA-A performs worse than BRA because it incurs an increased number of random I/Os. Since it has to scan more than two 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 in BRA-A must be split according to the base relations and, hence, each relation is assigned with a smaller buffer. Finally, BRA needs around 34% of the ORT time on 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 GALEN8 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 grows according to the second multiplication method. Again the behaviour is linear, but we notice that the effect of the pruning heuristics 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), IR3 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.

Summary. The performance gains of BRA stem from the batch application of the rules as described in Section 4. This is highlighted by the superior performance of BRA with respect to ORT. The storage scheme we propose in Section 3 improves the performance of BRA with respect to BRA-A by reducing the random I/Os (for reading and writing) during the evaluation of the inference rules. In addition, the logical optimizations of Section 5 further improve the performance of BRA with respect to SN by reducing the number of duplicated tuples at each step of the iteration. As a final comment, there are no I/O-aware methods for performing saturation in SROEL, and even the approaches that can solve the problem in main memory cannot outperform BRA in most cases (the only exceptions are YAP and XSB on SNOMED CT).

8. RELATED WORK

Artificial Intelligence. Regarding the fragment of OWL2-EL, the rule-based reasoning 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 [30] 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 “\(\text{subClassOf}(c,X)\)” 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 “\(\text{subClassOf}(X,Y)\)” which have no bounded variables. Regardless of 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 inferring axioms that are irrelevant to the query [20]. However, in the saturation process 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 [42]). On the other hand, a bottom-up strategy for a query “\(\text{subClassOf}(X,Y)\)” 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 saturation, i.e., when the algorithm has to compute “everything”. The algorithm BRA we described in Section 4 follows a pure bottom-up semi-naive evaluation. The only difference of BRA from a typical bottom-up algorithm is that BRA applies all rules in batch and within the same I/O operations whereas the existing algorithms apply the rules sequentially. As a final comment, the well-known Magic Sets optimization [38, 20] has been proposed for reducing the irrelevant facts in a bottom-up evaluation of a (partially) bounded query, hence, a saturation algorithm cannot benefit from it.

An interesting db-oriented approach for ontology management is currently 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. The theoretical framework presented in these papers is targeted to query answering and it is not optimized for saturation.

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, [34] 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 [39, 40]. 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. [37] and [45] are motivated by this 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. These systems offer generic rule engines where the OWL2-EL rules can be defined manually. Jena supports top-down and bottom-up evaluation whereas OWLIM has a pure bottom-up engine.

9. CONCLUSIONS

In this paper we proposed an I/O-aware algorithm that can efficiently saturate large OWL2-EL ontologies under a set of complex inference rules. 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.

10. REFERENCES