Transaction and Incremental Type Inference from Data Updates

YU LIU* AND PETER MCBRIEN*

Department of Computing, Imperial College London, 180 Queen’s Gate, London, SW7 2AZ, UK
*Corresponding author: yu.liu11@imperial.ac.uk, p.mcbrien@imperial.ac.uk

A distinctive property of relational database systems is the ability to perform data updates and queries in atomic blocks called transactions, with the well-known atomicity, consistency, isolation and durability (ACID) properties. To date, the ability of systems performing reasoning to maintain the ACID properties, even over data held within a relational database, has been largely ignored. This article studies an approach to reasoning over data from web ontology language (OWL) 2 RL ontologies held in a relational database, where the ACID properties of transactions are maintained. Taking an incremental approach to maintaining materialised views of the result of reasoning, the approach is demonstrated to support a query and reasoning performance comparable to or better than other OWL reasoning systems, yet adding the important benefit of supporting transactions.

Keywords: OWL; incremental reasoning; DBMS transactions; ACID properties; materialised views

Received 1 November 2015; revised 1 June 2016
Handling editor: Sebastian Maneth

1. INTRODUCTION

Many approaches to reasoning over knowledge bases take a query-rewriting approach (e.g. Ontop [1], Stardog [2], DLDB [3]), where a query over the knowledge base is rewritten to a (often complex) query over the base facts in the knowledge base. When the number of queries made on the knowledge base greatly exceeds the number of updates, it might be more efficient to adopt a materialised approach (e.g. OWLIM [4], WebPIE [5], RDFox [6], Oracle’s Resource Description Framework (RDF) store [7], Minerva [8]), where the extent of the knowledge base is calculated after updates to the knowledge base, and hence queries are answered directly from the inferred facts.

Even if data is stored in a relational database, such as in Minerva, the reasoning in materialised approaches is normally conducted outside of the core relational database management system (RDBMS) engine, and hence fails to provide transactional reasoning [9]. In transactional reasoning, the result of reasoning from data is available at the commit of any transaction that inserts or deletes data, and hence reasoning obeys the normal atomicity, consistency, isolation and durability (ACID) properties [10] of transactions.

This article considers reasoning over knowledge bases expressed in the RL profile [11] of the web ontology language (OWL) [12] (a.k.a. OWL 2 RL). It restricts itself to consider the issue of efficiently handling ontology queries where there are updates occurring to the assertion box (A-Box) (in database terms the data) and not to the terminology box (T-Box) (in database terms the schema), and the number of queries greatly exceeds the number of updates. Hence, the reasoning performed is type inference (i.e. deriving for each instance its membership of classes and properties), and it adopts a materialised approach. This article sets out to provide transactional type inference for ontologies held in an RDBMS, providing type inference over OWL 2 RL ontologies that can fully integrate with the data of existing RDBMS applications, and maintain the full ACID properties of transactions.

To illustrate the issues addressed, consider a T-Box with three rules:

\[
\begin{align*}
\text{Man} & \subseteq \text{Person} \\
\text{Parent} & \subseteq \text{Human} \\
\text{Person} & \equiv \text{Human}
\end{align*}
\]

which define that (1) every man is a person, (2) every parent is a human, and (3) a person is equivalent to a human. Now suppose that A-Box for the ontology is held as four tables Man, Parent, Person and Human in a database, and
four transactions, $T_1$ inserting Man(John), $T_2$ inserting Person(John), $T_3$ deleting Man(John) and finally $T_4$ deleting Parent(John), are executed. The expected changes of the database are illustrated in Fig. 1.

The first transaction $T_1$ should change the state of the database from $S_0$ to $S_1$. After executing $T_1$, John should be viewed not only from Man but also from Person and Human, because John should be inferred as both a Person and a Human as a result of Rules (1) and (3). Transactional type inference requires that any other transaction $T_c$ concurrent with $T_1$ should view the database either as $S_0$ or $S_1$, but not any intermediate state. For example, the query Man(John) $\land$ ~Person(John) should always evaluate to false in $T_c$. Next, $T_2$ executes, and adds Parent(John). Note that Rule (2) infers Human(John) from Parent(John), and from that Rule (3) infers Person(John). However, these two inferences were already made during the execution of $T_1$, and therefore should not be added to the database again.

Transaction $T_3$, which only deletes Man(John), should not delete inferred facts Person(John) or Human(John) (although these two facts were inserted because of inserting Man(John)), since Parent(John) and Rules (2) and (3) can still infer them (i.e. the database is changed to $S_3$). However, the same inferred facts must be deleted when Parent(John) is deleted in $T_4$, returning the database to an empty state.

Furthermore, we need to reject user attempts to delete implicit facts. For example, when in database states $S_1$, $S_2$ or $S_3$, allowing a user to delete Person(John) would make the knowledge base inconsistent, and therefore such an update must be rejected.

Our approach is based on incremental type inference, and uses the Delete & Rederive (DRed) algorithm [13] for incrementally maintaining materialised views (which might be recursive when considering OWL 2 RL). A feature of DRed is that it does not keep any additional information at the stage of materialising derived facts. When deleting explicit facts from base relations which form the materialised view, it first ‘over deletes’ all facts from the view which can be derived from the deleted facts, and then rederives some facts which are still inferable from the remaining facts. Hence, this algorithm is inefficient when derived facts have many different derivations, and have relations to other inferred facts [14]. In this article, we present a variant of DRed, and outline our implementation of this variant using RDBMS triggers, which support transactional and incremental type inference. Our approach has the following advantages:

- We assign each fact a state when materialising data. Then, deletions over the database invoke triggers to update the state of related facts, which reduces the number of real deletes and reinserts.
- The triggers in the RDBMS will be invoked whenever a user updates the database; consequently, our approach preserves ACID properties of transactions to cover the results of reasoning.
- Since our approach materialises the results of reasoning, it supports more efficient query processing compared to non-materialising approaches. The reason for this is that non-materialising approaches require queries to be rewritten into often complex sub-queries. A materialised approach completely avoids such costs during query processing, at the expense of using more storage, and as our evaluation confirms, results in faster query processing times at the expense of slower update processing compared to non-materialising approaches.
- Our approach can be incorporated into any standard RDBMS supporting triggers, in order to enhance their database schemas with type inference reasoning.

We have implemented this approach as an extension of SQOWL [15], which provided type inference only after inserts were made to an RDBMS, to now perform type inference after both inserts and deletes. As will be seen, maintaining the materialised view after deletes are made is a significantly more complex problem. We call our system SQOWL2, and provide this first RDBMS-based system supporting transactional and incrementally materialised type inference. We show that the completeness of query processing is comparable to the same task for other rule-based engines (e.g. OWLim). In addition, SQOWL2’s query processing is shown to be more efficient than comparable systems (i.e. Stardog and OWLim).

This article presents a substantially extended version of Ref. [16], and is organised as follows. In Section 2, we review the OWL and DRed, on which the contents of this article are based. Section 3 demonstrates our approach, especially the process of generating triggers, which are then used for incremental type inference. Section 4 extends Ref. [16] by providing a complete analysis of how all OWL 2 RL axioms are handled. Note that because we adopt a unique name assumption (UNA) this analysis excludes sameAs. We also provide an analysis about the completeness of our approach. In Section 5, we give a more detailed discussion about our SQOWL2 implementation, and in
Section 6, we provide an extended evaluation of our system compared to that in Ref. [16], in particular discussing how the database may be tuned to improve performance. Section 7 provides a brief summary of similar systems, and finally, Section 8 draws some conclusions from this work.

2. PRELIMINARIES

In this section, we briefly introduce both OWL and the DRed algorithm, which aids understanding of the main content of this article.

2.1. Web ontology language

OWL is a knowledge representation (KR) [17] language endorsed by the World Wide Web Consortium (W3C). It expresses information in a knowledge domain as an ontology; in particular, the structure of a knowledge domain is represented in a T-Box, and known facts are stored as an A-Box.

OWL currently has two major releases; the latest version OWL 2 [18] extends the previous OWL 1 in terms of expressiveness. Because of the extension, reasoning over OWL 2 ontologies requires a more complex reasoner than OWL 1. To mitigate the complexity of this reasoning, OWL 2 also provides three profiles [19] (i.e., sub-languages), called OWL 2 EL, OWL 2 QL and OWL 2 RL, each of which is a subset of the full OWL 2 language, and is useful in specific application scenarios.

The T-Box is the schema (or structure) of the ontology. A class (e.g., Man, Parent, Person and Human in Rules (1)–(3)) is used to denote a set of individuals (e.g., John). A property is a binary relation between individuals (each instance of a property being a pair of individuals). For example, we can define a property called hasParent to relate people to their parents. Besides classes and properties, OWL provides various constructors to represent more complex axioms. For example, the constructor SubClassOf can be used to define an axiom of a subsumption relation between two classes, and EquivalentClasses can denote an axiom that two classes contain exactly the same set of individuals. Apart from Rule (1)–(3) (axioms using SubClassOf or EquivalentClasses expressed in the description logic (DL) syntax [20]), we can extend the family T-Box with more axioms as follows (classes by convention begin with upper case letters, and properties by convention start with lower case letters):

\[
\text{Father} \equiv \text{Man} \sqcap \text{Parent} \tag{4}
\]

\[
\top \sqsubseteq \forall \text{hasParent}.\text{Person} \tag{5}
\]

\[
\top \sqsubseteq \forall \text{hasParent}.\text{Parent} \tag{6}
\]

\[
\text{hasParent} \sqsubseteq \text{hasAncestor} \tag{7}
\]

\[
\text{hasParent} \circ \text{hasParent} \sqsubseteq \text{hasGrandparent} \tag{8}
\]

where Rule (4) uses the construct IntersectionOf to state that fathers are things that are both a man and a parent. Rule (5) restricts the subjects of hasParent to be individuals from Person by defining the Domain of this property. Similarly to Rule (5), Rule (6) specifies the Range of hasParent to be Parent (i.e., to define that the objects of this property are individuals from Parent). Rule (7) demonstrates a subsumption relation from the property hasParent to hasAncestor by using SubPropertyOf. Rule (8) defines that a property hasGrandparent is a super property of a PropertyChain (symbol ○) formed by concatenating hasParent and hasParent. Finally, Rule (9) uses a construct TransitiveProperty to add the transitive semantics to the property hasAncestor, represented by making a PropertyChain be a subset of the property being chained.

The A-Box is the data (or the set of known facts) in the ontology, and is stated as ground atoms over the classes or properties of the T-Box. A sample A-Box for the family ontology is as follows:

\[
\text{Man}(\text{John}) \tag{10}
\]

\[
\text{Man}(\text{Lewis}) \tag{11}
\]

\[
\text{Father}(\text{John}) \tag{12}
\]

\[
\text{Father}(\text{Jack}) \tag{13}
\]

\[
\text{Parent}(\text{John}) \tag{14}
\]

\[
\text{hasParent}(\text{Jack}, \text{Mike}) \tag{15}
\]

\[
\text{hasAncestor}(\text{John}, \text{Jack}) \tag{16}
\]

where Rules (10)–(14) are five class facts, which define the membership of individuals. Rules (15) and (16) are two property facts, each of which specifies that an individual is related to another by a property.

The Semantics of OWL 2 considered in this article is the Direct Semantics [21] (denoted as OWL 2 DL), which is based on the DL SROIQ (D). The alternative RDF-based Semantics [22] has undecidable reasoning. The OWL 2 DL semantics is decidable, but not tractable (i.e., computation of typical reasoning tasks is beyond a polynomial time complexity), and thus is not suitable for ontologies with large A-Boxes. Therefore, we instead focus on the OWL 2 RL profile, which aims to provide applications with scalable reasoning without losing too much expressivity. The profile is inspired from the description logic program [23] and pD* [24], and it enables applications to perform reasoning through a rule-based mechanism. OWL 2 RL defines a syntactic fragment of OWL 2 DL, in which the use of constructors is restricted in a syntactic way.
depending on their positions in axioms of the form \(x \sqsubseteq y\), as listed in Table 1. For example, an expression formed by a constructor \(\text{SomeValuesFrom}\) (i.e. \(\exists P.D\) in DL), which denotes an existential restriction, is not allowed as a superclass expression (i.e. \(y\)), but is allowed as a subclass expression (i.e. \(x\)). These restrictions avoid the cases of inferring the existence of individuals which are not explicitly present in the knowledge base, and also prevent nondeterministic reasoning.

In order to make the article more self-contained, we detail the semantics of OWL 2 RL by reviewing its interpretation \(I\):

**Definition 2.1.** An interpretation \(I\) of OWL 2 RL consists of a non-empty set of objects \(\Delta\) (a.k.a. the knowledge Universe), and an interpretation function \(I\), which maps some set-theoretic elements of \(\Delta\) to permitted expressions. For the most basic OWL components, namely, classes, properties and individuals, \(I\) associates:

- each individual \(a\) with an object in the Universe: \(a^I \in \Delta\),
- each class \(C\) with a set of objects: \(C^I \subseteq \Delta\),
- each property \(P\) with a subset of the Cartesian product of the Universe: \(P^I \subseteq \Delta \times \Delta\).

In a nutshell, \(I\) defines the formal semantics of OWL 2 RL by explaining its permitted expressions in a set-theoretic way. Apart from interpreting the classes, properties and individuals, Table 1 interprets other allowed expressions, and also provides the possible positions each expression can appear in an OWL 2 RL axiom.

As can be seen from the table, except the last two lines which are about interpreting property expressions, the remaining interprets class expressions. For example, the \(\text{IntersectionOf}\) expression \(C \cap D\) is interpreted as the common individuals belonging to both classes \(C\) and \(D\). These expressions can be used to construct OWL 2 RL axioms as exemplified as Rules (1)–(9). The interpretation \(I\) also defines under which conditions can an axiom hold. We denote by \(I \models \text{axiom}\) the case that the \(\text{axiom}\) holds under appropriate conditions defined in \(I\) (a.k.a. \(I\) satisfies the axiom). Take Rule (1), which denotes an axiom of a subsumption relationship between \(\text{Man}\) and \(\text{Person}\), as an example, the condition under which \(I\) satisfies the axiom is illustrated as

\[I \models \text{Man} \sqsubseteq \text{Person} \iff \text{Man}^I \subseteq \text{Person}^I,\]

\(I\) also defines the truth condition for A-Box facts asserted by A-Box statements such as Rules (10)–(16). For example, \(I\) satisfying the class fact \(\text{John}\) is a \(\text{Man}\) in Rule (16) can be written as

\[I \models \text{Man(John)} \iff \text{John}^I \in \text{Man}^I.\]

Similarly, \(I\) satisfying the property fact \(\text{Jack}\) has \(\text{Mike}\) as his parent in Rule (15) can be written as

\[I \models \text{hasParent(Jack, Mike)} \iff \text{Jack}^I, \text{Mike}^I \in \text{hasParent}^I.\]

We summarise the appropriate conditions under which other OWL 2 RL axioms can be satisfied by \(I\) in Table 2.

### 2.2. Delete & Rederive algorithm

In Ref. [13], the DRed algorithm is introduced alongside another algorithm called counting, both of which can be used for incrementally maintaining materialised views. DRed does not need to store any additional information at the stage of materialising explicit and implicit facts. When a deletion of explicit facts is executed, it first deletes all the facts that might be affected because of the deletion, and then rederives a subset of these facts which is still inferable from the remaining facts. By contrast, the counting algorithm records the number of different derivations of each fact during the materialisation stage. When certain derivations of a fact are no longer valid (because of deletions), the

| Construct name | DL syntax | Direct semantics interpretation | Subclass exp. | Superclass exp. |
|----------------|-----------|---------------------------------|--------------|----------------|
| Class exp.     |           |                                 |              |                |
| IntersectionOf | \(C \cap D\) | \(C^I \cap D^I\)                | ✓            | ✓              |
| UnionOf        | \(C \sqcup D\) | \(C^I \cup D^I\)               | ✓            | x              |
| ComplementOf   | \(~C\)    | \(\Delta \setminus C^I\)        | x            | ✓              |
| AllValuesFrom  | \(\forall P.D\) | \(\{x\mid \forall y: \text{if } (x, y) \in P^I \text{ then } y \in D^I\}\) | x            | ✓              |
| SomeValuesFrom | \(\exists P.D\) | \(\{x\mid \exists y: (x, y) \in P^I \text{ and } y \in D^I\}\) | ✓            | x              |
| HasSelf        | \(P.Self\) | \(\{x\mid (x, x) \in P^I\}\)   | ✓            | ✓              |
| HasValue       | \(\exists P.\{a\}\) | \(\{x\mid (x, a^I) \in P^I\}\) | ✓            | ✓              |
| MaxCardinality | \(\leq P\) | \(\{x\mid \#\{y\mid (x, y) \in P^I\} \leq 1\}\) | x            | ✓              |
| OneOf          | \(\{a_1, \ldots, a_n\}\) | \(\{a_1^I, \ldots, a_n^I\}\) | ✓            | x              |
| Property exp.  |           |                                 |              |                |
| InverseOf      | \(P^{-}\) | \(\{(y, x)\mid (x, y) \in P^I\}\) | ✓            | ✓              |
| PropertyChain  | \(P\circ Q\) | \(\{(x, z)\mid (x, y) \in P^I \text{ and } (y, z) \in Q^I\}\) | ✓            | x              |
counting algorithm decrements the count of the fact. A fact must be removed when its count drops to zero.

DRed is claimed to be capable of handling recursive views, and thus can support OWL 2 RL constructs such as EquivalentClasses and TransitiveProperty, while the counting algorithm cannot correctly deal with recursive views [25].

We now use a simple example to demonstrate how DRed incrementally maintains a materialised view. By considering Rule (1) and the following new T-Box rule (every father is a person):

\[ \text{Father} \sqsubseteq \text{Person} \]  \hspace{1cm} (17)

the view of the OWL class Person (denoted as Person) can be defined as a union of the base relations of classes Man and Father (denoted as Man and Father, respectively) by using the following Datalog rules:

\[
\begin{align*}
\text{Person}(x) & :- \text{Man}(x) \\
\text{Person}(x) & :- \text{Father}(x)
\end{align*}
\]

Table 2. Satisfying OWL 2 RL axioms.

| Construct name | Truth condition |
|----------------|-----------------|
| Class axioms   | \( \text{I} \models \text{CE}_1 \sqsubseteq \text{CE}_2 \iff \text{CE}_1 \sqsubseteq \text{CE}_2 \) |
| EquivalentClasses | \( \text{I} \models \text{CE}_1 \equiv \text{CE}_2 \iff \text{CE}_1 \equiv \text{CE}_2 \) |
| DisjointClasses | \( \text{I} \models \text{CE}_1 \cap \text{CE}_2 \sqsubseteq \top \iff \text{CE}_1 \cap \text{CE}_2 = \emptyset \) |
| SelfRestriction | \( \text{I} \models \text{CE} \equiv \exists \text{PE}, \text{Self} \iff \text{CE} = \{ x | (x, x) \in \text{PE} \} \) |
| SameIndividual | \( \text{I} \models = (a_1, a_2) \iff a_1 = a_2 \) |
| DifferentIndividuals | \( \text{I} \models \neq (a_1, a_2) \iff a_1 \neq a_2 \) |
| Property axioms | \( \text{PE}_1 \sqsubseteq \text{PE}_2 \iff \text{PE}_1 \sqsubseteq \text{PE}_2 \) |
| EquivalentProperties | \( \text{I} \models \text{PE}_1 \equiv \text{PE}_2 \iff \text{PE}_1 \equiv \text{PE}_2 \) |
| PropertyDomain | \( \text{I} \models \forall x, y : (x, y) \in \text{PE} \text{ if } x \in \text{CE} \) |
| PropertyRange | \( \text{I} \models \forall x, y : (x, y) \in \text{PE} \text{ if } y \in \text{CE} \) |
| InverseProperty | \( \text{I} \models \text{PE}_1 \equiv \text{PE}_2 \iff \text{PE}_1 = \{ (y, x) | (x, y) \in \text{PE} \} \) |
| SymmetricProperty | \( \text{I} \models \text{PE} \equiv \text{PE} \text{ if } \forall x, y : (x, y) \in \text{PE} \text{ and } (y, x) \in \text{PE} \) |
| AsymmetricProperty | \( \text{I} \models \text{PE} \sqsubseteq \text{PE} \text{ if } \forall x, y : (x, y) \in \text{PE} \text{ and } (y, x) \not\in \text{PE} \) |
| DisjointProperties | \( \text{I} \models \text{PE}_1 \sqcap \text{PE}_2 \sqsubseteq \top \iff \text{PE}_1 \sqcap \text{PE}_2 = \emptyset \) |
| IrreflexiveProperty | \( \text{I} \models \neg \exists \text{PE}, \text{Self} \text{ if } \forall x : (x, x) \not\in \text{PE} \) |
| FunctionalProperty | \( \text{I} \models \forall x, y, z : (x, y) \in \text{PE} \text{ and } (x, z) \in \text{PE} \text{ if } y = z \) |
| InverseFunctionalProperty | \( \text{I} \models \forall x, y, z : (x, y) \in \text{PE} \text{ and } (z, x) \in \text{PE} \text{ if } y = z \) |
| TransitiveProperty | \( \text{I} \models \text{PE} \circ \text{PE} \sqsubseteq \text{PE} \text{ if } \forall y, z : (x, y) \in \text{PE} \text{ and } (y, z) \in \text{PE} \text{ if } (x, z) \in \text{PE} \) |
| PropertyChain | \( \text{I} \models \text{PE}_1 \circ \text{PE}_2 \sqsubseteq \text{PE}_3 \text{ if } \forall x, y, z : (x, y) \in \text{PE}_1 \text{ and } (y, z) \in \text{PE}_2 \text{ if } (x, z) \in \text{PE}_3 \) |
| NegativePropertyAssertion | \( \text{I} \models \neg \text{PE}(a_1, a_2) \iff (a_1, a_2) \not\in \text{PE} \) |

Over-deletion: In the phase of over-deletion, when explicit facts are deleted, DRed ‘over deletes’ a number of facts, if the facts have a derivation from the deleted facts. For example, if we delete John and Lewis from Man, DRed will ‘over delete’ John and Lewis from Person in this phase.

Rederivation: Following the above stage of over-deletion, this second stage is responsible for ‘rederiving’ some facts which have been deleted in the first phase, but are still inferable from the remaining data. Continuing our example of deleting John and Lewis from Man, in this rederivation stage, John in Father can still infer John as a member of Person, while no further information infers Lewis as an instance of Person; therefore, John (but not Lewis) will be rederived as a member of Person.

3. OUR APPROACH

This section describes what we call an auto type inference database (ATIDB). Our approach separates the T-Box and A-Box reasoning as shown in Fig. 2. While this separation does entail that the reasoning of a system is not complete [26], it is not uncommon in other large-scale reasoners, and as documented in Section 6, the completeness achieved matches that of other approaches.

The T-Box of an ontology is firstly classified by a tableau-based reasoner, such as Pellet [27], FaCT++ [28] or Hermit [29], in order to obtain the complete set of subsumption relationships w.r.t. the T-Box. For example, the T-Box Rule (4) can be classified as three additional subsumption rules.
The ATIDB version of the ontology can never enter an inconsistent state. The following properties of our system should be noted:

- For inserts into the database, triggers will ignore repeated inserts of the same data to avoid duplication.
- If the insert succeeds (i.e. because it is not a duplicate), it might cause triggers to execute to infer additional data, which might cascade, to infer additional data from other inferred data.
- For deletes from the database, a label & check process is performed to first recursively label all items of data that might be deleted because of this transaction, and then to check as to whether the labelled data can still be inferred or not from non-labelled data. If an item of labelled data is still inferable, then we keep this data item; otherwise, we remove the data.

3.1. The state of each data item

In the ATIDB, each value $x$ in the domain of a class $C$ as a fact $C(x)$ that passes through four states as outlined in Fig. 3 (a property fact $P(x, y)$ is also viewed through similar state transitions), where each state has the following semantics:

- $C(x)_{\theta}$: A fact that does not hold, and hence is not stored in the database.
- $C(x)_{e}$: A fact stored because an explicit A-Box statement asserts it.
- $C(x)_{i}$: A fact implicitly inferred from other facts.
- $C(x)_{d}$: A fact that has lost one of its supporting arguments for being in the database, and a process of checking if the data is still inferable from other data is being conducted.

The state of a fact can be changed by insert and delete operations. We identify two categories of inserts. An ontology insert means that some user or application is inserting a new explicit fact into the database, and is detected by the trigger when $C(x)_{i}$. By contrast, a reasoner insert means that a reasoner has derived some implicit fact from the existing facts in the database, and is detected by the trigger when $C(x)_{i}$. Similarly, an ontology delete is some user or application deleting an explicit fact from the database, and is detected by when $\neg C(x)_{e}$, and a reasoner delete is when some supporting evidence for a fact has been deleted, detected in triggers by when $\neg C(x)_{e}$. Figure 3 gives an overview of the possible state transitions which can occur. For inserts:

- For a data item which is not present in the table, $C(x)_{\theta}$. An ontology insert of $C(x)_{i}$ changes state $C(x)_{\theta}$ to $C(x)_{i}$, and a reasoner insert of $C(x)_{i}$ changes state $C(x)_{\theta}$ to $C(x)_{i}$. In essence, $\phi$ is a logical state that we apply for ease of expressing a fact which is not present
in the database, and as can be seen in our implementation, we do not explicitly store \( C(x)_0 \).

- For a data item stored in the implicit state \( C(x)_i \), further reasoner inserts of \( C(x)_i \) do not change the state, so that repeated inference of other facts based on \( C(x)_i \) is avoided. However, inserting \( C(x)_e \) gives explicit semantics, and thus updates the state \( C(x)_i \) to \( C(x)_e \).
- For a data item stored in the explicit state \( C(x)_e \), inserting \( C(x)_i \) does not change the state, in order to avoid duplicated inference. However, to implement normal database semantics, a rollback occurs if further attempts are made to insert \( C(x)_e \).

For deletes:

- For a data item \( C(x)_a \), ontology or reasoner deletes are ignored, to match the normal database semantics that deletes of data not present cause no errors.
- For a data item \( C(x)_i \), attempting an ontology delete \( \neg C(x) \) causes inconsistencies since the assertion of being no \( C(x) \) conflicts with what can be inferred from other facts, and the transaction should be rolled back. The reasoner delete \( \neg C(x)_i \), by contrast, changes \( C(x)_i \) to \( C(x)_d \), in order to label the data for checking.
- For a data item \( C(x)_e \), attempting the ontology delete \( \neg C(x)_e \) changes it to \( C(x)_d \), because the data might still be inferable even after removing the explicit semantics. However, a reasoner delete \( \neg C(x)_e \) does not change the state, since only ontology deletes can remove the explicit semantics.

Note that when the state of \( C(x) \) is updated to \( d \), a recursive labelling process is conducted to implicitly delete other data which depends on \( C(x)_d \). When the whole labelling process is finished, all data items labelled with \( d \) are checked as to whether they are inferable from data in state \( e \) or \( i \) (i.e. non-labelled). If they are still inferable, we change \( C(x)_d \) to \( C(x)_i \); otherwise, \( C(x)_d \) is updated to \( C(x)_a \) (i.e. deleted from the database).

### 3.2. Transactional and incremental type inference by triggers

Now, we demonstrate for the example in the introduction how our approach can achieve type inference in an incremental manner.

Figure 4 breaks down the execution of \( T_1 \) illustrated in Fig. 1 as a transition from \( S_0 \) to \( S_1 \) passing through two intermediate database states \( S_0 \) and \( S_0' \). Firstly, the attempt to insert \( \text{Man}(\text{John})_0 \) is checked by a before trigger (indicated by the \( \neg \) prefix):

\[
\text{when } \neg \text{Man}(x)_e \text{ if } \neg \text{Man}(x)_e \text{ then } \text{Man}(x)_e.
\]

Since \( \text{Man}(\text{John})_0 \) is true, the insert is permitted, and the database enters \( S_0 \). Rule (1) is translated into an after trigger (denoted by the \( + \) prefix):

\[
\text{when } + \text{Man}(x)_{e/0} \text{ then } \text{Person}(x)_{i/0}.
\]

Thus, after \( \text{Man}(\text{John})_0 \) is inserted, this trigger is invoked to infer a reasoner insert of \( \text{Person}(\text{John})_0 \), updating \( S_0 \) to \( S_0' \).

An equivalent relationship between two classes can be treated as two subsumption relations; therefore, Rule (3) is translated into two triggers:

\[
\text{when } + \text{Person}(x)_{e/0} \text{ then } \text{Human}(x)_{i/0},
\]

\[
\text{when } + \text{Human}(x)_{e/0} \text{ then } \text{Person}(x)_{i/0}.
\]

After the database is updated to the intermediate state \( S_0' \), the insert of \( \text{Person}(\text{John})_0 \) causes the attempt to insert \( \text{Human}(\text{John})_0 \), which changes \( \text{Human}(\text{John})_0 \) to \( \text{Human}(\text{John})_1 \) (i.e. \( S_0' \) is updated to \( S_1 \)). The insert of \( \text{Human}(\text{John})_1 \) generates the attempt to insert \( \text{Person}(\text{John})_1 \), again because of the after insert trigger on \( \text{Human} \). However, this attempt to insert \( \text{Person}(\text{John})_1 \) is ignored, because \( \text{Person}(\text{John})_1 \) is true in \( S_1 \) (i.e. the database stays at \( S_1 \)).

Figure 5 illustrates the process of executing \( T_2 \), which inserts \( \text{Parent}(\text{John})_0 \). \( T_2 \) first attempts to insert
Parent(John)\textsubscript{e}, which updates the state of Parent(John) from \(\emptyset\) to \(e\) (i.e. \(S_1\) is updated to \(S_2\)). Afterwards, the after insert trigger below on Parent from Rule (2):

\textbf{when } \textbf{ +Parent(John) \_e/vi, then Human(John)\textsubscript{i},}

generates a new reasoner insert of Human(John)\textsubscript{i}, which is then ignored by the before insert trigger on Human, since Human(John)\textsubscript{i} is already true (i.e. the database stays at \(S_2\)).

Attempting ontology deletes of data in state \(i\) without removing the explicit facts which infer them causes inconsistencies in the database. Thus, from the Person we design a before trigger:

\textbf{when } \textbf{ -~Person(John) \_e, if Person(John) \_i, then rollback},

so that attempting the ontology delete to Person(John)\textsubscript{i} results in the rollback of the transaction. In fact, we only allow users to delete explicit facts, \(C(x)\textsubscript{e}\) or \(P(x, y)\textsubscript{e}\). As a fact in state \(e\) may also be implicitly stated, when executing ontology deletes to such a fact, the following label & check process is conducted to determine if the fact is still implicitly inferred.

\textbf{Label:} When a user attempts to delete \(C(x)\textsubscript{e}\), a before delete trigger on \(C\) changes this to an update of \(C(x)\textsubscript{i}\) to \(C(x)\textsubscript{d}\). This update leads to reasoner deletes to data inferred from \(C(x)\textsubscript{i}\). For example, because Rule (1) means that data in Man infers the same in Person, an after trigger:

\textbf{when } \textbf{ +Man(John) \_i, then -~Person(John) \_i},

is generated to cause the reasoner to attempt the delete of \(x\) from Person after changing the state of \(x\) to \(d\) in Man. This reasoner delete is captured by a before trigger below:

\textbf{when } \textbf{ -~Person(John) \_i, if Person(John) \_i, then Person(John) \_d},

which continues to change the state of \(x\) to \(d\) in Person, since Person(John)\textsubscript{i} is true (i.e. the labelling process cascades).

Note that the above two triggers can be folded in as one single trigger:

\textbf{when } \textbf{ +Man(John) \_i, if Person(John) \_i, then Person(John) \_d},

which results in a more efficient implementation as demonstrated in Section 5.

Similarly, Rule (2) gives a trigger:

\textbf{when } \textbf{ +Parent(John) \_d \_if Human(John) \_i, then Human(John) \_d},

and Rule (3) gives two triggers for the purpose of labelling:

\textbf{when } \textbf{ +Person(John) \_d \_if Human(John) \_i, then Person(John) \_d},

\textbf{when } \textbf{ +Human(John) \_d \_if Person(John) \_i, then Person(John) \_d}.

Hence, an ontology delete cascades to attempt reasoner deletes on all inferred facts.

Thus, as shown in Fig. 6, \(T_3\), deleting \(\text{Man(John)}\), first changes \(\text{Man(John)}\textsubscript{e} \_d\) to \(\text{Man(John)}\textsubscript{d}\) as shown from \(S_2\) to \(S_3\). Then, a reasoner delete of \(\text{Person(John)}\), is generated, which leads to the change of \(\text{Person(John)}\) to \(\text{Person(John)}\textsubscript{d}\) (i.e. \(S_3\) to \(S_4\)). Similarly, after \(S_4\), the trigger created on \(\text{Person}\) for labelling causes reasoner delete of \(\text{Human(John)}\), changing its state to \(\text{Human(John)}\textsubscript{d}\). The labelling process is finished as the database becomes \(S_5\).

\textbf{Check:} All data items with the state \(d\) are checked to determine if they can be inferred from the data stated with \(e\) or \(i\). If so, the \(d\) is changed to \(i\), otherwise \(d\) is changed to \(\emptyset\). In order to perform the check, we create for each table a Datalog-style \textit{inference rule} (with heads \(C_{\text{int}}(x)\) or \(P_{\text{int}}(x, y)\)), that contains all the inference logic for the table. The checking is then conducted by the following trigger created on each table:

\textbf{when } \textbf{ +C(x) \_d \_if C_{\text{int}}(x) \_i, then C(x), else C(x)\_i}.

Obviously, if there is no way to infer data to a table, the inference rule is omitted. For example, when only considering Rules (1)-(3), there are no inference rules for tables \text{Man} and \text{Parent}. The inference rule of \text{Person} or \text{Human} contains two parts as data can be inferred from the subclass table and the equivalent class table:

\text{Person_{int}(x)}::\text{Man(John)\_e/vi}
\text{Person_{int}(x)}::\text{Human(John)\_e/vi}
\text{Human_{int}(x)}::\text{Parent(John)\_e/vi}
\text{Human_{int}(x)}::\text{Person(John)\_e/vi}

In Fig. 6, the checking starts from \(\text{Man(John)}\textsubscript{d}\) in \(S_2\). Since \(\text{Man_{int}(x)}\) is empty, \(\text{Man(John)}\textsubscript{d}\) is changed to \(\text{Man(John)\_d}\) (i.e. \(S_3\) to \(S_4\)). If \(\text{Person(John)\_d}\) is then checked, it is also changed to \(\text{Person(John)\textsubscript{d}}\) (i.e. \(S_3\) to \(S_4\)), because the state of John in Man is \(\emptyset\) and in Human is \(d\). Next, \(\text{Human(John)\_d}\) is checked and then changed to \(\text{Human(John)\_d}\) because of \(\text{Parent(John)\_e}\) (i.e. \(S_4\) to \(S_5\)). Because of the occurrence of \(\text{Human(John)\_d}\), the trigger for \(\text{inverting Person(John)\_d}\) is invoked, to change the database state from \(S_3\) to \(S_4\). Similarly, if we adopt the label & check process to deal with \(T_4\) which deletes \(\text{Parent(John)}\), the database is changed back to the empty one \(S_0\).
4. GENERATING TRIGGERS AND INFERENCE RULES

We now review how we implement all of the OWL 2 RL axioms. Note that since we are using the UNA, the `sameAs` can never be used, since it would imply that a single thing shared two names. First we review in detail how we represent classes and properties in a database context, and then review how class axioms, and then property axioms, are implemented. Finally, we analyse the soundness and completeness of our system.

4.1. OWL classes and properties

Class: For each OWL class, we specify before triggers that handle ontology and reasoner inserts, and perform the state transitions illustrated in Fig. 3:

\begin{align*}
C \rightarrow & \text{ when } \neg C(x)_e \text{ if } C(x)_a \text{ then } C(x)_e \\
& \text{ if } C(x)_i \text{ then } C(x)_e \\
& \text{ if } C(x)_e \text{ then rollback} \\
& \text{ when } \neg C(x)_i \text{ if } C(x)_a \text{ then } C(x)_i \\
& \text{ if } C(x)_e \text{ then ignore} \\
\end{align*}

and similarly before triggers to handle ontology and reasoner deletes:

\begin{align*}
C \rightarrow & \text{ when } \neg C(x)_e \text{ if } C(x)_a \text{ then ignore} \\
& \text{ if } C(x)_i \text{ then rollback} \\
& \text{ if } C(x)_e \text{ then } C(x)_d \\
& \text{ when } \neg C(x)_i \text{ if } C(x)_a \text{ then } C(x)_d \\
& \text{ if } C(x)_e \text{ then ignore} \\
& \text{ if } C(x)_i \text{ then } C(x)_d \\
\end{align*}

and finally after triggers to perform checking once the class fact enters the `d` state:

\begin{align*}
C \rightarrow & \text{ when } +C(x)_d \text{ if } C_{ht}(x) \text{ then } C(x)_i \text{ else } C(x)_a. \\
\end{align*}

Property: Handling the state transitions of a property fact is analogous to handling those of a class fact. Hence, for each OWL property we generate before triggers which handle ontology and reasoner inserts:

\begin{align*}
P \rightarrow & \text{ when } \neg P(x, y)_e \text{ if } P(x, y)_a \text{ then } P(x, y)_e \\
& \text{ if } P(x, y)_i \text{ then } P(x, y)_e \\
\end{align*}

and before triggers which handle ontology and reasoner deletes:

\begin{align*}
P \rightarrow & \text{ when } \neg P(x, y)_e \text{ if } P(x, y)_a \text{ then ignore} \\
& \text{ if } P(x, y)_i \text{ then rollback} \\
& \text{ if } P(x, y)_d \text{ then } P(x, y)_e \\
\end{align*}

and finally after triggers which are used for checking:

\begin{align*}
P \rightarrow & \text{ when } +P(x, y)_d \text{ if } P_{ht}(x, y) \text{ then } P(x, y)_i \\
& \text{ else } P(x, y)_a. \\
\end{align*}

With the above logical triggers, any ontology/reasoner insert/delete will be properly handled with consideration of not only persisting/deleting the data but also updating its state correctly.

4.2. The semantics of class axioms

We have already described how we implement `SubClassOf` and `EquivalentClasses`. OWL 2 RL also provides some other constructs to express particular restrictions to OWL classes. Here, we choose `DisjointClasses`, `AllValuesFrom`, `SomeValuesFrom`, `HasValue`, `MaxCardinality`, `MinCardinality`, `Cardinality`, `MinInverseCardinality`, and `MaxInverseCardinality` in addition to `DisjointUnion`, `Intersect`, and `Union`.

\[ a \text{ SubClassOf } b \]

\[ a \text{ EquivalentClasses } b \]

\[ a \text{ DisjointClasses } b \]

\[ a \text{ AllValuesFrom } b \]

\[ a \text{ SomeValuesFrom } b \]

\[ a \text{ HasValue } b \]

\[ a \text{ MaxCardinality } n \]

\[ a \text{ MinCardinality } n \]

\[ a \text{ Cardinality } n \]

\[ a \text{ MinInverseCardinality } n \]

\[ a \text{ MaxInverseCardinality } n \]

\[ a \text{ DisjointUnion } b \]

\[ a \text{ Intersect } b \]

\[ a \text{ Union } b \]
TABLE 3. Triggers and inference rules generated from class axioms.

| OWL constructs | DL syntax | Triggers | Inference rules |
|----------------|-----------|----------|-----------------|
| SomeValuesFrom | \( P \rightarrow D \subseteq C \) | when \( +P(x, y) \) if \( D(y) \) then \( C(x) \), \( +D(y) \) if \( P(x, y) \) then \( C(x) \), \( +P(x, y) \) if \( D(y) \) then \( \neg C(x) \), \( +D(y) \) if \( P(x, y) \) then \( \neg C(x) \) | \( \text{Infer}(x) : -P(x, y), D(y) \) |
| HasValue       | \( C \subseteq \exists P \{ a \} \) | when \( +C(x) \) then \( P(x, a) \), \( +C(x) \) then \( \neg P(x, a) \) | \( \text{Infer}(x) : -P(x, a) \) |
| SelfRestriction| \( C \equiv \exists P \cdot \text{Self} \) | when \( +P(x, a) \) then \( C(x) \), \( +P(x, a) \) then \( \neg C(x) \) | \( \text{Infer}(x) : -P(x, a) \) |
| MinCardinality | \( \geq n P \subseteq C \) | when \( +P(x, y) \) if \( \text{count}[P(x, \_)] \geq n \) then \( C(x) \), \( +P(x, y) \) if \( \text{count}[P(x, \_)] \geq n \) then \( \neg C(x) \) | \( \text{Infer}(x) : \neg \text{count}[P(x, \_)] \geq n \) |
| MinCardinality | \( \geq n P \cdot D \subseteq C \) | when \( +P(x, y) \) if \( \text{count}[P(x, y), D(y)] \geq n \) then \( C(x) \), \( +D(y) \) if \( \text{count}[P(x, y), D(y)] \geq n \) then \( \neg C(x) \) | \( \text{Infer}(x) : \neg \text{count}[P(x, y), D(y)] \geq n \) |

IntersectionOf and OneOf to demonstrate how axioms formed by them can be translated to triggers and inference rules. Triggers and inference rules translated from class axioms using other constructors are listed in Table 3.

DisjointClasses: OWL 2 RL uses DisjointClasses to state that two classes do not share any same individual. For example, the following T-Box rule:

\[ \text{Father} \cap \text{Mother} \subseteq \bot \]  

expresses that the class Father is disjoint with the class Mother. Thus, we require before triggers to reject an insertion (either ontology or reasoner insert) to a class table if the data of the insertion is already in the class’s disjoint classes:

\[ C \cap D \subseteq \bot \Rightarrow \]

- when \( \neg C(x) \) if \( D(y) \) then rollback
- when \( \neg D(y) \) if \( C(x) \) then rollback.

As can be seen from the triggers listed above, no reasoner insert is generated (i.e., no implicit information is inferred); therefore, we do not conduct a label & check process when database users attempt to delete data from one of the disjoint classes.

AllValuesFrom: denoted in DL as \( \forall P, D \) defines a set of individuals such that if \( x \) is a member of that set, and \( (x, y) \) is in the property \( P \), then \( y \) is always an individual of the class \( D \). Note that as a consequence of this definition, if \( P \) has no property facts of the form \( (x, y) \), it is still the case that \( \forall P, D \) holds for \( x \).

OWL 2 RL only allows the AllValuesFrom to be a superclass expression, and the production rule which generates logical triggers for handling inserts is shown as follows:

\[ C \subseteq \forall P, D \Rightarrow \]

- when \( \neg C(x) \) if \( P(x, y) \) then \( D(y) \), \( \neg C(x) \) if \( P(x, y) \) then \( D(y) \), \( +P(x, y) \) if \( C(x) \) then \( D(y) \).
This case \( C \subseteq \forall P. D \) (i.e. a class is a subset of an AllValuesFrom expression) means that any individual \( x \) of class \( C \) that appears in an instance \((x, y)\) of \( P \) infers \( y \) as a member of \( D \). This case needs two after triggers: one trigger on inserting \( x \) to \( C \) captures that if the tuple \((x, y)\) already exists in \( P \) then \( y \) is implicitly inserted into \( D \), and another trigger on inserting \((x, y)\) to \( P \) captures that if \( x \) already exists in \( C \) then \( y \) is inserted into \( D \). Note that free variables in conditions, such as the \( y \) which appears in \( P(x, y)_{\text{evi}} \), are universally quantified.

Here, although OWL 2 RL does not allow the AllValuesFrom as a subclass expression, it is still interesting to mention that, based on an open-world assumption, we cannot infer anything for the case of \( \forall P. D \subseteq \exists C. \). The reason for this is that, even if \((x, y)\) holds for \( P \) and \( y \) holds for \( D \), we are unable to infer \( x \) as a member of \( C \), because with the open-world assumption, there might exist another \((x, z)\) of \( P \) that is not yet recorded, where \( z \) does not belong to \( D \).

With regard to deletes, we need the following production rules which generate triggers for labelling:

\[
C \subseteq \forall P. D \rightarrow
\begin{align*}
&\text{when } +{C}(x, y) \text{ if } P(x, y) \text{ then } \lnot D(y); \\
&\text{when } +{P}(x, y) \text{ if } C(x) \text{ then } \lnot D(y).
\end{align*}
\]

The generated triggers guarantee that after labelling either \( C(x) \) or \( P(x, y) \) with the state \( d \), a reasoner delete \( \lnot D(y) \) is generated to cause the attempt to label \( y \) in \( D \). When considering the check phase, we specify the inference rule of the table \( D \) as \( D(y) \rightarrow {C}(x)_{\text{evi}}, P(x)_{\text{evi}} \).

IntersectionOf: in OWL 2 RL \( E_1 \cap \ldots \cap E_n \) is used for expressing a set of individuals which exist in all of \( \{E_1, \ldots, E_n\} \), where \( E_i, 1 \leq j \leq n \) can be any OWL 2 RL allowed expression which denotes a set of individuals (e.g. \( \forall P. D \)). Then, an OWL class \( C \) can be restricted to be either a subset of or equivalent to the IntersectionOf expression (i.e. \( C \subseteq E_1 \cap \ldots \cap E_n \) or \( C \equiv E_1 \cap \ldots \cap E_n \)). For example, Rule (4) defines that the class Father is equivalent to common individuals which exist in both Man and Parent.

In order to demonstrate how triggers and inference rules are generated from axioms using IntersectionOf in a more straightforward way, we assume \( E_j \) as only OWL classes, and note that other types of expressions are handled in a similar way. In addition, because \( C \equiv E_1 \cap \ldots \cap E_n \) contains the two cases \( C \subseteq E_1 \cap \ldots \cap E_n \) and \( E_1 \cap \ldots \cap E_n \subseteq C \), we choose it for a more general demonstration.

A tableaux-based reasoner classifies \( C \equiv E_1 \cap \ldots \cap E_n \) into \( n + 1 \) subsumption relationships as follows (like Rule (4) is classified to Rules (18)–(20)):

\[
\begin{align*}
&n \text{ subsumption axioms} \\
&C \subseteq E_1, \ldots, C \subseteq E_n \\
&E_1 \cap \ldots \cap E_n \subseteq C
\end{align*}
\]

where the first \( n \) subsumption relations are actually axioms using SubClassOf (i.e. \( C \subseteq E_j, 1 \leq j \leq n \)), which result in the following production rule for generating triggers:

\[
C \subseteq E_j \rightarrow \text{when } +{C}(x)_{\text{evi}} \text{ then } E_j(x), \quad \text{when } +{E}(x)_{\text{evi}} \text{ then } \lnot E_j(x), \quad 1 \leq j \leq n.
\]

Triggers translated from the above production rule ensure that a reasoner insert of \( x \) into \( E_j \) is generated after inserting \( x \) into \( C \), and a reasoner delete of \( x \) from \( E_j \) is also conducted after labelling \( x \) in \( C \) with the state \( d \). Moreover, the inference rule of \( E_j \) will include non-labelled \( x \) in \( C \) (i.e. \( E_{\text{del}}(x):=\lnot C(x)_{\text{evi}} \)). For example, Rules (18) and (19) are translated to the below triggers for generating reasoner inserts and deletes:

\[
\begin{align*}
&\text{when } +{\text{Father}}(x)_{\text{evi}} \text{ then } \text{Man}(x), & \& \text{Parent}(x), \\
&\text{when } +{\text{Father}}(x)_{\text{evi}} \text{ then } \lnot \text{Man}(x), & \& \lnot \text{Parent}(x),
\end{align*}
\]

and the inference rules of both tables Man and Parent include non-labelled data in Father as shown as follows:

\[
\begin{align*}
&\text{Man}_{\text{int}}(x):={\text{Father}}(x)_{\text{evi}},  \\
&\text{Parent}_{\text{int}}(x):={\text{Father}}(x)_{\text{evi}}.
\end{align*}
\]

For the last subsumption axiom, we define the following production rule:

\[
E_1 \cap \ldots \cap E_n \subseteq C \rightarrow
\begin{align*}
&\text{when } +{E_1}(x)_{\text{evi}} \text{ if } E_2(x)_{\text{evi}}, \ldots, E_n(x)_{\text{evi}} \text{ then } C(x), \\
&\text{when } +{E_2}(x)_{\text{evi}} \text{ if } E_1(x)_{\text{evi}}, \ldots, E_{n-1}(x)_{\text{evi}} \text{ then } C(x), \\
&\text{when } +{E_n}(x)_{\text{evi}} \text{ if } E_1(x)_{\text{evi}}, \ldots, E_{j-1}(x)_{\text{evi}}, E_{j+1}(x)_{\text{evi}}, \ldots, E_n(x)_{\text{evi}} \text{ then } C(x), & 1 \leq j \leq n, \\
&\text{when } +{E_1}(x)_{\text{evi}} \text{ if } E_2(x), \ldots, E_n(x) \text{ then } \lnot C(x), \\
&\text{when } +{E_2}(x)_{\text{evi}} \text{ if } E_1(x), \ldots, E_{n-1}(x) \text{ then } \lnot C(x), \\
&\text{when } +{E_n}(x)_{\text{evi}} \text{ if } E_1(x), \ldots, E_{j-1}(x), E_{j+1}(x), \ldots, E_n(x) \text{ then } \lnot C(x), & 1 \leq j \leq n.
\end{align*}
\]

With these triggers, after inserting \( x \) into \( E_n \) if the same \( x \) appears in all the other \( E_k \) (where \( k \neq j \)) with state \( e \) or \( i \), a reasoner insert of \( C(x) \) will be conducted. Similarly, after labelling \( x \) with \( d \) in \( E_j \), if the same \( x \) appears on all the other \( E_k \) \( (k \neq j) \), a reasoner delete of \( \lnot C(x) \) will be processed as well, since we have detected an event where we might be loosing support for inferring \( C(x) \). Taking Rule (20) as an example, we create the following triggers:
TABLE 4. Triggers and inference rules generated from property axioms.

| OWL constructs    | DL syntax | Triggers                                         | Inference rules |
|-------------------|-----------|--------------------------------------------------|-----------------|
| InverseProperty   | $P \equiv Q^-$ | when $\neg P(x, y)_{evi}$ then $Q(y, x)$       | $P_{int}(y, x):=P(x, y)_{evi}$ |
|                   |           | when $P(x, y)_{evi}$ then $Q(y, x)$            | $Q_{int}(y, x):=P(x, y)_{evi}$ |
|                   |           | when $Q(x, y)_{evi}$ then $P(y, x)$            | $P_{int}(y, x):=Q(x, y)_{evi}$ |
|                   |           | when $\neg P(x, y)_{evi}$ then $\neg Q(y, x)$ |                |
|                   |           | when $\neg Q(x, y)_{evi}$ then $\neg P(y, x)$ |                |
| SymmetricProperty | $P \equiv P^-$ | when $P(x, y)_{evi}$ then $P(y, x)$            | $P_{int}(y, x):=P(x, y)_{evi}$ |
|                   |           | when $\neg P(x, y)_{evi}$ then $\neg P(y, x)$ |                |
| AsymmetricProperty| $P \sqcap P^c \subseteq \bot$ | when $P(x, y)_{evi}$ if $P(x, y)_{evi}$ then rollback |                |
| DisjointProperties| $P \sqcap Q \subseteq \bot$ | when $P(x, y)_{evi}$ if $Q(x, y)_{evi}$ then rollback |                |
|                   |           | when $Q(x, y)_{evi}$ if $P(x, y)_{evi}$ then rollback |                |
| ImreflexiveProperty| $\top \equiv \neg \exists P. Self$ | when $P(x, y)_{evi}$ then rollback |                |
| NegativePropertyAssertion| $\neg P(x, y)$ | when $P(x, y)_{evi}$ then rollback |                |

Moreover, from the last subsumption axiom, the inference rule of $C$ will include data items common to all of $E_j, 1 \leq j \leq n$ (i.e. $C(x)_{int} := E_1(x)_{evi}, \ldots, E_n(x)_{evi}$), which means that if an individual appears in all classes forming the IntersectionOf expression, this individual will be inferred again as a member of the class $C$. Again, from Rule (20), the inference rule of Father will include common individuals (which are not labelled with $d$) from tables Man and Parent as

$\text{Father}(x)_{int} := \text{Man}(x)_{evi}, \text{Parent}(x)_{evi}$.

Note that deleting an individual $x$ from $C$ in the axiom $C \equiv E_1 \sqcap \ldots \sqcap E_n$ might appear to lead to a view update problem [32], i.e. which one of $E_i$ should $x$ be deleted from? However, in our approach it is clear that in the label process we will update any $x$ in state $i$ to state $d$ in all the classes $E_1 \ldots E_n$, and during the check phase, all such labelled data items will be verified if they are still inferable.

Finally, the construct OneOf is used to define an anonymous class by enumerating its exact individuals $\{a_1 \ldots a_n\}$. Our approach treats an anonymous class by creating an anonymous table which contains exact the individuals defined from an ontology, and afterwards preventing any insertion or deletion.

4.3. The semantics of property axioms

This section demonstrates how logical triggers and inference rules are generated from axioms related to properties. Again, like in Section 4.2, we choose several constructs as examples to demonstrate the production rules, and handling property axioms which use the remaining constructs is summarised in Table 4.

**Domain and Range**: an OWL property may have its domain or range to be explicitly restricted to an OWL class. For example, Rules (5) and (6) restrict that for the property hasParent the values of its Domain and Range should be individuals from classes Person and Parent, respectively. Thus, based on the semantics, we specify the following production rules for translating Domain and Range axioms to triggers and inference rules:

$\top \equiv \forall P. C \rightarrow$ when $\neg \exists P(x, y)_{evi}$ then $C(x)$
\begin{align*}
\text{when }& P(x, y)_{evi} \text{ then } C(x) \\
\text{C}_{int}(x) := & P(x, y)_{evi}
\end{align*}

$\top \equiv \forall P. D \rightarrow$ when $\neg \exists P(x, y)_{evi}$ then $D(y)$
\begin{align*}
\text{when }& P(x, y)_{evi} \text{ then } D(y) \\
\text{D}_{int}(y) := & P(x, y)_{evi}
\end{align*}

Hence, from Rules (5) and (6), we create the trigger shown below on hasParent to generate reasoner inserts to Person and Parent, after making ontology or reasoner inserts into this property table:

$\text{when }\neg \text{hasParent}(x, y)_{evi} \text{ then } \text{Person}(x), \text{Parent}(y)$

Note the rules above state two triggers should be created, but since they share the same when and if sections, they can for efficiency be combined into a single trigger.
Also, the following trigger guarantees that the related individuals in Person and Parent will be labelled, if a property fact in hasParent is labelled:

\[
\text{when } \text{hasParent}(x, y) \text{ then } \neg \text{Person}(x), \neg \text{Parent}(y).
\]

Finally, the inference rules of Person and Parent will include non-labelled domain and range values in hasParent, respectively:

\[
\begin{align*}
\text{Person}_{\text{nl}}(x) :&: \text{hasParent}(x, y)_{\text{ei} vi} \\
\text{Parent}_{\text{nl}}(y) :&: \text{hasParent}(x, y)_{\text{ei} vi}.
\end{align*}
\]

SubPropertyOf: Defining triggers and inference rules for handling SubPropertyOf is similar to what we have introduced for dealing with SubClassOf. The production rule below generates triggers to perform reasoner inserts (or reasoner deletes) over the super property table after inserting data into (or labelling data in) the sub property table. In addition, the inference rule associated with the super property table will include non-labelled data in the sub property table:

\[
P \subseteq Q \implies \text{when } +P(x, y)_{\text{ei} vi} \text{ then } Q(x, y)_{\text{ ei} vi} \\
\text{when } +P(x, y)_{\text{d} vi} \text{ then } \neg Q(x, y)_{\text{d} vi} \\
Q_{\text{nl}}(x, y) :+P(x, y)_{\text{ei} vi}.
\]

TransitiveProperty: In a transitive property (exemplified as Rule (9)), if \((x, y)\) and \((y, z)\) exist, then \((x, z)\) is also an instance of this property. We use the production rule below to generate logical triggers which achieve the semantics of transitivity:

\[
P_{\text{d} vi} P \subseteq P \implies \\
\text{when } +P(x, y)_{\text{ei} vi} \text{ if } P(y, z)_{\text{ei} vi} \text{ then } P(x, z)_{\text{ei} vi} \\
\text{when } +P(x, y)_{\text{d} vi} \text{ if } P(y, z)_{\text{d} vi} \text{ then } P(x, z)_{\text{d} vi} \\
\text{when } +P(x, y)_{\text{d} vi} \text{ if } P(y, z) \text{ then } \neg P(x, z)_{\text{d} vi} \\
\text{when } +P(x, y)_{\text{d} vi} \text{ if } P(y, z) \text{ then } \neg P(x, z)_{\text{d} vi}.
\]

The triggers generated for a transitive property guarantee that (i) after each insertion, the system will check whether there will be transitive tuples inferred or not, and if so, inferred tuples will be inserted implicitly; (ii) a tuple will be labelled, if other data which can infer it because of transitivity is labelled. Moreover, the inference rule of a transitive property will be defined as \(P_{\text{nl}}(x, z) := P(x, y)_{\text{ei} vi}, P(y, z)_{\text{ei} vi}\).

If we consider a family T-Box fragment composed of a SubPropertyOf axiom and a TransitiveProperty axiom (i.e. Rules (7) and (9)), based on the production rules for translating \(P \subseteq Q\) and \(P_{\text{d} vi} P \subseteq P\) into triggers, the two rules are mapped to triggers as follows:

\[
\text{when } +\text{hasParent}(x, y)_{\text{ei} vi} \text{ then } \neg \text{hasAncestor}(x, y)_{\text{ei} vi} \\
\text{when } +\text{hasAncestor}(x, y)_{\text{ei} vi} \text{ if } +\text{hasAncestor}(y, z)_{\text{ei} vi} \text{ then } \neg \text{hasAncestor}(x, z)_{\text{ei} vi} \\
\text{when } +\text{hasParent}(y, z)_{\text{ei} vi} \text{ if } +\text{hasParent}(x, y)_{\text{ei} vi} \text{ then } \neg \text{hasAncestor}(x, z)_{\text{ei} vi}.
\]

Therefore, if we insert the A-Box facts (15) and (16), (Jack, Mike) and (John, Jack) will be explicitly stored in hasParent and hasAncestor, respectively (i.e. hasParent(Jack, Mike)), and hasAncestor(John, Jack). Moreover, (Jack, Mike) will be inferred as a member of hasAncestor, which can be combined with hasAncestor(John, Jack) to infer (John, Mike) as a member of hasAncestor because of the transitivity.

Because there are two inference cases to the table hasAncestor, namely: subsumption and transitivity, the inference rule of hasAncestor contains two parts:

\[
\begin{align*}
\text{hasAncestor}_{\text{nl}}(x, z) :&: \text{hasParent}(x, z)_{\text{ei} vi} \\
\text{hasAncestor}_{\text{nl}}(x, z) :&: \text{hasAncestor}(x, y)_{\text{ei} vi}, \text{hasParent}(y, z)_{\text{ei} vi}.
\end{align*}
\]

PropertyChain: A new feature of OWL 2 RL (compared to OWL 1) is the property chain, which allows for a property to be defined from the concatenation of two or more other properties. For example, we can define that a parent of a parent is a grandparent by Rule (8). Naturally, the following triggers can be produced for handling inserts to hasParent w.r.t Rule (8):

\[
\begin{align*}
\text{when } +\text{hasParent}(x, y)_{\text{ei} vi} \text{ if } +\text{hasParent}(y, z)_{\text{ei} vi} \text{ then } \neg \text{hasGrandparent}(x, z)_{\text{ei} vi} \\
\text{when } +\text{hasParent}(y, z)_{\text{ei} vi} \text{ if } +\text{hasParent}(x, y)_{\text{ei} vi} \text{ then } \neg \text{hasGrandparent}(x, z)_{\text{ei} vi}.
\end{align*}
\]

and the below triggers for the purpose of labelling:

\[
\begin{align*}
\text{when } +\text{hasParent}(x, y)_{\text{d} vi} \text{ if } +\text{hasParent}(y, z) \text{ then } \neg \text{hasGrandparent}(x, z)_{\text{d} vi} \\
\text{when } +\text{hasParent}(y, z)_{\text{d} vi} \text{ if } +\text{hasParent}(x, y) \text{ then } \neg \text{hasGrandparent}(x, z)_{\text{d} vi}.
\end{align*}
\]

and the following inference rule on the table hasGrandparent for the purpose of checking:

\[
\text{hasGrandparent}_{\text{nl}}(x, z) := \text{hasParent}(x, y)_{\text{ei} vi}, \text{hasParent}(y, z)_{\text{ei} vi}.
\]

We also provide the general production rule for handling the situation that a chain property contains \(n\) subchain properties. The production rule which generates triggers for inserts is
we have implemented all the OWL 2 RL Direct Semantics rules as SQL triggers in a database, and the task of reasoning which we consider (i.e. type inference) exactly follows the conditions defined in Theorem PR1. Moreover, as shown later in Section 6, we empirically evaluate the completeness of our system by using the test suites provided by SyGENia [33].

The major incompleteness of our approach comes from separating T-Box and A-Box reasoning. As addressed in Ref. [26], this separation makes certain T-Box subsumptions unable to be derived, because they may require considering both T-Box and A-Box statements. For example, in an ontology containing the T-Box rules \( A \sqsubseteq \exists P \{ b \} \) and \( \exists P. B \sqsubseteq C \), one cannot infer the subsumption relation \( A \sqsubseteq C \), unless an additional A-Box fact \( B(b) \) is added. However, missing this relation still does not affect the completeness of type inference, i.e. our approach is still able to derive \( C(x) \) from \( A(x) \) by invoking the SQL triggers translated from \( A \sqsubseteq \exists P \{ b \} \) and \( \exists P. B \sqsubseteq C \) without needing \( B(b) \). In addition, we focus on the situation that the T-Box is much smaller when comparing to the size of A-Boxes, and this enables the combination of a tableaux-based reasoner and a rule-based reasoning engine (e.g. DLEJena [34] and Minerva). Using a tableaux-based reasoner to classify the T-Box gives a more complete subsumption relationships than simply evaluating the OWL 2 RL/RDF rules provided by Ref. [11], as shown in Ref. [35].

Apart from the incompleteness due to T-Box and A-Box separation, another difference between our system and other OWL reasoners is the use of UNA. The semantics of UNA denotes the case that different individual names always denote different objects, which is semantically equivalent to the use of AllDifferent in OWL. This contrasts the semantics behind the OWL constructor sameAs (or others which may derive sameAs statements, such as FunctionalProperty, InverseFunctionalProperty and HasKey), which allows users to state the scenario that different individual names might refer to the same object. Nevertheless, UNA is commonly applied in rule-based systems, because their logical fundament Datalog follows UNA and the closed-world assumption; whereas our approach extends this with the open-world assumption.
exposing such an extra column to users will expose the details of the reasoning implementation to user applications.

To avoid this, we create for each ontology two schemas, as outlined in Fig. 7: one is a back-end schema $\mathcal{SB}$ which consists of materialised tables storing both data items and their states, and the second is a front-end schema $\mathcal{SF}$ which contains SQL views (i.e. virtual tables) showing only the data items stored in $\mathcal{SB}$. The views and tables are created by using a set of SQL CREATE VIEW and CREATE TABLE statements, based on the following mappings from OWL classes and properties to Datalog rules:

$$C \rightarrow \mathcal{SF}.C(id):=\mathcal{SF}.C(id, st)$$

$$P \rightarrow \mathcal{SF}.P(domain, range):=\mathcal{SB}.P(domain, range, st)$$

Schema $\mathcal{SF}$ is used for accepting any inserts and deletes. Then, these database operations are transferred to $\mathcal{SB}$, where the type inference is processed.

5.2. Logical triggers to physical triggers

SQL BEFORE triggers (PL/pgSQL [37]) or INSTEAD OF triggers (Transact SQL [38]) are used to implement $-$ events, and SQL AFTER triggers are used for $+$ events.

Considering first inserts, ontology inserts executed over views in $\mathcal{SF}$ will be automatically translated to corresponding tables in $\mathcal{SB}$. For example, two ontology inserts over the view $\mathcal{SF}.\text{Man}$:

```
INSERT INTO $\mathcal{SF}.\text{Man}$(id)
VALUES ('v1'), ('v2')
```

will be translated as:

```
INSERT INTO $\mathcal{SB}.\text{Man}$(id, st)
VALUES ('v1',null), ('v2',null)
```

into $\mathcal{SB}.\text{Man}$. The inserts executed over $\mathcal{SB}.\text{Man}$ are then captured by the BeforeInsertSBManTrigger (listed in Fig. 8) created on this table. The BeforeInsertSBManTrigger contains three parts. First, if the data is already present in the explicit state, a rollback is performed to prevent repeated ontology inserts. Next, if the data is already present with the implicit state, this state is updated to the explicit state. Finally, if the data is simply not present, a simple insert of the new data is performed. Supposing, $v_1$ and $v_2$ are not in the database, the SQL statement that the trigger actually performs is:

```
INSERT INTO $\mathcal{SB}.\text{Man}$(id, st)
VALUES ('v1','e'), ('v2','e')
```

which stores the two data items with the explicit state.

Another important feature in our implementation is that all triggers are implemented with table-level semantics rather than row-level semantics to improve the performance of type
inference. For example, as shown in Fig. 9, in the physical trigger AfterInsertSBManTrigger created on \( S_B . \) Man for when \( \text{`Man}(x)_e \text{v1} \text{ then Person}(x)_i \text{v1} \), when a transaction inserts multiple data items to \( S_B . \) Man, all inserted items will be inserted to \( S_B . \) Person as one reasoner insert rather than inserting each of them separately. Thus, after persisting \( v_1 \) and \( v_2 \) with state \( e \) in \( S_B . \) Man, the AfterInsertSBManTrigger will generate and execute another SQL statement:

\[
\text{INSERT INTO } S_B . \text{Person}\left(id, st\right) \\
\text{VALUES}\left('v1', 'i', 'v2', 'i'\right)
\]

over \( S_B . \) Person, which stores \( v_1 \) and \( v_2 \) in \( S_B . \) Person with the implicit state.

When implementing triggers for deletes, the labelling process must be guaranteed to finish before starting the checking process. This is achieved by creating ‘before delete’ triggers (e.g. BeforeDeleteSFManTrigger in Fig. 10) on views in \( S_F . \) Man to control the labelling and checking which are actually processed in \( S_B . \)

Each of these triggers essentially contains three parts: the first part is used to rollback inconsistent ontology deletes (i.e. an ontology delete which attempts to delete a data item with implicit state). The second part is designed to start the labelling process, and only when the whole labelling is finished (i.e. all other data which has a derivation from the data of the ontology deletes is labelled with \( d \)), can the third part start the checking phase. On each table in \( S_F . \), we create SQL UPDATE triggers for labelling, and SQL DELETE triggers for checking.

Figure 10 gives the BeforeDeleteSFManTrigger created on \( S_F . \) Man. Besides the first part which is used to rollback inconsistent ontology deletes, part 2 of the trigger is designed for starting the labelling process by updating data items

\[
\text{CREATE TRIGGER BeforeDeleteSFManTrigger} \\
\text{ON SF.Man INSTEAD OF DELETE AS BEGIN} \\
\quad \text{IF EXISTS} \\
\quad \quad (\text{SELECT } *) \\
\quad \quad \text{FROM deleted} \\
\quad \quad \text{JOIN } S_B . \text{Man} \\
\quad \quad \text{ON deleted.id}=S_B . \text{Man.id} \\
\quad \quad \text{AND } S_B . \text{Man.st}='i'\) \\
\quad \quad \text{RAISERROR}('ERROR_MESSAGE', 10, 1) \\
\quad \quad \text{ROLLBACK TRANSACTION} \\
\quad \text{return} \\
\quad \text{-- part2: start labelling process} \\
\quad \text{UPDATE } S_B . \text{Man} \\
\quad \text{SET } st='d' \\
\quad \text{WHERE id IN (SELECT id FROM deleted)} \\
\quad \quad \text{AND } st='e' \\
\quad \text{-- part3: start checking process} \\
\quad \text{DELETE FROM } S_B . \text{Man} \\
\quad \text{WHERE st='d'} \\
\text{END}
\]

**FIGURE 10.** Before delete trigger on \( S_F . \) Man.

which might be deleted in \( S_B . \) Man to \( d \). Thus, if we take a consistent ontology delete:

\[
\text{DELETE FROM SF.Man} \\
\text{WHERE id = 'v1'}
\]
as an example, this ontology delete will be actually changed as an SQL UPDATE statement:

\[
\text{UPDATE SF.Man} \\
\text{SET } st='d' \\
\text{WHERE id = 'v1'} \\
\text{AND } st='e'
\]

This state update is then captured by the after update trigger created on \( S_B . \) Man (i.e. AfterUpdateSBManTrigger shown in Fig. 11) to continue the labelling process, which will further update the state of the same data in \( S_B . \) Person to \( d \), the SQL statement for this update is:

\[
\text{UPDATE SF.Man} \\
\text{SET } st='d' \\
\text{WHERE id = 'v1'} \\
\text{AND } st='i'
\]

Only when the labelling has finished, can part 3 of the BeforeDeleteSFManTrigger start the check process by conducting an attempt to delete labelled data in \( S_B . \) Man:

\[
\text{DELETE} \\
\text{FROM SF.Man} \\
\text{WHERE st='d'}
\]

\[
\text{CREATE TRIGGER AfterUpdateSBManTrigger} \\
\text{ON SB.Man AFTER UPDATE AS} \\
\quad \text{-- labelling cascades to SB.Person} \\
\quad \text{UPDATE SB.Person} \\
\quad \text{SET } st='d' \\
\quad \text{WHERE } st='i' \\
\quad \text{AND } id IN (SELECT id FROM inserted) \\
\quad \quad \text{WHERE st='d'} \\
\quad \text{END}
\]

**FIGURE 11.** After update trigger on \( S_B . \) Man.

\[
\text{CREATE TRIGGER BeforeDeleteSBManTrigger} \\
\text{ON SB.Man INSTEAD OF DELETE AS BEGIN} \\
\quad \text{-- part1: keep inferable data} \\
\quad \text{-- no action required} \\
\quad \text{-- part2: remove non-inferable data} \\
\quad \text{DELETE FROM S_B.Man} \\
\quad \text{WHERE } st='d'} \\
\text{END}
\]

**FIGURE 12.** Before delete trigger on \( S_B . \) Man.
which is then detected by the BeforeDeleteSBManTrigger on SB.Man in Fig. 12 to change the state of still inferable data items to i and to delete non-inferable data. However, since the inference rule created on SB.Man is empty, part 2 of BeforeDeleteSBManTrigger will simply delete v1 (which is with the state d).

The checking phase also cascades as the order of labelling process. In this particular example, an after delete trigger AfterDeleteSBManTrigger on SB.Man (shown in Fig. 13) should be designed to further cause the check of labelled data in SB.Person by conducting an SQL DELETE statement over SB.Person, i.e.:

DELETE
FROM SB.Person
WHERE st = ‘d’

This delete will fire a before delete trigger BeforeDeleteSBSPersonTrigger (which is similar to BeforeDeleteSBManTrigger and is shown in Fig. 14) on SB.Person to perform the check (i.e. check if there exists still inferable data to SB.Person from non-labelled data in SB.Man and SB.Human). Until the total checking phase is finished, part 3 of BeforeDeleteSFManTrigger can terminate, which means that the process of handling the ontology delete DELETE FROM SF.Man WHERE id = ‘v1’ ends.

5.3. Optimisation

The only optimisation in our system at present is that an index on the column (id) is created for each class table, and two indexes (domain, range) and (range, domain) are created for each property table. These indices improve the query processing performance, both in reasoning, and in user applications.

6. EVALUATION

This section compares SQOWL2 to Stardog and OWLim. Stardog is a non-materialising reasoner, while OWLim is a materialising reasoner. They both store their data outside of an RDBMS, and thus do not provide transactional reasoning.

For the comparison of speed1 of incremental type inference and query processing, we used the well-known Lehigh University Ontology Benchmark (LUBM) [39], which covers a university domain. It provides a T-Box of 43 OWL classes, 32 OWL properties, and approximately 200 axioms.

TABLE 6. Data loading speed (inserts/s).

|            | LUBM-25 | LUBM-50 | LUBM-100 | LUBM-200 |
|------------|---------|---------|----------|----------|
| SQOWL2     | 5684    | 5966    | 6176     | 5978     |
| OWLim      | 42 067  | 41 623  | 39 946   | 35 933   |
| Stardog    | 242 323 | 262 752 | 271 332  | –        |

Note: 1All experiments were processed on a machine with Intel i7–2600 CPU @ 3.40 GHz, 8 Cores, and 16 GB of memory, running Microsoft SQL Server 2014. SQOWL2 used OWL API v3.4.3 for ontology loading and Pellet v2.3.1 for classification. For comparisons, we used OWLim-Lite v5.4.6486 and Stardog-Community v2.2.1.

TABLE 5. Data loading time (s).

|            | LUBM-25 | LUBM-50 | LUBM-100 | LUBM-200 |
|------------|---------|---------|----------|----------|
| SQOWL2     | 583     | 1115    | 2133     | 4465     |
| OWLim      | 79      | 160     | 336      | 743      |
| Stardog    | 14      | 26      | 51       | –        |

FIGURE 13. After delete trigger on SB.Man.

FIGURE 14. Before delete trigger on SB.Person.
answers to this query w.r.t. the T-Box. If a reasoning system successfully passes the test suite, then it is complete to answer this query w.r.t. the T-Box and any arbitrary A-Box of data.

6.1. Performance of incremental type inference

Data loading: We loaded into each system LUBM data sets with four increasing A-Box sizes (i.e. LUBM-25, LUBM-50, LUBM-100 and LUBM-200). Table 5 shows the loading time needed by each system, and Table 6 demonstrates the speed of data loading by them. All systems showed scalable data loading, as each system's data loading time grew linearly from LUBM-25 to LUBM-200. The speed of loading different data sets by each system was stable (e.g. Data loading speed of SQOWL2 was around 6000 inserts/s). Stardog was the fastest, as it does not perform reasoning during data loading. SQOWL2 was the slowest, because it performs type inference as part of database transactions with full ACID properties, and materialises the result of reasoning during inserts. Indeed, due to the overheads associated with providing ACID properties for database updates, even without reasoning (i.e. without SQOWL2 triggers), the speed of data loading for LUBM in the SQL Server database used for testing was only around 14 600 inserts/s. Thus, the process of reasoning caused a significant, but not impractical, overhead to normal RDBMS database operations.

Data deleting: After data loading, we used each system to execute a number of random deletes of A-Box data. Table 7 only shows the average speed of handling deletes by SQOWL2 and Stardog, as the Lite version of OWLim performs the whole reasoning again after deleting any facts. Indeed, as we tested, even over LUBM-25 executing a delete of fact by OWLim required about 15 s for not only removing this fact but also recomputing the inference materialisation. However, as shown in Table 7, SQOWL2 on average performed 305 deletes/s over LUBM-25 (i.e. just approximately 3 ms required for a deletion). As Stardog does not consider reasoning when inserting or deleting facts, its speed of data deleting was stable over different data sets, and was much faster than SQOWL2. As expected, the label & check process meant that the speed of handling deletes by SQOWL2 decreased when processing deletes from larger data sets, except from LUBM-25 to LUBM-50. The reason for this improvement was the RDBMS switching to a more efficient

| Table 7. Data deleting speed (deletes/s). |
|------------------------------------------|
| LUBM-25 | LUBM-50 | LUBM-100 | LUBM-200 |
| SQOWL2  | 305     | 581      | 420      | 224      |
| Stardog | 29 268  | 28 759   | 28 903   | –        |

6.2. Performance of query processing

Completeness: We used SQOWL2 to process the 14 LUBM queries over LUBM-1, and our experiment shows that SQOWL2 generated exactly the same answers as Pellet; therefore, SQOWL2 is sound and complete for processing the 14 queries over LUBM-1. However, since the LUBM benchmark data is not very exhaustive [40], we further used our system to process the test suites generated by SyGENiA for the LUBM T-Box and the 14 queries. SQOWL2 passed all test suites, and therefore is complete for the 14 queries w.r.t the T-Box and any arbitrary LUBM A-Boxes. In Ref. [33], the most complete system out of four tested was found to be OWLim. Our tests show even now OWLim provides incomplete answers for Q6, Q8 and Q10 with completeness 0.96, 0.93 and 0.96, respectively (the completeness level of processing each test suite is calculated as dividing the number of passed inference cases in this test suite by the number of all inference cases contained in this test suite). For example, Q6 is incomplete, since OWLim does not totally handle reasoning which includes existential quantification.

Efficiency: We used each of the three systems to simulate a load over its stored knowledge data by cycling through the 14 LUBM queries, and recorded the time used for executing each of them. Table 8 shows the average speed when processing queries (i.e. how many queries on average can be processed per minute) by each of the three systems. The speed of query processing by SQOWL2 or OWLim was much faster than Stardog, because the two systems store explicit and implicit data at the data loading stage. For example, the average speed for executing queries over LUBM-100 by SQOWL2 was about 100 times as fast as Stardog. When only comparing the two materialising systems, SQOWL2’s average speed was significantly faster than OWLim for LUBM-25, LUBM-50 and LUBM-100, and was comparable to OWLim for LUBM-200. The average speed of query processing by SQOWL2 dropped sharply from LUBM-100 to LUBM-200 due mostly to Q2 (which needed 481 ms over LUBM-100 but about 21 s over LUBM-200). The query plans used by the RDBMS show that it chose Nested Loops for joining tables when processing Q2.
TABLE 9. Detailed query processing time (ms).

| LUBM-n     | System    | Q1  | Q2  | Q3  | Q4  | Q5  | Q6  | Q7  | Q8  | Q9  | Q10 | Q11 | Q12 | Q13 | Q14 |
|------------|-----------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| LUBM-25    | SQOWL2    | 0.0 | 248 | 0.0 | 2.0 | 7.2 | 31  | 1.0 | 70  | 277 | 0.0 | 3.4 | 12  | 1.0 | 17  |
|            | OWLim     | 168 | 202 | 1.2 | 5.4 | 7.8 | 97  | 2.2 | 404 | 922 | 1.2 | 1.4 | 28  | 2.2 | 39  |
|            | Stardog   | 57  | 259 | 52  | 85  | 57  | 1972| 279 | 681 | 243 | 394 | 56  | 77  | 56  | 53  |
| LUBM-50    | SQOWL2    | 0.0 | 196 | 0.0 | 2.2 | 7.0 | 25  | 0.8 | 116 | 489 | 0.0 | 7.0 | 4.0 | 2.0 | 36  |
|            | OWLim     | 278 | 512 | 1.6 | 5.6 | 7.8 | 164 | 2.0 | 698 | 1867| 1.0 | 1.4 | 43  | 1.8 | 80  |
|            | Stardog   | 56  | 492 | 53  | 91  | 59  | 46084| 506 | 1213| 445 | 701 | 55  | 85  | 56  | 52  |
| LUBM-100   | SQOWL2    | 0.0 | 481 | 0.0 | 2.0 | 7.2 | 38  | 1.0 | 112 | 931 | 0.0 | 14  | 4.2 | 5.4 | 22  |
|            | OWLim     | 476 | 1969| 1.0 | 5.4 | 8.0 | 290 | 2.6 | 1368| 3831| 1.2 | 1.8 | 86  | 2.4 | 155 |
|            | Stardog   | 57  | 830 | 55  | 90  | 58  | 157022| 879 | 2178| 831 | 1106| 57  | 89  | 57  | 52  |
| LUBM-200   | SQOWL2    | 0.0 | 21040| 0.0| 2.2 | 8.0 | 57  | 1.2 | 133 | 1855| 0.0 | 2.0 | 5.2 | 12  | 44  |
|            | OWLim     | 900 | 7872| 1.6 | 6.2 | 8.2 | 472 | 2.2 | 2623| 7639| 1.6 | 1.2 | 178 | 4.2 | 315 |

TABLE 10. Query improvements (ms) by SQOWL2 after database tuning (results in Table 9 → improved performance).

| System    | Q3  | Q4  | Q5  | Q6  | Q7  | Q8  | Q9  | Q10 | Q11 | Q12 | Q13 | Q14 |
|-----------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| LUBM-25   | 248 | 123 | 7.2 | 0.0 | 0.0 | 1.0 | 277 | 241 | 0.0 | 0.0 | 1.0 | 0.0 |
| LUBM-50   | 248 | 123 | 7.2 | 0.0 | 0.0 | 1.0 | 277 | 241 | 0.0 | 0.0 | 1.0 | 0.0 |
| LUBM-100  | 248 | 123 | 7.2 | 0.0 | 0.0 | 1.0 | 277 | 241 | 0.0 | 0.0 | 1.0 | 0.0 |
| LUBM-200  | 248 | 123 | 7.2 | 0.0 | 0.0 | 1.0 | 277 | 241 | 0.0 | 0.0 | 1.0 | 0.0 |

over LUBM-200, which is less efficient than the Hash Match used over LUBM-100. We intentionally did not tune the database to solve this problem, but note that as in any RDBMS application, larger data sets may require certain queries to be manually tuned by a database administrator.

Table 9 gives more detailed results, listing the average query processing time per query for each of the three systems studied. Over all LUBM data sets, SQOWL2 was faster than OWLim in most cases. The only query which SQOWL2 was always slower than OWLim is Q11. This query’s processing time by SQOWL2 increased from LUBM-25 to LUBM-100, but was significantly decreased to just 2 ms over LUBM-200 (which is very close to 1.2 ms used by OWLim). This is because, over LUBM-200, the RDBMS behind SQOWL2 chose a more efficient Nested Loops plan for the joins rather than the Merge Join used over LUBM-25 to LUBM-100. Another interesting query is Q2, as SQOWL2 was faster than OWLim over LUBM-50 and LUBM-100, but was slightly slower over LUBM-25 and was much slower over LUBM-200. SQOWL2 processed Q2 over LUBM-50 even faster than over LUBM-25, because it started to use a more efficient Parallelism plan from LUBM-50 to LUBM-200, but not over LUBM-25. Again, it chose Nested Loops for joining tables when processing Q2 over LUBM-200, which is less efficient than the Hash Match used over LUBM-100. Another query which SQOWL2 did not dominate the performance is Q13, as SQOWL2 only outperformed OWLim over LUBM-25, but was slower over the other three data sets. The query plans for Q13 did not change when the size of data sets grew, and the Q13’s processing time by SQOWL2 did increase linearly, though the time was slightly slower than OWLim.

SQOWL2 was much faster than Stardog when processing all LUBM queries over LUBM-25, LUBM-50 and LUBM-100, except Q9, which was just slightly slower than Stardog. The reason for this is that Q9 only contains one inference case (in the test suite generated by SyGENiA for this query), which only needs to join three tables in order to generate answers to this query; however, SQOWL2 processed the original Q9, which requires five joins among six tables. Stardog was significantly slower when answering Q6 and Q10 than both SQOWL2 and OWLim, since these two queries can be rewritten to numerous sub-queries, which are very complex to compute the answers (169 and 168 inference cases are contained in the test suites for Q6 and Q10, respectively).

Although here we did not provide a systematic way to tuning the database which stores the materialisation, certain manual optimisations, such as adding SQL foreign keys (FKs) and forcing RDBMS to use more efficient query plans, did significantly improve the query processing performance. We provide the improved performance of SQOWL2 after manually tuning the RDBMS in Table 10 (where ‘-’ represents that there was not a significant improvement).

We shall start to introduce our tuning strategies by considering Q9, the only case where SQOWL2 was slower than Stardog. The query optimisation we did was to add SQL FKs (shown below) to represent the semantics of Domain and Range
of the property teacherOf, and the RDBMS later adapted to use a better query plan, especially for joining the tables teacherOf, Faculty and Course. Note that these foreign keys are otherwise unnecessary, since our triggers ensure that the constraint implied by the foreign key always holds in the database:

\[
S_B, \text{teacherOf}(\text{domain}) \rightarrow S_B, \text{Faculty}(\text{id}),
\]

\[
S_B, \text{teacherOf}(\text{range}) \rightarrow S_B, \text{Course}(\text{id}).
\]

We have averaged the processing time of this query after adding these two FKs to SQOWL2 (i.e., 241 ms over LUBM-25, 431 ms over LUBM-50 and 811 ms over LUBM-100); and comparing the new timings with Stardog shows that with foreign keys SQOWL2 was slightly faster than Stardog over all data sets.

Besides Domain and Range, both SubClassOf and SubPropertyOf could also cause new FKs to be added. After adding new FKs, Q5, Q7 and Q13 required much less time by SQOWL2 than without these FKs; most of their processing times were reduced to 0 ms (e.g. Q13 over LUBM-200 was reduced to 0 ms, in which case SQOWL2 became faster than OWLIm).

With regard to Q2, one FK representing the Range of undergraduateDegreeFrom shown as follows can be added:

\[
S_B, \text{undergraduateDegreeFrom}(\text{range}) \rightarrow S_B, \text{University}(\text{id}).
\]

This additional FK caused the RDBMS to use the more efficient Hash Match plan when we executed Q2 over LUBM-25 by using SQOWL2. However, adding this FK surprisingly slowed the processing time for LUBM-50 and LUBM-100, as the RDBMS adapted itself ‘wrongly’ to use the less efficient Nested Loops plan. This kind of ‘wrong’ self-education in an RDBMS is not abnormal, and after forcing the query to be executed by using Hash Match, we re-obtained the quicker performance. Moreover, we investigated the time required over LUBM-200 by forcing the RDBMS to use the Hash Match plan, and the result was as expected to be improved to approximately 1.7 s, which was much faster than the 21 s used before and even much faster than the time 7.871 s required by OWLIm.

As noted above, when considering data updates, our triggers created for SubClassOf, SubPropertyOf, and Domain and Range actually enforce the foreign keys to be satisfied; therefore, at the stage of data uploading and deleting, we can temporarily disable the FK checks (or make them deferable in Postgres).

7. RELATED WORK

Type inference systems over large scale data can be classified as dynamic (i.e. query-rewriting) and materialised approaches, as shown in Fig. 15. Systems using the former approach (e.g. DLDB [3], Stardog [2], Ontop [1] and Lutz et al. [41]) store only explicit facts and conduct reasoning only when there is a query executed over the ontology, where no incremental type inference is required. DLDB, Ontop and Lutz et al. can be considered as transactional reasoning systems, as they use temporal views inside an RDBMS as a method to rewrite executed queries. Take DLDB as an example and consider Rules (1)–(3), DLDB translates these rules into the following SQL views:

\[
\text{Person}(x) \rightarrow \text{Person}(x) \rightarrow \text{Person}(x) \rightarrow \text{Man}(x),
\]

\[
\text{Person}(x) \rightarrow \text{Person}(x) \rightarrow \text{Person}(x) \rightarrow \text{Human}(x),
\]

\[
\text{Human}(x) \rightarrow \text{Human}(x) \rightarrow \text{Human}(x) \rightarrow \text{Parent}(x),
\]

\[
\text{Human}(x) \rightarrow \text{Human}(x) \rightarrow \text{Human}(x) \rightarrow \text{Person}(x).
\]

Note that in query-rewriting approaches, there might exist Datalog mutual recursions (e.g. the above view definitions of Person and Human recursively include the other).

DLDB and Ontop also choose an RDBMS as their data storage mechanism, while Stardog is not an RDBMS-based system for searching, querying and reasoning knowledge data.

Reasoning systems based on materialised approaches store both explicit and implicit data, in order to provide a fast query processing service [42]. Most systems tend to perform reasoning outside an RDBMS (i.e. not proper transactional reasoning), even though they still choose an RDBMS as a possible data container. WebPIE [5], as a sample inference engine, applies the MapReduce model and builds the reasoning mechanism for RDF semantics [43] and OWL ter Horst.

FIGURE 15. Categories of type inference systems.
semantics [24] on top of a Hadoop cluster [44]. WebPIE only supports incremental data loading, but not incremental deleting. OWLIM [4] is another triple-store system, which uses a file system instead of an RDBMS as a container for storing semantic data. Its standard and enterprise versions support incremental data loading and deleting, but not in a transactional manner. RDFox [6] adopts a so called Backward/Forward algorithm (can be more efficient than DRed in certain cases) to achieve incremental reasoning without using an RDBMS, and it is, therefore, not a transactional reasoning system. DynamiTE [45] is another system handling both incremental inserts and deletes w.r.t. the minimal RDFS reasoning [46], by using proposed variants of the Counting and DRed algorithms in Ref. [13]. Again, it does not adopt an RDBMS as its mechanism of reasoning computation and closure storage. Minerva [8] only uses an RDBMS to hold the materialised results generated by an extra reasoner outside the RDBMS, so we do not consider it as a transactional or incremental reasoning system. Oracle’s RDF store [7], by contrast, loads the explicit data in advance, and then uses reasoning rules to generate an inference closure of the loaded data. Although the reasoning is inside the database, it is not performed in an incremental manner. Similar to Oracle’s RDF store, Orel [47] also uses a relational database to compute and store a reasoning closure, and their inference rules address the EL and the RL profiles of OWL 2.

8. CONCLUSION

We have demonstrated an approach using SQL triggers which extends an RDBMS to have type inference capabilities. SQOWL2 is the first approach to provide transactional and incremental type inference from both inserts and deletes of A-Box data. A further advantage of holding data in an RDBMS is that it allows ontology reasoning to be integrated into mainstream data processing, with transactional semantics extended to the reasoning. The evaluation shows that SQOWL2, compared to two fast reasoners, is faster at query processing, and the completeness of query answering is comparable to or better than the same task for other rule-based engines. Of course, the approach is not suited to applications where the inferred data is very large, and queries are relatively infrequent compared to updates.

As our approach is built as a separate layer over the RDBMS, our work has not yet addressed the key issue of optimising the efficiency of handling updates to be as fast as the other engines which are specially designed for triple store (e.g. OWLIM and RDFox), which must be the subject of future work.

REFERENCES

[1] Bagosi, T., Calvanese, D., Hardi, J., Komla-Ebri, S., Lanti, D., Rezk, M., Rodriguez-Muro, M., Slusnys, M. and Xiao, G. (2014) The Ontop Framework for Ontology Based Data Access. In Proc. CSWS 2014, Wuhan, China, August 8–12, pp. 67–77. Springer-Verlag, Berlin Heidelberg.
[2] Pérez-Urbina, H., Rodríguez-Díaz, E., Grove, M., Konstantinidis, G. and Sirin, E. (2012) Evaluation of Query Rewriting Approaches for OWL 2. In Proc. SSWS+HPCSW, Boston, USA, November 11, pp. 32–44. CEUR-WS.org.
[3] Pan, Z., Zhang, X. and Heffin, J. (2008) DDBL2: A Scalable Multi-perspective Semantic Web Repository. In Proc. WI-IAT’08, Sydney, NSW, Australia, December 9–12, pp. 489–495. IEEE, New York.
[4] Giryakov, A., Ognyanov, D. and Manov, D. (2005) OWLIM–A Pragmatic Semantic Repository for OWL. In Proc. WISE Workshops 2005, New York, NY, November 20–22, pp. 182–192. Springer-Verlag, Berlin Heidelberg.
[5] Urbani, J., Kotoulas, S., Maassen, J., Van Harmelen, F. and Bal, H. (2012) WebPIE: A Web-scale Parallel Inference Engine using MapReduce. J. Web Sem., 10, 59–75.
[6] Motík, B., Renov, Y., Piró, R., Horrocks, I. and Olteanu, D. (2014) Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems. In Proc. AAAI 2014, Québec City, Québec, Canada, July 27–31, pp. 129–137. AAAI Press, Palo Alto, California.
[7] Wu, Z., Eadon, G., Das, S., Chong, E.L., Kolovski, V., Annamalai, M. and Srivivasan, J. (2008) Implementing an Inference Engine for RDFS/OWL Constructs and User-Defined Rules in Oracle. In Proc. ICDE 2008, Cancún, México, April 7–12, pp. 1239–1248. IEEE, New York.
[8] Zhou, J., Ma, L., Liu, Q., Zhang, L., Yu, Y. and Pan, Y. (2006) Minerva: A Scalable OWL Ontology Storage and Inference System. In Proc. ASWC 2006, Beijing, China, September 3–7, pp. 429–443. Springer-Verlag, Berlin Heidelberg.
[9] Liu, Y. and McBrien, P. (2013) SQOWL2: Transactional Type Inference for OWL 2 DL in an RDBMS. Informal Proc. DL 2013, Ulm, Germany, July 23–26, pp. 779–790. CEUR-WS.org.
[10] Härder, T. and Reuter, A. (1983) Principles of Transaction-Oriented Database Recovery. ACM Comput. Surv., 15, 287–317.
[11] Motík, B., Grau, B.C., Horrocks, I., Wu, Z., Fokoue, A. and Lutz, C. (eds.) (2012) OWL 2 Web Ontology Language: Profiles (Second Edition). W3C Recommendation. Latest version available at http://www.w3.org/TR/owl2-profiles/.
[12] McGuinness, D.L. and Van Harmelen, F. (eds.) (2004) OWL Web Ontology Language Overview. W3C Recommendation. Latest version available at https://www.w3.org/TR/owl-features/.
[13] Gupta, A., Mumick, I.S. and Subrahmanian, V.S. (1993) Maintaining Views Incrementally. In Proc. ACM SIGMOD 93, Washington, D.C., USA, May 26–28, pp. 157–166. ACM Press, New York, USA.
[14] Motík, B., Renov, Y., Piró, R. and Horrocks, I. (2015) Incremental Update of Datalog Materialisation: the Backward/Forward Algorithm. In Proc. AAAI 2015, Austin, Texas, USA, January 25–30, pp. 1560–1568. AAAI Press, Palo Alto, California.
[15] McBrien, P., Rizopoulos, N. and Smith, A.C. (2010) SQOWL: Type Inference in an RDBMS. In Proc. ER 2010, Vancouver, BC, Canada, November 1–4, pp. 362–376. Springer-Verlag, Berlin Heidelberg.

[16] Liu, Y. and McBrien, P. (2015) Transactional and Incremental Type Inference from Data Updates. In Proc. BICOD 2015, Edinburgh, UK, July 6–8, pp. 206–219. Springer–Verlag, Berlin Heidelberg.

[17] Sowa, J.F. (2000) Knowledge Representation: Logical, Philosophical, and Computational Foundations. Brooks Cole Publishing Co.

[18] Grau, B.C., Horrocks, I., Motik, B., Parsia, B., Patel-Schneider, P. and Sattler, U. (2008) OWL 2: The next step for OWL. J. Web Sem., 6, 309–322.

[19] Krötzsch, M. (2012) OWL 2 Profiles: An Introduction to Lightweight Ontology Languages. In Proc. Reasoning Web – Semantic Technologies for Advanced Query Answering – 8th Int. Summer School 2012, Vienna, Austria, September 3–8, Springer-Verlag, Berlin Heidelberg.

[20] Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D. and Patel-Schneider, P.F. (2010) The Description Logic Handbook: Theory, Implementation, and Applications (Second Edition). Cambridge University Press.

[21] Motik, B., Patel-Schneider, P.F. and Grau, B.C. (eds.) (2012) OWL 2 Web Ontology Language: Direct Semantics (Second Edition). W3C Recommendation. Latest version available at http://www.w3.org/TR/owl2-direct-semantics/

[22] Schneider, M. (ed.) (2012) OWL 2 Web Ontology Language: RDF-Based Semantics (Second Edition). W3C Recommendation. Latest version available at http://www.w3.org/TR/owl2-rdf-based-semantics/

[23] Grosof, B.N., Horrocks, I., Volz, R. and Decker, S. (2003) Description Logic Programs: Combining Logic Programs with Description Logic. In Proc. WWW 2003, Budapest, Hungary, May 20–24, pp. 48–57. ACM Press, New York, USA.

[24] ter Horst, H.J. (2005) Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. J. Web Sem., 3, 79–115.

[25] Nicolas, J. and Yazdanian, K. (1983) An Outline of BDGEN: A Deductive DBMS. In Proc. IFIP Congress, Paris, France, September 19–23, pp. 711–717. Springer–Verlag, Berlin Heidelberg.

[26] Krötzsch, M. (2012) The Not-So-Easy Task of Computing Class Subsumptions in OWL RL. In Proc. ISWC 2012, Boston, MA, November 11–15, pp. 279–294. Springer-Verlag, Berlin Heidelberg.

[27] Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A. and Katz, Y. (2007) Pellet: A practical OWL-DL reasoner. J. Web Sem., 5, 51–53.

[28] Tsarkov, D. and Horrocks, I. (2006) FaCT++ Description Logic Reasoner: System Description. In Proc. IJCAR 2006, Seattle, WA, August 17–20, pp. 292–297. Springer-Verlag, Berlin Heidelberg.

[29] Glimm, B., Horrocks, I., Motik, B., Stoilos, G. and Wang, Z. (2014) HermiT: An OWL 2 Reasoner. J. Autom. Reason., 53, 245–269.

[30] Donnelly, K. (2006) SNOMED-CT: The advanced terminology and coding system for eHealth. Studies in Health Technology and Informatics, 121, 279.

[31] Delaitre, V. and Kazakov, Y. (2009) Classifying ELH Ontologies In SQL Databases. In Proc. OWLED 2009, Chantilly, VA, October 23–24. CEUR-WS.org.

[32] Date, C.J. (2000) An Introduction to Database Systems (Seventh edition). Addison-Wesley-Longman.

[33] Stoilos, G., Grau, B.C. and Horrocks, I. (2010) How Incomplete is Your Semantic Web Reasoner? In Proc. AAAI 2010, Atlanta, Georgia, USA, July 11–15. AAAI Press, Palo Alto, California.

[34] Medtiskos, G. and Bassiliades, N. (2010) DLEJena: A practical forward-choaining OWL 2 RL reasoner combining Jena and Pellet. J. Web Sem., 8, 89–94.

[35] Medtiskos, G. and Bassiliades, N. (2008) Combining a DL Reasoner and a Rule Engine for Improving Entailment-Based OWL Reasoning. In Proc. ISWC 2008, Karlsruhe, Germany, October 26–30, pp. 277–292. Springer-Verlag, Berlin Heidelberg.

[36] EAP: A Java API for OWL ontologies. In Semantic Web, 2, 11–21.

[37] Momjian, B. (2001) PostgreSQL: Introduction and Concepts. Addison-Wesley.

[38] Kline, K., Gould, L. and Zanevsky, A. (1999) Transaction-SQL Programming: Covers Microsoft SQL Server 6.5/7.0 and Sybase Adaptive Server 11.5. O’Reilly Media.

[39] Guo, Y., Pan, Z. and Heffin, J. (2005) LUBM: A benchmark for OWL knowledge base systems. J. Web Sem., 3, 158–182.

[40] Grau, B.C., Motik, B., Stoilos, G. and Horrocks, I. (2012) Completeness Guarantees for Incomplete Ontology Reasoners: Theory and Practice. J. Artif. Intell. Res. (JAIR), 43, 419–476.

[41] Lutz, C., Toman, D. and Wolter, F. (2009) Conjunctive Query Answering in the Description Logic EL Using a Relational Database System. In Proc. IJCAI 2009, Pasadena, California, July 11–17, pp. 2070–2075. AAAI Press, Palo Alto, California.

[42] McBrien, P., Rizopoulos, N. and Smith, A.C. (2012) Type Inference Methods and Performance for Data in an RDBMS. In Proc. SWIM 2012, Scottsdale, AZ, USA, 20 May 6. ACM Press, New York, USA.

[43] Hayes, P. and Patel-Schneider, P. (2014) RDF 1.1 Semantics. W3C Recommendation. Latest version available at http://www.w3.org/TR/rdf11-mt/.

[44] White, T. (2015) Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (Fourth edition revised & updated). O’Reilly Media.

[45] Urbani, J., Margara, A., Jacobs, C.J.H., van Harmelen, F. and Bal, H.E. (2013) Dynamic RDF Data. In Proc. DL 2010, Sydney, Australia, October 21–25, pp. 657–672. Springer-Verlag, Berlin Heidelberg.

[46] Muñoz-Venegas, S., Pérez, J. and Gutierrez, C. (2009) Simple and Efficient Minimal RDFS. J. Web Sem., 7, 220–234.

[47] Krötzsch, M., Mehdi, A. and Rudolph, S. (2010) Orel: Database-Driven Reasoning for OWL 2 Profiles. In Proc. DL 2010, Waterloo, Ontario, Canada, May 4–7, pp. 114. CEUR-WS.org.