SQOWL: Performing OWL-DL type inference in SQL

P.J. McBrien, N. Rizopoulos, and A.C. Smith
Imperial College London

th November 2009

Abstract

In this report we describe a method to perform type inference over data stored in an RDBMS, where rules over the data are specified using OWL-DL. Since OWL-DL is an implementation of the **Description Logic** (**DL**) $\mathcal{SHOIN}(\mathbf{D})$, we are in effect implementing a method for $\mathcal{SHOIN}(\mathbf{D})$ reasoning in relational databases. Reasoning make be broken down into two processes of **classification** and **type inference**. Classification may be performed efficiently by a number of existing reasoners, and since classification alters the schema, it need only be performed once for any given relational schema as a preprocessor of the schema before creation of a database schema. However, type inference needs to be performed for each data value added to the database, and hence needs to be more tightly coupled with the database system. Previously, no technique has been proposed that implements $\mathcal{SHOIN}(\mathbf{D})$ type inference within an RDBMS. We propose such a technique, involving the use of triggers to perform reasoning over the data values as they inserted into the database. We demonstrate the soundness and performance of our approach by comparing an implementation of our technique against other existing approaches for less powerful reasoning over data in an RDBMS. The results show we provide the fastest query processing of any technique, despite having a more powerful reasoner.

1 Introduction

There is currently a growing interest in the development of systems that store and process large amounts of Semantic Web knowledge [9, 16, 19]. A common approach is to represent such knowledge as data in RDF tuples [7], together with rules in OWL-DL [1]. When large quantities of **individuals** in a ontology need to be processed efficiently, it is natural to consider that the individuals are held in a relational database management system (RDBMS), in which case we refer to the individuals as data, and make the unique name assumption. Hence, the question arises of how knowledge expressed in OWL-DL can be deployed in a relational database context, and take advantage of the RDBMS platforms in use today to process data in an ontology.

To illustrate the issues we address in this report, consider a fragment from the **terminology box** (**TBox**) of the Wine Ontology [2] expressed in DL:

$$\mathsf{Loire} \equiv \mathsf{Wine} \sqcap \mathsf{locatedIn} : \{\mathsf{LoireRegion}\} \quad (1)$$
$$\mathsf{WhiteLoire} \equiv \mathsf{Loire} \sqcap \mathsf{WhiteWine} \quad (2)$$
$$\mathsf{WhiteLoire} \sqsubseteq \forall \mathsf{madeFromGrape}.\{\mathsf{CheninBlanc}, \mathsf{PinotBlanc}, \mathsf{SauvignonBlanc}\} \quad (3)$$
$$\top \sqsubseteq \forall \mathsf{locatedIn}^-.\mathsf{Region} \quad (4)$$
$$\top \sqsubseteq \forall \mathsf{madeFromGrape}.\mathsf{Wine} \quad (5)$$
$$\top \sqsubseteq \forall \mathsf{madeFromGrape}^-.\mathsf{WineGrape} \quad (6)$$

To differentiate between classes and properties, classes start with an upper case letter, *e.g.* Wine. Properties start with a lower case letter, *e.g.* madeFromGrape. Individuals start with an upper case letter and appear inside curly brackets, *e.g.* {LoireRegion}.

Obviously, there is a simple mapping from classes and properties in DL to unary and binary relations in an RDBMS. Thus, from the above DL statements we can infer a relational schema:

```
Wine(id)        WineGrape(id)
Loire(id)       Region(id)
WhiteWine(id)   WhiteLoire(id)
madeFromGrape(domain,range)
locatedIn(domain,range)
```

Furthermore, each property has a domain and a range which can be restricted. The restrictions on the domain and range of locatedIn and madeFromGrape above allow us to infer foreign key constraints:

```
locatedIn.range → Region.id
madeFromGrape.domain → Wine.id
madeFromGrape.range → Grape.id
```

However, as it stands, the relational schema, with its closed world semantics, does not behave in the same manner to the open world semantics of the DL. For example, we can insert into the database the following facts (which in DL would be called the **assertion box** (**ABox**):

$$\mathsf{Loire}(\mathsf{SevreEtMaineMuscadet}) \quad (7)$$
$$\mathsf{WhiteWine}(\mathsf{SevreEtMaineMuscadet}) \quad (8)$$
$$\mathsf{madeFromGrape}(\mathsf{SevreEtMaineMuscadet}, \mathsf{PinotBlancGrape}) \quad (9)$$

Based on these rules, SevreEtMaineMuscadet would still not be a member of Wine, despite that being implied by TBox rule (1) from ABox rule (7) and by (5) from (9), nor is it a member of WhiteLoire, despite that being implied by TBox rule (2) from ABox rule (7) and (8) together. Performing **classification** using a reasoner on the TBox can partially solve these inconsistencies. In particular, classification would infer the following additional rules:

$$\mathsf{Loire} \sqsubseteq \mathsf{Wine} \quad (10)$$
$$\mathsf{Loire} \sqsubseteq \mathsf{locatedIn}.\{\mathsf{LoireRegion}\} \quad (11)$$
$$\mathsf{WhiteLoire} \sqsubseteq \mathsf{Loire} \quad (12)$$
$$\mathsf{WhiteLoire} \sqsubseteq \mathsf{WhiteWine} \quad (13)$$

which will then infer additional foreign key constraints:

```
Loire.id → Wine.id
WhiteLoire.id → Loire.id
WhiteLoire.id → WhiteWine.id
```

Now, the insert of data value SevreEtMaineMuscadet into Loire would be disallowed unless the data value was already in Wine. However, this does not capture the open world semantics of the DL statement, which in **type inference** performed by a reasoner on the ABox would always allow you to insert SevreEtMaineMuscadet into Loire provided that data value was either a member of Wine already, or it could be inserted into Wine. Furthermore, we still do not have inferred that SevreEtMaineMuscadet should be a member of WhiteLoire, which again type inference would give us.

When performing type inference over data in an RDBMS, we must first decide if the reasoning should be performed by a separate application outside the database, or within the database system. Taking the former approach has the disadvantage that each change to the data will require the external application to reload the data and recompute type inference, and so clearly is unsuited to applications where data is frequently updated. Thus we will study in this report performing the type inference within the RDBMS.

One previously studied approach is to use views to compute inferred types, and for each relation have an intentional definition based on rules and extensional definition with stored values. An alternative approach studied in this report is to use triggers to perform type inference as data values are inserted into the database. This approach has the advantage that since the classes are all materialised, query processing is much faster than when using the view based approach, at the cost of additional data storage for the materialised views, and additional time taken to insert data into the database. However, since most database applications are query intensive rather than update intensive, there will be a greater range of applications that would benefit from approaches that use triggers to materialise the instances of classes and properties. The SQOWL approach presented in this report is the first complete implementation of type inference for OWL-DL on data held in an RDBMS that uses the trigger based approach.

Compared to previous reasoners for use on ontologies with large numbers of individuals, our approach has the following advantages:

- In common with other rule based approaches [14, 9], our approach to type inference is much more efficient than tableaux based reasoners [3], since we do not need to use a process of refutation to infer instances as being members of classes.
- Apart from SOR [10], we are the only rule based approach implementing the full $\mathcal{SHOIN}(\mathbf{D})$ DL [3] of OWL-DL, in particular supporting `oneOf` and `hasValue` restrictions.
- Apart from DLDB2 [16], we are the only approach of any type where all type inference is performed within the RDBMS, and hence we allow RDBMS based applications to incorporate OWL-DL knowledge without alteration to the RDBMS platform.
- Since we materialise the data instances of classes, we support faster query processing than any other approach.

The remainder of this report is structured as follows. Section 2 gives an outline of how the SQOWL approach works, describing the basic technique for implementing type inference implied by OWL-DL constructs using relational schemas and triggers on the schema. The set of production rules for mapping all OWL-DL constructs to relational schemas with triggers is presented in Section 3. Section 4 runs a number of previously published benchmarks on our prototype implementation of SQOWL, and compares the performance of SQOWL to other approaches for type inference over large datasets. We give a more detailed description of related work in Section 5, and give our summary and conclusions in Section 6.

2 The SQOWL Approach

Our approach to reasoning over large volumes of data is based on a three stage approach to building the reasoning system, which we describe below, together with some technical details of the prototype implementation of the SQOWL approach that we developed into order to run the benchmark tests.

1. Classification and consistency checking of the TBox of an OWL-DL ontology is performed with any suitable reasoner to produce the inferred closure of the TBox. In our prototype system, we load an OWL-DL ontology as a Jena OWL model using the Protege-OWL API, and use Pellet [17], a tableaux based reasoner.
2. From the TBox we produce an SQL schema, that can store the classes and properties of the TBox. In our prototype system, we take the simple approach of implementing each class as a unary relation, each property as a binary relation, which generates a set of ANSI SQL `CREATE TABLE` statements, with foreign key declarations implementing the domain, range and subclass rules such as (4),(5),(6),(10),(12), and (13).
3. We use a set of production rules, that generate SQL trigger statements that perform the type inference and ABox consistency checking. The production rules map statements in OWL-DL to triggers in an abstract syntax. In our prototype system, the production rules are programmed in Java, and produce the concrete syntax of PostgreSQL function definitions and trigger definitions.

Note that once steps (1)–(3) have been performed, the database is ready to accept ABox rules such as (7),(8) and (9) implemented as insertions to the corresponding relations in the database.

We have already illustrated in the introduction how steps (1) and (2) of the above process work to produce a set of SQL tables. However, one detail omitted in the introduction is that **anonymous classes** such as that for the enumeration of individuals in TBox rule (3) will also cause a table to be created for the anonymous class (which in the example would be named `cheninblanc_pinotblanc_sauvignonblanc`).

Now we shall introduce the abstract trigger syntax we use in step (3) above, and how the triggers serve to perform type inference within the RDBMS. The triggers are ECA rules in the standard **when** *event* **if** *condition* **then** *action* form, where:

- *event* will always be some insertion of a tuple to a table, prefixed with a '$^-$' if the condition and action is to execute before the insertion of the tuple is applied to the table, or prefixed with a '$^+$' if the condition and action is to execute after the insertion of the tuple to the table is applied.
- *condition* is some Datalog query over the database. Each comma in the condition specifies a logical AND operator.
- *action* is one of
  - some list of tuple(s) to insert into the database, or
  - reject if the whole transaction involving the *event* is to be aborted, or
  - exit if the *event* can be completed normally, but nothing else done, or
  - false if the event is to be ignored. This *action* may only be used if *event* is prefixed by $-$, *i.e.* is a before trigger.

In order to perform type inference within the RDBMS, we require that we have a trigger for each table that appears in the left-hand side (LHS) of a sufficient ($\sqsubseteq$) DL rule, with that table as the *event*. The remainder of the LHS is re-evaluated in the *condition*, and if it holds, then the changes to the right-hand side (RHS) of the DL rule made as the *action*. These actions must be made before changes to the table are applied in the database, and hence we must have a 'before trigger'.

For instance, for TBox rule (12), we can identify a trigger rule:

when $^-$`WhiteLoire`(x) **if** true **then** `Loire`(x)

This in turn may be implemented by an SQL trigger, the implementation on which in PostgreSQL is presented in Figure 1(a). Due to the design of PostgreSQL, the trigger has to call a function that implements the actions of the trigger. The function `insert_Loire()` first checks whether the new tuple (`NEW.id`) already exists in `Loire`, and if not, then inserts the new tuple.

For each necessary and sufficient ($\equiv$) TBox rule, we require a trigger on any table appearing in the RHS of the rule to reevaluate the RHS and then assert the LHS after the RHS is inserted into the database. Thus the trigger will have the table in the RHS as the *action*, the remainder of the RHS in the *condition*, and the tables of the LHS

```
CREATE FUNCTION insert_Loire()
RETURNS OPAQUE AS 'BEGIN
  IF NOT EXISTS(SELECT id FROM Loire
    WHERE id = NEW.id)
   INSERT INTO Loire(id) VALUES(NEW.id);
  END IF;
RETURN NEW;
END;'
LANGUAGE 'plpgsql';

CREATE TRIGGER propagateTo_Loire
BEFORE INSERT ON WhiteLoire
FOR EACH ROW EXECUTE PROCEDURE
insert_Loire();
```

(a) WhiteLoire $\sqsubseteq$ Loire

```
CREATE FUNCTION skip_insert_Loire()
   RETURNS OPAQUE AS 'BEGIN
   IF EXISTS(SELECT id FROM Loire
     WHERE id = NEW.id)
   THEN RETURN NULL;
   END IF;
   RETURN NEW;
END;' LANGUAGE 'plpgsql';

CREATE TRIGGER skipinsert
   BEFORE INSERT ON Loire
   FOR EACH ROW EXECUTE
   PROCEDURE skip_insert_Loire();
```

(b) Loire is open world

```
CREATE FUNCTION reject_insert_cps()
RETURNS OPAQUE AS 'BEGIN
IF NOT EXISTS(SELECT id
     FROM cheninblanc_pinotblanc_sauvignonblanc
     WHERE id = NEW.id)
   THEN RAISE EXCEPTION 'Unable to change enumeration';
END IF;
RETURN NULL;
END; '
LANGUAGE 'plpgsql';

CREATE TRIGGER rejectinsert
BEFORE INSERT ON
cheninblanc_pinotblanc_sauvignonblanc
FOR EACH ROW EXECUTE PROCEDURE
reject_insert_cps();
```

(c) {CheninBlanc, PinotBlanc, SauvignonBlanc}

Figure 1: Some examples of Postgres triggers implementing type inference for DL statements

in the *action.* The trigger is an 'after trigger' since we are asserting facts equivalent to the originally asserted facts.

For example, for TBox rule (2), we have two triggers, one for each table in the RHS:

when $^{+}$`Loire`(x)
  if `WhiteWine(x)` **then** `WhiteLoire`(x)
when $^{+}$`WhiteWine`(x)
  if `Loire(x)` **then** `WhiteLoire`(x)

3 Production Rules

In this section we will describe how we translate an OWL-DL KB into ECA rules introduced in the previous section, which in turn may be easily translated into any specific implementation of SQL triggers that supports both `BEFORE` and `AFTER` triggers. Table 1 lists the OWL-DL constructs, and their equivalents in DL and FOL [6]. The following subsections describe how each of those constructs may be mapped to triggers in our abstract syntax.

There are two main constructs in OWL-DL: classes and properties. First we examine how OWL-DL classes and semantic relationships between OWL-DL classes are translated into our ECA rules, then we consider OWL-DL properties and semantic relationships between properties, and finally we examine restrictions on properties.

3.1 OWL-DL classes and individuals

An OWL-DL ontology contains declarations of classes. In our translation to SQL, each class declaration C maps to an SQL table C. The production rule is:

`Class` : $C \leadsto$ *create_table*(C),
when $^{-}C(x)$ **if** $C(x)$ **then** false

The semantics of the production rule is that any class C found in OWL-DL causes two additions to the relational schema. The first is described by the *create_table* macro, defined as follows:

create_table(C) := `CREATE TABLE` C `(id VARCHAR PRIMARY KEY)`

The second is an SQL trigger on table C to ignore any insertions of a tuple value x where x already exists in C:

when $^{-}C(x)$ **if** $C(x)$ **then** false

Note that the trigger is fired before x is actually inserted into C. If x has already been inserted, then $C(x)$ will evaluate to true, and the trigger returns false (meaning that the insertion is ignored).

Note that $C(x)$ has different translations depending on whether it appears as an *event*, as a *condition* or as an *action*:

when $^{-}C(x)$	:=	`BEFORE INSERT ON` C
when $^{+}C(x)$	:=	`AFTER INSERT ON` C
if $C(x)$	:=	`EXISTS( SELECT id FROM` C `WHERE id=`x`)`
then $C(x)$	:=	`IF NOT EXISTS (SELECT id FROM` C `WHERE id=`x`) THEN INSERT INTO` C`(id) VALUES(`x`) END IF;`

Additionally, the SQL translation of the action false is:

false := `RETURN NULL`

For example, the declaration of class Loire in TBox rule (1) produces:

create_table(Loire)
when $^{-}$Loire(x) **if** Loire(x) **then** false

which in SQL creates a table `Loire(id)` and a trigger illustrated in Figure 1(b). The trigger executes the function `skip_insert_Loire()` before an insertion on table `Loire`. The function checks whether the value to be inserted already exists. If it exists, then the function returns `NULL`, which corresponds to ignoring the insert. If it does not exist, then the function returns `NEW`, which is the value to be inserted.

An OWL-DL class may contain **individual**s. Each individual of class C will be inserted into table C with the name of the individual as the `id`. The `id` of each individual in our implementation is of type `VARCHAR`. The production rule is:

individual : $a : C \leadsto$ *insert*$(C(a))$

where *insert*$(C(a))$ is translated into SQL as follows:

insert$(C(a))$:= `INSERT INTO` C `(id) VALUES(`a`)`

3.2 Semantic relationships between OWL-DL classes

In OWL-DL classes might be related to one another. A class C might be declared to be a subclass of another class D. In this case, the translation into SQL will create a foreign key in table C that refers to table D. The

OWL Construct Name	DL Syntax		FOL
`Class`	C		$\forall x.C(x)$
`Property`	P		$\forall x, y.P(x, y)$
`TransitiveProperty`	$P \in \mathbf{P}_+$		$\forall x, y, z(P(x, y) \wedge P(y, z)) \rightarrow P(x, z)$
`SymmetricProperty`	$P \equiv P^-$		$\forall x, y(P(x, y) \Leftrightarrow P(y, x))$
`intersectionOf`	$C \sqcap D$		$C(x) \wedge D(x)$
`unionOf`	$C \sqcup D$	$\mathcal{S}$	$C(x) \vee D(x)$
`complementOf`	$\neg C$		$\neg C(x)$
`someValuesFrom`	$\exists P.C$		$\exists y.(P(x, y) \vee C(y))$
`allValuesFrom`	$\forall P.C$		$\forall y.(P(x, y) \rightarrow C(y))$
`subPropertyOf`	$P \sqsubseteq Q$	$\mathcal{H}$	$\forall x, y.P(x, y) \rightarrow Q(x, y)$
`oneOf`	$\{a_1, \ldots, a_n\}$	$\mathcal{O}$	$x = a_1 \vee \ldots \vee x = a_n$
`hasValue`	$P{:}\{a\}$		$P(x, a)$
`inverseOf`	$P \equiv Q^-$	$\mathcal{I}$	$\forall x, y.P(x, y) \Leftrightarrow Q(y, x)$
`FunctionalProperty`	$\top \sqsubseteq \leqslant 1P$		$\forall x, y, z.(P(x, y) \wedge P(x, z)) \rightarrow y = z$
`minCardinality`	$\geqslant nP$	$\mathcal{N}$	$\exists y_1, \ldots, y_n. \bigwedge_{1 \leqslant i \leqslant n}(P(x, y_i)) \wedge \bigwedge_{1 \leqslant i < n, i < j \leqslant n} y_i \neq y_j$
`maxCardinality`	$\leqslant nP$		$\forall y_1, \ldots, y_{n+1}. \bigwedge_{1 \leqslant i \leqslant n+1}(P(x, y_i)) \rightarrow (\bigvee_{1 \leqslant i < j \leqslant n+1} y_i = y_j)$
`subClassOf`	$C \sqsubseteq D$		$\forall x.C(x) \rightarrow D(x)$
`equivalentClass`	$C \equiv D$		$\forall x.C(x) \rightarrow D(x) \wedge \forall x.D(x) \rightarrow C(x)$
`range`	$\top \sqsubseteq \forall P.C$		$\forall x.P(x, y) \rightarrow C(y)$
`domain`	$\top \sqsubseteq \forall P^-.C$		$\forall x.P(y, x) \rightarrow C(y)$
`Thing`	$\top$		$\forall x.x \in \mathcal{D}$
`Nothing`	$\bot$		$\forall x.x \notin \mathcal{D}$
individual	$a{:}C$		$C(a)$
property of individual	$\langle a, b\rangle{:}P$		$P(a, b)$

Table 1: OWL constructors and their DL and FOL equivalents

foreign_key macro that produces the SQL code is defined as follows:

$foreign\_key\_for\_class(C, D)$:= `ALTER TABLE` C `ADD FOREIGN KEY (id) REFERENCES` D `(id)`

Additionally, an SQL trigger is added on table C to specify that before inserting any tuple x in C, the tuple must be inserted in D. This trigger allows for forward chaining inference. The production rule for the subclass relationship is the following:

`subClassOf` : $C \sqsubseteq D \rightsquigarrow foreign\_key(C, D)$
when $^{-}C(x)$ **if** true **then** $D(x)$

An example of the SQL code generated by the above ECA rule is shown in Figure 1(a) for the TBox rule (10).

A class D might be declared to be the complement of another class C. In this case, one trigger is created that checks whether a tuple $\{x\}$ exists in C, before x is inserted in D. If it does exist, then the insertion is rejected and the transaction that initiated it is rolled back. Similarly, another trigger is created that checks insertions in table C. The production rule is:

`complementOf` : $D \equiv \neg C \rightsquigarrow$
when $^{-}D(x)$ **if** $C(x)$ **then** reject
when $^{-}C(x)$ **if** $D(x)$ **then** reject

where reject in PostgreSQL is defined as:

reject := `RAISE EXCEPTION 'error message'`

Note that no tuples are inferred for any of the classes based on this construct. For example, in the Wine ontology we have that

$$\text{NonConsumableThing} \equiv \neg\text{ConsumableThing} \tag{14}$$

If the statement ConsumableThing(SevreEtMaineMuscadet) is not true, then we cannot infer that NonConsumableThing(SevreEtMaineMuscadet) is true. The reason is that inference in OWL-DL is based on the open world assumption (OWA) and thus negation as failure does not apply.

In OWL-DL, a class C_1 might also be declared to be the union of classes C and D. Based on *classification*, this implies that both C and D are subclasses of C_1. Thus, the *type inference* on the union relationship can be performed using the `subClassOf` construct. The same holds for intersection. A class C_1 might be declared to be the intersection of classes C and D. Based on classification, this implies that C_1 is a subclass of both C and D, and therefore type inference for the intersection can be performed based on these subclass relationships.

Additionally, a class C_1 might be declared to be disjoint with another class C_2, which indicates that the two classes do not have any individuals in common. In SQL we treat this case as the complement relationship.

Finally, OWL-DL allows the definition of enumeration classes using the oneOf construct. The oneOf construct enables a class C to be defined by exhaustively enumerating its instances, $\{a_1, a_2, \ldots, a_n\}$. The extent of the defined class contains exactly the enumerated individuals, not more or less. In our system where we make the unique name assumption, the enumeration class corresponds to a table that contains only the instances $a_1, a_2, \ldots, a_n$. The production rule of oneOf first inserts the instances into table C. Then creates a trigger that discards any further inserts into table C.

`oneOf` : $C \equiv \{a_1, \ldots, a_n\} \rightsquigarrow$
$insert(C(a_1)), \ldots, insert(C(a_n))$
when $^{-}C(x)$ **if** true **then** reject

For example, the TBox rule (3) in the introduction, introduces an anonymous class which is an enumeration. The anonymous class is translated into a table, and then the production rule for the enumeration performs three inserts on that table:

insert(`cheninblanc_pinotblanc_sauvignonblanc(CheninBlanc)`)
insert(`cheninblanc_pinotblanc_sauvignonblanc(PinotBlanc)`)
insert(`cheninblanc_pinotblanc_sauvignonblanc(SauvignonBlanc)`)

and then defines the trigger

when $^{-}$`cheninblanc_pinotblanc_sauvignonblanc`(x)
if $\neg$`cheninblanc_pinotblanc_sauvignonblanc`(x)
then reject

which causes any further inserts on that table to cause the transaction in which they take place to abort. The implementation of this trigger in PostgreSQL is illustrated in Figure 1(c).

3.3 OWL-DL properties

In this section, we are going to examine how OWL-DL properties and relationships between these properties are translated into SQL.

An OWL **property** defines a binary predicate, $P(D, R)$, where D is called the domain and R the range of the property P. There are two types of properties in OWL-DL:

datatype properties are constructs whose domain is an object class and range is a datatype

object properties are constructs whose both domain and range is an object class

A **datatype property** with range R, states that the value of the property, *i.e.* the range, comes from the datatype R. Available datatypes in OWL-DL, include RDF literals and XML Schema datatypes as defined in [4]. The instances of a datatype class are members of the sets defined for the equivalent XML Schema datatype or set of RDF literals.

The SQL translation of a datatype property $P(D, R)$ is an SQL table P. The $create\_table\_for\_property(P(D, R))$ macro is used to define the table:

```
$create\_table\_for\_datatype\_property(P(D,R))$ :=
    CREATE TABLE $P$
    (domain VARCHAR, range $sqlType(R)$).
```

function $sqlType(R)$ returns the SQL data type corresponding to the OWL-DL datatype R.

The domain of a property $P(D, R)$ is a class D. Therefore, in our translation column `domain` of table P is of type `VARCHAR`, since it represents the `id`s of individuals of D. Additionally, `domain` is specified as a foreign key that refers to column `id` of D. A trigger is defined to be executed before each insertion of a tuple $\{x, y\}$ on table P. If the tuple $\{x\}$ does not exist in table D then it is inserted into D. Finally, a trigger is defined to ignore any inserts of tuples that already exist. The production rule is as follows:

```
datatypeProperty : $P(D,R) \rightsquigarrow$
  $create\_table\_for\_property(P)$,
  $foreign\_key\_for\_property(P(D), D)$
  when $^{-}P(x,y)$ if true then $D(x)$
  when $^{-}P(x,y)$ if $P(x,y)$ then false
```

The $foreign\_key\_for\_property(P(D), D)$ macro is defined as:

```
$foreign\_key\_for\_property(P(D),D)$ :=
    ALTER TABLE $P$
    ADD FOREIGN KEY ($D$)
    REFERENCES $D$ (id).
```

An **object property** links two classes. The production rule for object properties is similar to the datatype properties rule. There are now two foreign key constraints and the trigger executed before each insertion checks both the `domain` value and the `range` value and propagate them to the corresponding classes.

```
objectProperty : $P(D,R) \rightsquigarrow$
  $create\_table\_for\_object\_property(P)$,
  $foreign\_key\_for\_property(P(D),D)$
  $foreign\_key\_for\_property(P(R),R)$
  when $^{-}P(x,y)$
      if true then $D(x), R(y)$
  when $^{-}P(x,y)$ if $P(x,y)$ then false
```

Properties can be **functional** and/or **inverse functional**. A functional property $P(D, R)$ can have only one tuple $\{x, y\}$ for each x in D, and an inverse functional can have only one tuple $\{x, y\}$ for each y in R. We translate these restrictions into SQL as primary key constraints. For example, in the Wine ontology property we have that:

$$\top \sqsubseteq \forall \mathsf{hasFlavor}.\mathsf{Wine} \tag{15}$$

$$\top \sqsubseteq \forall \mathsf{hasFlavor}^{-}.\mathsf{WineFlavor} \tag{16}$$

$$\top \sqsubseteq \leqslant 1 \mathsf{hasFlavor} \tag{17}$$

which means that the property hasFlavor with domain Wine and range WineFlavor is functional, *i.e.* each Wine can only have one WineFlavor. Therefore, we add a primary key constraint on table `hasFlavor` on its `domain` column. This constraint will not allow the same wine to appear in the `hasFlavor` table twice, therefore it will enforce the functional constraint on the property. The production rule of a functional object property P will create the table P for the property as follows:

```
CREATE TABLE $P$
(domain VARCHAR PRIMARY KEY, range VARCHAR).
```

If the property is inverse functional then the rule adds a primary key constraint on the `range` column, and if it is both functional and inverse functional the primary key is defined on both `domain` and `range`.

Properties can also be **transitive** and/or **symmetric**. For example, the locatedIn property in the Wine ontology is transitive. This means that if we already know that locatedIn(ChateauChevalBlancStEmilion,BordeauxRegion) and locatedIn(BordeauxRegion,FrenchRegion), then we can infer that locatedIn(ChateauChevalBlancStEmilion,FrenchRegion) is also true.

The production rule for a transitive property P needs to define a trigger to be executed after each insert of tuple $\{x, y\}$ in P. The rule will insert for each $\{y, z\}$ existing in P the tuple $\{x, z\}$ and for each $\{z, x\}$ in P the tuple $\{z, y\}$ will be inserted. The macro $foreach$ must be used in the production rule that performs these iterations:

```
$foreach(z, P(y,z), P(x,z))$ :=
  FOR $z$ IN (SELECT range FROM $P$
    WHERE domain= $y$)
  LOOP IF $x,z$ NOT IN SELECT domain,range FROM $P$
    THEN INSERT INTO $P$ VALUES $x,z$
  END IF; END LOOP;
```

The production rule is as follows:

```
TransitiveProperty : $P \in \mathbf{P}_{+} \rightsquigarrow$
  when $^{+}P(x,y)$ if true then
    $foreach(z, P(y,z), P(x,z))$,
    $foreach(z, P(z,x), P(z,y))$
```

If a property P is declared to be symmetric, then a rule needs to be defined that will insert in P the tuple $\{y, x\}$ after an event inserts tuple $\{x, y\}$ on P:

```
SymmetricProperty : $P \equiv P^{-} \rightsquigarrow$
  when $^{+}P(x,y)$ if true then $P(y,x)$
```

3.4 Semantic relationships between OWL-DL properties

Like classes, OWL-DL properties can be related to one another. For example, a property P might be declared to be a subproperty of Q, which means that each tuple of P is also a tuple of Q. For example, in the Wine ontology hasFlavor is a subproperty of hasWineDescriptor:

$$\mathsf{hasFlavor} \sqsubseteq \mathsf{hasWineDescriptor}$$

An SQL trigger is added on table P to specify that after inserting any tuple $\{x, y\}$ in P, then the tuple must be inserted in Q. The production rule is as follows:

```
subPropertyOf : $P \sqsubseteq Q \rightsquigarrow$
  when $^{+}P(x,y)$ if true then $Q(x,y)$
```

A property P might be declared as the inverse of another property Q. This declaration asserts that for each tuple $\{x, y\}$ in P, the inverse tuple $\{y, x\}$ exists in Q, and vice versa. An SQL trigger is added on table P to specify that after each insertion on P the inverse tuple must be inserted on Q, if it does not already exist. Note that in our methodology, for each such property declaration two `inverseOf` constructs are created: $P \equiv Q^-$ and $Q \equiv P^-$. The production rule for the `inverseOf` construct is :

`inverseOf` : $P \equiv Q^- \rightsquigarrow$
when $^+P(x,y)$ **if** $\neg Q(y,x)$ **then** $Q(y,x)$

Finally, a property P might be declared to be equivalent to another property Q. In this case, an SQL trigger is added on table P that after each insertion of tuple $\{x,y\}$ in P the trigger inserts the tuple on table Q, and vice versa. The production rule is:

`equivalentProperty` : $P \equiv Q \rightsquigarrow$
when $^+P(x,y)$ **if** true **then** $Q(x,y)$
when $^+Q(x,y)$ **if** true **then** $P(x,y)$

3.5 Restrictions on Properties

Properties can be used to define restrictions on classes. Each restriction on a property $P_i(D_i,R_i)$ forms an anonymous class S_i. A set of restrictions on multiple properties $P_1(D_1,R_1),\dots,P_n(D_n,R_n)$ can be used to define a **named class** C, which becomes a subclass of all S_i classes.

- The allValuesFrom restriction on property P_i requires that for each instance x in S_i and each tuple $\{x,y\}$ in P_i, y is a member of the class indicated by the allValuesFrom clause. If y is not a member of the particular class then it is inferred to be. Note that the allValuesFrom restriction can be satisfied trivially if there are no tuples for property P_i.
  If each individual x of D is restricted to have all values for property P_i from class C then this translates into a trigger executed after an insertion of tuple x in table D. If x is not associated with any value in table P_i, then the restriction is satisfied. If a tuple $\{x,y\}$ exists in table P_i such that $\{y\}$ is not a tuple of C, then the trigger inserts tuple $\{y\}$ in C.
  `allValuesFrom` : $D \sqsubseteq \forall P_i.C \rightsquigarrow$
  when $^+D(x)$ **if** $P_i(x,y)$ **then** $C(y)$
  For example, based on TBox rule (6), we know that for each individual x which has a tuple $\{x,y\}$ in madeFromGrape, then y is a WineGrape. Thus, based on ABox rule (9) we can infer that PinotBlancGrape is a WineGrape.
- someValuesFrom requires for each x in S_i *at least one* tuple $\{x,y\}$ of P_i exists with y a member of the class indicated by the someValuesFrom restriction. For the implementation of this restriction we could define that for each insertion $D(x)$ on the class we insert a tuple $\{x, null\}$ on P_i. However, this is not necessary since the existence of the tuple $\{x, null\}$ cannot be used for any kind of inference. Thus the rule's body is empty $-$.
  `someValuesFrom` : $D \sqsubseteq \exists P_i.C \rightsquigarrow -$
  For example, based on the rules
  Wine(TaylorPort) (18)
  Wine $\sqsubseteq \exists$locatedIn.Region (19)
  we know that TaylorPort is locatedIn a Region, but we cannot insert any tuples on table `locatedIn` since we do not know the exact Region.
- cardinality, minCardinality, maxCardinality restrict the number of tuples $\{x,y\}$ in P_i an individual x of S_i can have. As in the previous restriction we could create a rule that adds tuples $\{x, null\}$ so that the cardinality restriction is satisfied. However, these tuples cannot be used for inference therefore the rule's body is empty, *e.g.*
  `cardinality` : $C \sqsubseteq = nP_i \rightsquigarrow -$
- hasValue specifies that each individual x of S_i has a tuple $\{x,a\}$ in P_i, where a is the value indicated by the hasValue restriction. If a class D is specified to be a subclass of S_i, then each insert of tuple $\{x\}$ on table D initiates a trigger which inserts into table P the tuple $\{x,a\}$.
  `hasValue` : $D \sqsubseteq \exists P_i : a \rightsquigarrow$
  when $^+D(x)$ **if** true **then** $P_i(x,a)$
  For example, based on the TBox rule (11) we know that each Loire wine is locatedIn LoireRegion. Thus, when the ABox rule (7) is examined, the trigger will insert tuple $\{$SevreEtMaineMuscadet, LoireRegion$\}$ in table `locatedIn`.

3.6 Complete and Partial Classes

Restrictions can be *necessary* ($\sqsubseteq$) or *necessary and sufficient* ($\equiv$). Named classes that have only necessary conditions are **partial** or **primitive** classes. Those that have at least one necessary and sufficient condition are called **complete** or **defined** classes.

The necessary and sufficient restrictions for a named class C need to be satisfied in order for an individual x to be inferred to be of class C. Each restriction r_i can either be on a property, as we show in the previous section, and thus forms an anonymous class S_i, or it can be another named class D_i. Class C is then equivalent to the intersection of all S_i and D_i classes. The `equivalentClass` construct specifies this intersection.

For example, TBox rule (2) specifies that WhiteLoire is a class equivalent to the intersection of Loire and WhiteWine. In this case, if there is an individual x which is both a member of Loire and a member of WhiteWine, then it is going to be inferred to be a member of WhiteLoire. The SQL translation should insert tuple $\{x\}$ in table `WhiteLoire`.

To achieve this we would need a trigger which is executed after a tuple $\{x\}$ is inserted in table `Loire`. The trigger would check if $\{x\}$ is a tuple in `WhiteWine` and if it is it would insert $\{x\}$ in table `WhiteLoire`:

when $^+$`Loire`(x) **if** `WhiteWine(x)` **then** `WhiteLoire`(x)

Additionally, we would need a similar trigger on table `WhiteWine`. In this case, the trigger would check if tuple $\{x\}$ is in `Loire` and if it is then it would insert it in `WhiteLoire`:

when $^+$`WhiteWine`(x) **if** `Loire(x)` **then** `WhiteLoire`(x)

Thus, we see that for an `equivalentClass` construct we need to create triggers for each class in the intersection.

The production rule for the `equivalentClass` construct is:

`equivalentClass` : $C \equiv C_1 \sqcap C_2 \sqcap \dots C_n \rightsquigarrow$
$rules(C, C_1, C_2 \sqcap \dots \sqcap C_n)$
$rules(C, C_2, C_1 \sqcap C_3 \sqcap \dots \sqcap C_n)$
$rules(C, C_3, C_1 \sqcap C_2 \sqcap C_4 \dots \sqcap C_n)$
...
$rules(C, C_n, C_1 \sqcap \dots \sqcap C_{n-1})$

Each C_i in the production rule can either be a restriction on a property forming an anonymous class S_i or a named class D_i, which implies the restriction $C \sqsubseteq D_i$.

The function $rules(C, C_i, E_1 \sqcap \dots E_n)$ performs a logical AND of the conditions that must be satisfied for C_i to be true with the conditions that must be satisfied for the intersection $E_1 \sqcap \dots E_n$ to be true. If all conditions are satisfied then the function will insert a new tuple for C. Each call to *rules* produces one `AFTER` trigger if C_i is a named class, *i.e.* if $C_i \equiv D_i$. The trigger is executed after an insertion on D_i. A call to *rules* produces one `AFTER` trigger if C_i is either a hasValue or a minCardinality restriction on property P. The trigger is performed after an insertion on P. Finally a call to *rules* produces two `AFTER` triggers if C_i is a someValuesFrom restriction $\exists P.D$. One trigger is executed after an insertion in D and another after an insertion in P.

$rules(C, D_i, E_1 \sqcap \ldots E_m) :=$
  when $^+D_i(x)$ if $cond(x, E_1 \sqcap \ldots E_m)$ **then** $C(x)$
$rules(C, P : \{a\}, E_1 \sqcap \ldots E_m) :=$
  when $^+P(x, \_)$ if $P(x, a), cond(x, E_1 \sqcap \ldots E_m)$ **then** $C(x)$
$rules(C, > nP, E_1 \sqcap \ldots E_m) :=$
  when $^+P(x, \_)$ if $count(P(x, \_)) > n, cond(x, E_1 \sqcap \ldots E_m)$ **then** $C(x)$
$rules(C, \exists P.D, E_1 \sqcap \ldots E_m) :=$
  when $^+D(x)$ if $P(x, y), D(y), cond(x, E_1 \sqcap \ldots E_m)$ **then** $C(x)$
  when $^+P(x, \_)$ if $P(x, y), D(y),\ cond(x, E_1 \sqcap \ldots E_m)$ **then** $C(x)$

where the function *cond* is used to define the condition in the triggers and it is defined as follows:

$cond(x, D \sqcap E_2 \sqcap \ldots E_m) := D(x), cond(E_2 \sqcap \ldots E_m)$
$cond(x, P : \{a\} \sqcap E_2 \sqcap \ldots E_m) := P(x, a), cond(E_2 \sqcap \ldots E_m)$
$cond(x, \exists P.D \sqcap E_2 \sqcap \ldots E_m) := P(x, y), D(y), cond(E_2 \sqcap \ldots E_m)$
$cond(x, > nP \sqcap E_2 \sqcap \ldots E_m) := count(P(x, \_)) > n, cond(E_2 \sqcap \ldots E_m)$
$cond(x, \textbf{empty}) :=$ true

The keyword **empty** stands for any empty intersection expression. It is used to specify the base case for the recursive definition of *cond*. The function $count(P(x, \_))$ returns the number of tuples $\{x, y\}$ in P for individual x.

For example, if we apply the above production rule to the TBox rule (2), we have that $C_1 \equiv$ Loire and $C_2 \equiv$ WhiteWine. The rule performs the following calls:
  rules(WhiteLoire, Loire, WhiteWine)
  rules(WhiteLoire, WhiteWine, Loire)

Regarding the first call, based on the definition of *cond*, we have that *cond*(x, WhiteWine) := WhiteWine(x), **true**, which is logically equivalent to WhiteWine(x). Thus, the call generates:
  when $^+$Loire(x) if WhiteWine(x) **then** WhiteLoire(x)

Similarly, the second call to *rules* generates
  rules(WhiteLoire, WhiteWine, Loire) :=
    when $^+$WhiteWine(x) if Loire(x) **then** WhiteLoire(x)

Thus, the production rule successfully generates the two ECA rules we talked about in the beginning of this section.

As another example, we can examine the TBox rule (1). In this case, $C_1 \equiv$ Wine and $C_2 \equiv$ locatedIn : {LoireRegion}. The production rule performs the following calls:
  rules(Loire, Wine, locatedIn : {LoireRegion})
  rules(Loire, locatedIn : {LoireRegion}, Wine)

The first call examines that the `hasValue` restriction is satisfied. It generates the ECA rule:
  when $^+$Wine(x) if locatedIn(x, LoireRegion) **then** Loire(x)

because
  cond(x, locatedIn : {LoireRegion}) :=
    locatedIn(x, LoireRegion)

The second call checks that both the `hasValue` and the `subClassOf` restriction are satisfied. Note that in the first call we do not need to examine the `subClassOf` restriction because we know that it is already satisfied since the trigger is performed after an insert on the super-class Wine. The second *rules* call generates the ECA rule
  when $^+$locatedIn(x, _) if
    locatedIn(x, LoireRegion), Wine(x) **then** Loire(x)

because *cond*(x, Wine) := Wine(x). The complete implementation of TBox rule (1) is presented in Figure 2.

Note that the cardinality and maxCardinality constructs are not dealt with in the `equivalentClass` construct. The reason is that due to the open world assumption we cannot be certain about the cardinality of a property (except if the property is functional). However, the `equivalentClass` construct deals with the minCardinality restriction. Similarly, the complementOf restriction is not examined, because if we do not have information about an individual we cannot be certain that the individual is not a member of a class. For example, if we have a restriction $C \sqsubseteq \neg C1$ in the equivalentClass construct, we cannot use it for inference, *e.g.* even if x is not a tuple in $C1$ that does not mean that we can infer that it is in C. Similarly, we cannot make any inference based on the allValuesFrom restriction therefore it is not considered in the equivalentClass construct.

```
CREATE FUNCTION define_Loire_Wine()
RETURNS OPAQUE AS 'BEGIN
   IF (EXISTS (SELECT domain FROM locatedIn
       WHERE domain=NEW.id and range='LoireRegion'))
       INSERT INTO Loire(id) VALUES (NEW.id);
   END IF; END;'
LANGUAGE 'plpgsql';

CREATE TRIGGER define_Loire_Wine
AFTER INSERT ON Wine
FOR EACH ROW EXECUTE PROCEDURE
define_Loire_Wine();

CREATE FUNCTION define_Loire_locatedIn()
RETURNS OPAQUE AS 'BEGIN
   IF (EXISTS (SELECT domain FROM locatedIn
       WHERE domain=NEW.id and range='LoireRegion'))
   AND (EXISTS (SELECT id FROM Wine
       WHERE id=NEW.id))
       INSERT INTO Loire(id) VALUES (NEW.id);
   END IF; END;'
LANGUAGE 'plpgsql';

CREATE TRIGGER define_Loire_locatedIn
AFTER INSERT ON locatedIn
FOR EACH ROW EXECUTE PROCEDURE
define_Loire_Wine();
```

Figure 2: The Postgres trigger implementing type inference for DL statement Loire $\equiv$ Wine $\sqcap$ locatedInt:{LoireRegion}

3.7 Semantic relationships between individuals

OWL-DL does not use the **Unique Name Assumption** (**UNA**) so individuals are not assumed to be distinct unless we explicitly state that they are using the differentFrom or AllDifferent constructs. The differentFrom construct indicates that two individuals a_1, a_2 are different from each other. The AllDifferent construct indicates that all associated individuals are pairwise different. Finally, OWL-DL includes the sameAs construct, which indicates that one individual a_1 refers to an existing individual a_2, *i.e.* the two individuals have the same identity.

In our SQL implementation, we do not support any of these constructs since SQL makes the UNA.

4 Experimental Evaluation

In common with other rule based approaches, we demonstrate the soundness of our approach by running a number of benchmarks previously used. In the next subsection we review the three benchmarks we have tested our prototype system with, then in Section 4.2 describe how SPARQL queries in the benchmarks may be translated into queries over our relational schema. The following three subsections then represent the results of running the three benchmarks.

4.1 OWL-DL Reasoner Benchmarks

We have chosen three ontologies to test our system on: The **Lehigh University Ontology Benchmark** (**LUBM**) [16], the W3C Wine ontology [2] and the **University Ontology Benchmark** (**UOBM**) [12]. These all pose significant, but different, type inference challenges. LUBM requires only simple inference, and comes with a data generator that can be used to create ontologies with large numbers of individuals. This was used to test the scalability of our approach. The Wine ontology does not have many individuals but has a complex TBox. We used this to test the inference capabilities of SQOWL. Finally, UOBM has a large number of individuals and a relatively complex ontology. We used this to test how well our approach scaled on ontologies with more complex inference rules. Other ontologies that have been used to benchmark DL reasoning systems [14] test their classification capabilities. As we are using existing reasoners to perform ontology classification, these ontologies are not relevant when testing our system.

LUBM and UOBM come with a set of queries and results, which we used to test the soundness of our approach. To test SQOWL against the Wine ontology, we compared our inference results with those of Pellet [17], the well established tableaux reasoner for $\mathcal{SHOIN}(\mathbf{D})$, and used its results as the reference for what instances should be inferred. Hence all three benchmarks could be used to demonstrate the completeness and soundness of our results.

The LUBM and UOBM queries are given in SPARQL. This reflects the fact that the systems they were developed for, DLDB2 and SOR respectively, have been designed in such a way that queries are posed on the source ontology. Our approach is different, in that we expect our database to be used independently of the ontology, so our queries are posed in SQL. In Section 4.2 we describe how we translate SPARQL queries like those in LUBM and UOBM into SQL queries appropriate for the schema we create.

The tests were carried out on a Windows Vista platform with an Intel Centrino 2 processor running at 2.6 GHz and 4GB of RAM. The Java VM was allocated 800M. DLDB2 exists as part of Hawk version 3 and uses MySQL as the RDBMS (we used version 5.1 for benchmarks), SQOWL used Postgres) version 8.3 for benchmarks, and SOR uses DB2 (Express 9.7 was used for the benchmarks).

4.2 Translating SPARQL queries into SQL

Queries in our are system are posed in the form of SQL SELECT statements. In this section we describe how we translate SPARQL queries from the LUBM and UOBM benchmarks into SQL statements appropriate for SQOWL.

Each `rdf:type` tuple maps to a table generated by a OWL class, while the other triples map to the tables generated by the respective OWL properties. To create the required query we join these tables together. Any literals in the SPARQL queries translate into literals in the SQL query. For example, the SPARQL query from the LUBM benchmark query 8:

```
SELECT DISTINCT ?X, ?Y, ?Z
WHERE {
  ?X rdf:type ub:Student .
  ?Y rdf:type ub:Department .
  ?X ub:memberOf ?Y .
  ?Y ub:subOrganizationOf <http://www.University0.edu> .
  ?X ub:emailAddress ?Z
}
```

translates into the following SELECT statement in SQOWL:

```
SELECT DISTINCT s.id, d.id, e.range
FROM ub_student s,
  JOIN ub_memberOf m ON p.id = m.domain
  JOIN ub_department d ON d.id = m.range
  JOIN ub_suborganizationof so ON d.id = so.range
  JOIN ub_emailaddress e ON s.id = e.domain
WHERE so.range = 'http://www.University0.edu';
```

4.3 LUBM

The LUBM ontology classifies individuals in a university. These individuals may be members of staff, publications, departments, research groups *etc.* Each university contains about 100000 individuals and property tuples. LUBM(1) contains one university, LUBM(2) two universities, and so on.

SQOWL and SOR were able to answer all the LUBM queries. DLDB2 failed on Q11 returning no results. Tables 2 show the query response times for the fourteen LUBM queries on LUBM(1) and LUBM(3) respectively. As expected SQOWL and SOR exhibit much faster query response times than DLDB2 because they materialise their inference results at load time. The slightly faster query response times for SQOWL over SOR are also to be expected, as the materialised schema produced by SQOWL avoids the many joins necessary to answer queries in the SOR view-based approach.

4.4 The Wine Ontology

We chose the Wine ontology [2] because it is the well known example OWL-DL ontology provided by the W3C, and it since it includes an example of each type of OWL-DL construct, type inference requires the processing of all the OWL-DL constructors. There are no widely used queries to test type inference for the Wine ontology, and it does not lend itself to the sort of queries that are used to test LUBM and UOBM. For these reasons we test our results against the type inference calculated by the tableaux reasoner Pellet, embedded in Protege 3.4.1 [18].

The queries used, which we call Q1 and Q2 respectively, were:

```
SELECT DISTINCT ?X
WHERE {
  ?X rdf:type wine:WhiteLoire .
}

SELECT DISTINCT ?X
WHERE {
  ?X rdf:type wine:AmericanWine .
}
```

We checked the WhiteLoire class as we have used it in our example throughout the report and the AmericanWine class because it was used in [14].

	Q1	Q2
SOR	1/2 16ms	24/24 15ms
SQOWL	2/2 0.5ms	24/24 0.5ms

Table 4: Wine Completeness and Soundness

The number of results for each query compared to the results obtained from Pellet, as well as the time taken, are shown in 4. Our system was able to correctly infer all the WhiteLoire and AmericanWine instances. SOR could not correctly infer one of the two WhiteLoire instances. The difference in query result times is greater here than in LUBM because of the greater complexity of the inference required to answer the queries. SOR needs to make joins for each restriction on a derived class that appears in a query. This benchmark includes derived classes with

System	Size	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q1
DLDB2	LUBM(1)	706	841	539	912	788	18649	19652	128096	20246	19877	–	110	538	67
SOR	LUBM(1)	4	47	3	30	9	16	7	114	58	3	4	7	3	15
	LUBM(3)	112	3650	150	2205	219	120	210	874	2730	140	168	359	187	93
SQOWL	LUBM(1)	8	12	4	14	6	2	11	30	33	5	2	4	1	2
	LUBM(3)	35	46	23	53	20	6	36	67	150	24	2	3	5	6

Table 2: LUBM Query Answer Times (ms)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13
DLDB2	100	82	100	100	0	0	0	0	0	83	0	20	56
SOR	100	100	100	100	100	100	100	100	100	100	100	100	64
SQOWL	100	100	100	100	100	98	100	99	100	88	97	100	99

Table 3: UOBM Query Completeness in percent

complex restrictions, and so more joins are required. All our type inference results are materialised at load time so we are able to directly query the relevant tables, hence our very fast response times.

4.5 UOBM

The UOBM is an extension of the LUBM with a slightly higher number of individuals but requiring significantly more complex inference testing more of the OWL-DL constructors [12]. There are two versions of the UOBM, an OWL-DL version and an OWL-Lite version. We have used the OWL-DL version for our tests.

We used the queries provided with UOBM to test SQOWL. All results for systems tested were sound, but DLDB2 was only able to answer three of the queries correctly as it is not able to perform as many type inference tasks as SQOWL or SOR. The completeness results are shown in Table 3. In the case of queries 13 and 8 SQOWL misses one individual. This is an individual that was the same as some other individual. This happens because OWL-DL does not adopt the UNA. Our system does so this discrepancy was to be expected. We are also missing results from Q10 and Q11 for the same reason. We provide a more complete answer to Q13 than SOR does as their system does not handle minimum cardinality fully [19].

The query times for the thirteen UOBM queries are shown in Table 5. We have not included query times for DLDB2 as it provided complete answers for so few of the queries. Once again our query times are faster than SOR. The complexity of the inference required in UOBM is somewhere between that of LUBM and Wine ontology so the difference in times here is in line with expectations.

5 Related Work

DL reasoners come in a number of forms [3], those based on tableaux algorithms are being the most common. Tableaux based reasoners like Racer, FacT++ and Pellet are very efficient at computing classification hierarchies and checking the consistency of a knowledge base. However, the tableaux based approach is not suited to the task of processing ontologies with large datasets (*i.e.* large numbers of individuals), since the tableaux algorithm uses a refutation procedure rather than a query answering algorithm [8].

Rule based reasoners provide an alternative to the tableaux based approach that is more promising for handling large datasets. Some of the best known reasoners are summarised in Table 6.

O-DEVICE [13] translates OWL rules into an in-memory representation using the CLIPS production rules system and the COOL OO language. It can process all of OWL-Lite, and in addition the OWL-DL constructs partial union of classes `hasValue` and class disjointedness. It does not support `oneOf`, `complementOf` or data ranges. The fact that the system is memory based provides fast load and query times, but means that it does not scale beyond tens of thousands of individuals. OWLIM [9] also does its reasoning in memory. It takes rules already defined for RDFS inference in the SAIL (Storage And Inference Layer) of Sesame [5], and adds support for a small subset of OWL-DL constructs, up to the expressiveness of Horn Logic. In common with O-DEVICE its reasoning is fast, but it cannot reason of large numbers of individuals.

KAON2 [14] does reasoning by means of theorem proving. The TBox is translated into first-order clauses, which are executed on a disjunctive Datalog engine of their own design to compute the inferred closure. KAON2 displays impressive load and query times but is unable to handle **nominals** (*i.e.* `hasValue` and `oneOf`, labelled as O in DL). DLog [11] adopts a similar approach, but uses Prolog to answer queries on individuals that are stored in an RDBMS. It performance characteristics are similar to those of KAON2.

Those most closely related to our approach are DLDB2 [15, 16] and SOR [19] (previously called Minerva) in that they use an RDBMS as their rule engine. DLDB2 stores the rules inside the database as views and does not materialise the inferred closure of the ontology at load time. Tables are created for each atomic role and class which are then populated with the individuals from the ontology. A separate DL reasoner is used to classify the ontology. The resulting TBox axioms are translated into non-recursive Datalog rules that are translated in SQL view create statements. DLDB2 enjoys very fast load times because the inferred closure of the database is not calculated at load time but its querying is slow. An advantage of the system is that because the closure is only calculated when queries are posed on the system, updates and deletes can be performed on the system.

SOR [19] also uses a standard tableaux based DL reasoner to first classify the ontology. It differs from DLDB2 in that rules are kept outside the database and the SQL statements created from the OWL-DL rules are not used to create views but are rather executed at load time materialise the inference results. This makes loading slower but query processing faster. SOR uses a fixed database schema that is not related to the ontology it is processing but rather to the OWL-DL constructors the system is modelling. For example, there are tables called hasValue and someValuesFrom. Each derived class whose restrictions include one of these constructors has an entry in the relevant table. At query time joins are created over these tables to provide the necessary reasoning capability. Because the rules are kept outside the database, any addi-

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13
SOR	600	280	1400	500	327	1216	608	265	5756	328	11248	561	1248
SQOWL	88	67	215	322	88	129	188	130	57	200	105	60	57

Table 5: UOBM Query Answer Times

Name	DL reasoning	Rules engine
O-DEVICE	$\mathcal{SHIQ}$ + hasValue	CLIPS rule engine
Dlog	$\mathcal{SHIQ}$	Prolog
KOAN2	$\mathcal{SHIQ}$	Bespoke Datalog engine
OWLIM	RDFS	Bespoke + Sesame
DLDB2	$\mathcal{SHOIN}(D)$ classification, DLP except `allValues` type inference	RDBMS (Views)
SOR	$\mathcal{SHOIN}(D)$ classification, DLP type inference	Java application
SQOWL	$\mathcal{SHOIN}(D)$	RDBMS (Triggers)

Table 6: Rule based data reasoners

tions to the database necessitate a rerun of the reasoning.

6 Summary and Conclusions

We have described a method of translating an OWL-DL ontology into an active database that can be queried and updated independently of the source ontology. In particular, we have implemented type inference for OWL-DL in relational databases, and have produced a prototype implementation that builds such type inference into Postgres databases. This prototype implementation has already been demonstrated to build databases that allow faster execution of queries over the inferred data than any other implementation we are aware of.

The current prototype is relatively crude in its generation of trigger statements, in that it does not attempt to combine multiple triggers on one table into a single trigger and function call. Such optimisation would substantially improve the load times for data into our system.

At the moment we do not correctly handle deletes and updates, since we do not propagate changes to data onto any data that might have been inferred from the data being changed. Although in some cases this would have been a simple task to implement, the complexity of removing inferred values from transitive properties means this is still the subject of investigation.

7 Acknowledgements

The work reported in this report was funded by the Systems Engineering for Autonomous Systems (SEAS) Defence Technology Centre established by the UK Ministry of Defence.

References

[1] Y. al Safadi *et al.* OWL Web Ontology Language Overview, 2004. http://www.w3.org/TR/owl-features/.

[2] Y. al Safadi *et al.* The wine ontology, 2004. www.w3.org/TR/owl-guide/wine.rdf.

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

[4] P. Biron *et al.* XML Schema part 2: Datatypes second edition. http://www.w3.org/TR/xmlschema-2, 2004.

[5] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *International Semantic Web Conference*, pages 54–68, 2002.

[6] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: combining logic programs with description logic. In *WWW*, pages 48–57, 2003.

[7] I. Herman *et al.* Resource Description Framework (RDF), 2001. http://www.w3.org/RDF/.

[8] U. Hustadt and B. Motik. Description logics and disjunctive datalog the story so far. In *Description Logics*, 2005.

[9] A. Kiryakov, D. Ognyanov, and D. Manov. Owlim - a pragmatic semantic repository for OWL. In *WISE Workshops*, pages 182–192, 2005.

[10] J. Lu, L. Ma, L. Zhang, J.-S. Brunner, C. Wang, Y. Pan, and Y. Yu. Sor: A practical system for ontology storage, reasoning and search. In *VLDB*, pages 1402–1405, 2007.

[11] G. Lukácsy and P. Szeredi. Efficient description logic reasoning in Prolog: the DLog system. *Theory and Practice of Logic Programming*, 09(03):343–414, May 2009.

[12] L. Ma, Y. Yang, Z. Qiu, G. T. Xie, Y. Pan, and S. Liu. Towards a complete OWL ontology benchmark. In *ESWC*, pages 125–139, 2006.

[13] G. Meditskos and N. Bassiliades. A rule-based object-oriented OWL reasoner. *IEEE Trans. Knowl. Data Eng.*, 20(3):397–410, 2008.

[14] B. Motik and U. Sattler. A comparison of reasoning techniques for querying large description logic aboxes. In *LPAR*, pages 227–241, 2006.

[15] Z. Pan and J. Heflin. DLDB: Extending relational databases to support semantic web queries. In *PSSS*, 2003.

[16] Z. Pan, X. Zhang, and J. Heflin. DLDB2: A scalable multi-perspective semantic web repository. In *Web Intelligence*, pages 489–495, 2008.

[17] Pellet. http://clarkparsia.com/pellet/.

[18] Protege. http://protege.stanford.edu/.

[19] J. Zhou, L. Ma, Q. Liu, L. Zhang, Y. Yu, and Y. Pan. Minerva: A scalable OWL ontology storage and inference system. In *ASWC*, pages 429–443, 2006.