SQOWL: Type Inference in an RDBMS

P.J. McBrien, N. Rizopoulos, and A.C. Smith

Imperial College London**, 180 Queen's Gate, London, UK

Abstract. In this paper 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**) called $\mathcal{SHOIN}(\mathbf{D})$, we are in effect implementing a method for $\mathcal{SHOIN}(\mathbf{D})$ reasoning in relational databases. Reasoning may 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. We propose a technique to meet this requirement based on the use of triggers, which is the first technique to fully implement $\mathcal{SHOIN}(\mathbf{D})$ as part of normal transaction processing.

1 Introduction

There is currently a growing interest in the development of systems that store and process large amounts of Semantic Web knowledge [6, 14, 17]. A common approach is to represent such knowledge as data in RDF tuples [1], together with rules in OWL-DL [10]. 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** (**UNA**). 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, and make inferences based on the open world assumption.

To illustrate the issues we address in this paper, consider a fragment from the **terminology box** (**TBox**) of the Wine Ontology [12] expressed in DL:

$$\mathsf{Loire} \equiv \mathsf{Wine} \sqcap \mathsf{locatedIn}{:}\{\mathsf{LoireRegion}\} \tag{1}$$

$$\mathsf{WhiteLoire} \equiv \mathsf{Loire} \sqcap \mathsf{WhiteWine} \tag{2}$$

$$\mathsf{WhiteLoire} \sqsubseteq \forall \mathsf{madeFromGrape}.\{\mathsf{CheninBlanc}, \mathsf{PinotBlanc}, \mathsf{SauvignonBlanc}\} \tag{3}$$

$$\top \sqsubseteq \forall \mathsf{locatedIn}^{-}.\mathsf{Region} \tag{4}$$

$$\top \sqsubseteq \forall \mathsf{madeFromGrape}.\mathsf{Wine} \tag{5}$$

$$\top \sqsubseteq \forall \mathsf{madeFromGrape}^{-}.\mathsf{WineGrape} \tag{6}$$

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}.

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

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

Furthermore, each property has a domain and a range which can be restricted. For example, (5) could infer the foreign key `madeFromGrape.domain` $\rightarrow$ `Wine.id` and (6) infer `madeFromGrape.range` $\rightarrow$ `WineGrape.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**)):

$$\mathsf{Loire(SevreEtMaineMuscadet)} \tag{7}$$

$$\mathsf{WhiteWine(SevreEtMaineMuscadet)} \tag{8}$$

$$\mathsf{madeFromGrape(SevreEtMaineMuscadet, PinotBlancGrape)} \tag{9}$$

Using these rules, we have the problems that in the DBMS version of the ontology:

P1 SevreEtMaineMuscadet would not be a member of Wine, despite that being implied by TBox rule (1) from ABox rule (7) and by (5) from (9)
P2 SevreEtMaineMuscadet is not a member of WhiteLoire, despite that being implied by TBox rule (2) from ABox rule (7) and (8) together.
P3 Assertion of the property membership (9) would fail, since we have not previously asserted the PinotBlancGrape as a member of WineGrape.

Performing **classification** using a reasoner on the TBox infers new subclass relationships (*i.e.* in a relational terms, foreign keys), and such classification can partially solve these problems. In particular, classification would infer the following:

$$\mathsf{Loire} \sqsubseteq \mathsf{Wine} \tag{10}$$

$$\mathsf{Loire} \sqsubseteq \mathsf{locatedIn}{:}\{\mathsf{LoireRegion}\} \tag{11}$$

$$\mathsf{WhiteLoire} \sqsubseteq \mathsf{Loire} \tag{12}$$

$$\mathsf{WhiteLoire} \sqsubseteq \mathsf{WhiteWine} \tag{13}$$

which will then allow us to infer additional foreign keys. For example, (10) would imply `Loire.id` $\rightarrow$ `Wine.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. In **type inference** performed by a reasoner on the ABox, data values can be inserted into classes if their presence is logically determined by the presence of data values in other classes. Hence in open world systems with type inference, you are allowed to insert SevreEtMaineMuscadet into Loire provided that data value was either a member of Wine already, or it could be inserted into Wine. Furthermore, type inference would determine that SevreEtMaineMuscadet should be a member of WhiteLoire based on TBox and ABox rules (2), (7) and (8).

In general, when performing type inference over data in an RDBMS, there is a choice between the reasoning being performed by a separate application outside the database, or being performed within the database system. Current implementations of the separate application approach all have the disadvantage that each change to the data requires the external application to reload the data and recompute type inference for all the individuals. Whilst data updates in semantic web applications might not be very intensive, even moderately sized databases will take a considerable time to be loaded and have type inference performed. For example, our tests [9] of the SOR [7] system with the LUBM benchmark [14] containing 350,000 individuals took 16 seconds to load the set of individuals. Hence, each addition to a database of that size would need to be locked for 16 seconds whilst type inference is performed.

It might be argued that it is possible to achieve tighter integration with an external application, such that a total reload is not required each time an insert is made. However, it will always be an inefficient process to keep the data in an external application synchronised with the contents of a DBMS. Hence we investigate in this paper how type inference can be performed within an RDBMS. One previous approach followed in DLDB2 [14] is to use views to compute inferred types, and for each relation have an intentional definition based on rules and an extensional definition with stored values. An alternative approach studied in this paper 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. Although there is to materialised view maintenance [4, 16], to implement OWL-DL we must allow a relation to be both a base table and have view statements to derive certain additional values. Our approach has the disadvantage of additional data storage for the materialised views, and additional time taken to insert data into the database. The prototype of our SQOWL approach is 1000 times faster than DLDB2 at query processing, but 10 times slower than DLDB2 at inserts on the LUBM case study. Since most database applications are query intensive rather than update intensive, there will be a large range of applications that would benefit from out approach.

The SQOWL approach presented in this paper is the first complete (in the sense of supporting all OWL-DL constructs) implementation of type inference for OWL-DL on data held in an RDBMS where the type inference is entirely performed within the RDBMS. Compared to previous reasoners for use on ontologies with large numbers of individuals, our approach has the following advantages (validated by experiments comparing our prototype with other implementations):

- In common with other rule based approaches [13, 6], our approach to type inference is much more efficient than tableaux based reasoners [2], since we do not need to use a process of refutation to infer instances as being members of classes.
- Apart from SOR [7], we are the only rule based approach implementing the full $\mathcal{SHOIN}(\mathbf{D})$ DL [2] of OWL-DL. By fully implement, we mean supporting all of the OWL-DL constructs including oneOf and hasValue restrictions, and providing a type inference complete enough to be able to run all queries in the LUBM [14] and UOBM [8] benchmarks.
- Apart from DLDB2 [14], 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 paper 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. We give a more detailed description of related work in Section 4, and give our summary and conclusions in Section 5.

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 in order to run the benchmark tests.

i. 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 [15], a tableaux based reasoner.
ii. 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, but use triggers to perform type inference rather than use foreign keys to maintain integrity constraints.
iii. 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 (i)–(iii) 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 (i) and (ii) 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 our prototype would be named `cheninblanc_pinotblanc_sauvignonblanc`).

Now we shall introduce the abstract trigger syntax we use in step (iii) 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
  - ignore if the event is to be ignored, which may only be used if the *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:

$\mathbf{when}\ ^{+}\texttt{WhiteLoire}(x)\ \mathbf{if}\ \mathsf{true}\ \mathbf{then}\ \texttt{Loire}(x)$

which states that after the insertion of x into the `WhiteLoire` table, we unconditionally go on to assert the x is a value of `Loire`. This in turn may be implemented by an SQL trigger, the PostgreSQL version being 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 there is one trigger for each table in the RHS, that table being the *event*, the remainder of the RHS in the *condition*, and the tables of the LHS in the *action*. For example, for TBox rule (2), we have two triggers, one for each table in the RHS:

$\mathbf{when}\ ^{+}\texttt{Loire}(x)\ \mathbf{if}\ \texttt{WhiteWine(x)}\ \mathbf{then}\ \texttt{WhiteLoire}(x)$
$\mathbf{when}\ ^{+}\texttt{WhiteWine}(x)\ \mathbf{if}\ \texttt{Loire(x)}\ \mathbf{then}\ \texttt{WhiteLoire}(x)$

3 Translating OWL-DL to ECA rules

In this section we will describe how we translate an OWL-DL KB into the ECA rules introduced in the previous section. Note that these ECA rules may in turn be translated into any specific implementation of SQL triggers that supports both `BEFORE` and `AFTER` triggers on row level updates. The outline of the mapping from our **when if then** ECA rules to SQL is as follows:

$\mathbf{when}\ ^{-}C(x)$	:= `BEFORE INSERT ON` C
$\mathbf{when}\ ^{+}C(x)$	:= `AFTER INSERT ON` C
$\mathbf{if}\ C(x)$	:= `IF EXISTS (SELECT id FROM` C `WHERE id=`x`)`
$\mathbf{if}\ \neg C(x)$	:= `IF NOT EXISTS (SELECT id FROM` C `WHERE id=`x`)`
$\mathbf{then}\ C(x)$	:= `THEN INSERT INTO` C`(id) VALUES(`x`) END IF;`
$\mathbf{then}$ ignore	:= `THEN RETURN NULL END IF;`
$\mathbf{then}$ reject	:= `THEN RAISE EXCEPTION '...' END IF;`

In the section we present the translation of basic OWL-DL classes and properties in Sections 3.1 and 3.2. Then we describe how class descriptions are translated in Sec-

```
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
AFTER INSERT ON WhiteLoire
FOR EACH ROW EXECUTE PROCEDURE
 insert_Loire();
```

(a) WhiteLoire $\sqsubseteq$ Loire

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

CREATE TRIGGER skipinsert
BEFORE INSERT ON Wine
FOR EACH ROW EXECUTE PROCEDURE
 skip_insert_Wine();
```

(b) Allow asserts on Wine

```
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}

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

CREATE TRIGGER propagateTo_Wine
AFTER INSERT ON Loire
FOR EACH ROW EXECUTE PROCEDURE
 insert_Wine();
```

(d) Loire $\sqsubseteq$ Wine

Fig. 1. Some examples of Postgres triggers implementing type inference for DL statements

tion 3.3, and the special case of intersections in class definitions is handled in Section 3.4. Finally we present the translation of restrictions on properties in Section 3.5.

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`(id VARCHAR PRIMARY KEY)`,
$\quad$ **when** $^{-}C(x)$ **if** $C(x)$ **then** ignore

with the semantics that any class C found in OWL-DL causes two additions to the relational schema. The first addition is a table to hold the known instances of this class, and the second addition is an SQL trigger on table C to ignore any insertions of a tuple value x where x already exists in C. Note that the trigger is fired before x is actually inserted into C, hence preventing spurious duplicate key error messages from the RDBMS when a fact x that has already been inserted, is attempted to be inserted again. To illustrate how a class is implemented in the RDBMS, the declaration of class Loire in TBox rule (1) produces:

`CREATE TABLE Loire(id VARCHAR PRIMARY KEY)`
when $^{-}$Loire(x) **if** Loire(x) **then** ignore

where the translation of the ECA rule to a Postgres trigger is illustrated in Figure 1(b). 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 production rule is:

individual : $C(a) \leadsto$ `INSERT INTO` C `VALUES` (a)

3.2 OWL-DL properties

An OWL-DL **property** defines a binary predicate $P(D,R)$, where the domain D is always an OWL-DL class, and the range R varies according to which of two types P belongs to. A **datatype property** has a range which is a datatype, normally defined as an RDF literal or XML Schema datatype [3]. An **object property** has a range which is an OWL-DL class.

Hence, there is a rough analogy between properties and subclass relationships, in that membership of a property implies membership in the domain class, and if it exists, the range class. Thus the implementation of properties is as a two-column table, with triggers to update any classes that the property references, giving the following rules:

datatypeProperty: $P(D,R) \leadsto$

`CREATE TABLE` P `(domain VARCHAR, range` $sqlType(R)$`)`

when $^{-}P(x,y)$ **if** $P(x,y)$ **then** ignore **if** true **then** $D(x)$

objectProperty: $P(D,R) \leadsto$

`CREATE TABLE` P `(domain VARCHAR, range VARCHAR)`

when $^{-}P(x,y)$ **if** $P(x,y)$ **then** ignore **if** true **then** $D(x), R(x)$

Note that the function $sqlType(R)$ returns the SQL data type corresponding to the OWL-DL datatype R. Applying the above to wine ontology rules $\top \sqsubseteq \forall \mathsf{hasFlavor}.\mathsf{Wine}$ and $\top \sqsubseteq \forall \mathsf{hasFlavor}^{-}.\mathsf{WineFlavor}$ will lead to a table to represent the object property being created:

`CREATE TABLE hasFlavor(domain VARCHAR, range VARCHAR).`

plus a trigger that causes an update to `hasFlavor` to be propagated to both `Wine` and `WineFlavor`. Instances of a property have the obvious mapping to insert statements on the corresponding property table:

propertyInstance: $P(x,y) \leadsto$ `INSERT INTO TABLE` P `VALUES` (x,y)

With these definitions, we have a solution to problem **P3** in the introduction, since assertions of membership such as in (9) will cause any 'missing' class memberships, such as PinotBlancGrape being a member of WineGrape to be created.

3.3 OWL-DL Class Descriptions

In OWL-DL, if C, D each denote a class, P denotes a property, and $a_1,\ldots,a_n$ individuals, then a class description takes the form:

$\langle$*class-des*$\rangle$ ::= $\langle$*class-expression*$\rangle$ | $\langle$*property-restriction*$\rangle$ |
$\langle$*class-des*$\rangle \sqcap \langle$*class-des*$\rangle$ | $\langle$*class-des*$\rangle \sqcup \langle$*class-des*$\rangle$

$\langle$*class-expression*$\rangle = D \mid \neg D \mid \{a_1,\ldots,a_n\}$

$\langle$*property-restriction*$\rangle = \forall P.D \mid \exists P.D \mid \exists P{:}a \mid =nP \mid >nP \mid <nP$

If E, E_1, E_2 are class descriptions, then a **partial** class definition $C \sqsubseteq E$ means that all instances of the class C must be instances of E, but not *vice versa*. A **complete** class description $C \equiv E$ means that the instances of C and E must always be the same. However, since $C \equiv E \quad \rightarrow \quad C \sqsubseteq E, E \sqsubseteq C$ we only need consider partial definitions of the form $C \sqsubseteq E$ or $E \sqsubseteq C$. Furthermore, for union $\sqcup$ and intersection $\sqcap$ we can state $C \sqsubseteq E_1 \sqcap E_2 \quad \rightarrow \quad C \sqsubseteq E_1, C \sqsubseteq E_2$ and $E_1 \sqcup E_2 \sqsubseteq C \quad \rightarrow \quad E_1 \sqsubseteq C, E_2 \sqsubseteq C$, thus not needing to consider further those uses of union and intersection. For $C \sqsubseteq E_1 \sqcup E_2$ there are no inferences possible, since values holding for C infer a value holding for E_1 or E_2, but we do not know which of the two expressions it holds for. Thus with the exception of $E_1 \sqcap E_2 \sqsubseteq C$ (which we will deal with in Section 3.4), all we need to consider are cases where $C \sqsubseteq E$ or $E \sqsubseteq C$, and E is a class expression or a property restriction.

The simplest class definition is a subclass relationship, which may be implemented as an ECA rule that ensures that instances of the table implementing the subclass are also instances of the table implementing the superclass:

subClassOf: $C \sqsubseteq D \rightsquigarrow$ **when** $^{+}C(x)$ **if** true **then** $D(x)$

An example of the Postgres Trigger generated by the above ECA rule is shown in Figure 1(d) for the TBox rule (10). Combined with the implementation of a class and its associated ECA rule, we have an implementation that is able to deal with problem **P1** in the introduction. Specifically, when ABox rule (7) is translated to

`INSERT INTO Loire VALUES (ServeEtMaineMuscadet)`

and the trigger in Figure 1(d) will cause an insert into `Wine` to be attempted. If the individual `ServeEtMaineMuscadet` had already been inserted into `Wine` this insert will be ignored by the trigger in Figure 1(b), otherwise the insert proceeds to correctly add a new instance to `Wine`.

A class D might be declared to be a subclass of 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. The production rule is:

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

In the case where the complement of C is a subclass of D, then an insertion of x on C means that x is not a member of the complement of C, which implies that x is not a member of D. Thus, the production rule is:

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

The oneOf construct enables a class C to be defined exactly by enumerating its instances, $\{a_1, \ldots, a_n\}$. In our system (where we make the UNA) the enumeration class corresponds to a anonymous table that contains only the instances $a_1, \ldots, a_n$. This can be created by inserting values into the table, and then creating a ECA rule that fails when any further inserts are performed.

enumeration: $\{a_1, \ldots, a_n\} \rightsquigarrow$ `INSERT INTO` D `VALUES (`a_1`), ..., (`a_n`)`
when $^{-}D(x)$ **if** $\neg D(x)$ **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 IN cheninblanc_pinotblanc_sauvignonblanc VALUES
  (CheninBlanc), (PinotBlanc), (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).

The allValuesFrom $\forall P.D$ restriction on property P defines a set of individuals x where $\langle x, y\rangle$ appears in P and y in D. Note that the allValuesFrom restriction can be satisfied trivially if there are no tuples for property P. In the case of $C \sqsubseteq \forall P.D$, then each individual x of C that also appears in a tuple $\langle x, y\rangle$ of P will infer that y is an individual of D. This restriction translates into two triggers based on the order tuples are inserted in tables C and P. The first trigger is executed after an insertion of x in table C. If x is not associated with any value in table P, then the restriction is satisfied. If a tuple $\langle x, y\rangle$ already exists in table P such that y is not an instance of D, then the trigger inserts y in D. The second trigger is executed after an insertion of tuple $\langle x, y\rangle$ in P. If x already exists in C then the trigger inserts y in D.

allValuesFrom: $C \sqsubseteq \forall P.D \rightsquigarrow$ **when** $^{+}C(x)$ **if** $P(x, y)$ **then** $D(y)$
when $^{+}P(x, y)$ **if** $C(x)$ **then** $D(y)$

For example, based on TBox rule (6), we know that for each individual x which has a tuple $\langle x, y\rangle$ in madeFromGrape, then y is a WineGrape. Thus, based on ABox rule (9) we can infer that PinotBlancGrape is a WineGrape. As a trigger, the C table corresponds to $\top$ in DL, and hence is **true** in the trigger, and we simply have a trigger:

when `madeFromGrape`(x, y) **if** true **then** `WineGrape`(y)

We cannot infer anything from $\forall P.D \sqsubseteq C$, since even if $\forall P.D$ holds for some $\langle x, y\rangle$ in P and y in D, there might (by open world semantics) be some $\langle x, y'\rangle$ in P that is correct but not yet inserted into the database. If the y' was not a correct value of D, then x should not be in $\forall P.D$.

someValuesFrom $\exists P.D$ holds for each x in where there is at least one tuple $\langle x, y\rangle$ of P such that y a member of class D. For the implementation of $C \sqsubseteq \exists P.D$ we could define that for each insertion $C(x)$ we insert a tuple $\langle x, \mathsf{null}\rangle$ on P. However, this is not necessary since the existence of the tuple $\langle x, \mathsf{null}\rangle$ cannot be used for any kind of inference. Thus the rule's body is empty $-$.

someValuesFrom: $C \sqsubseteq \exists P.D \rightsquigarrow -$

For example, based on the rules Wine(TaylorPort) and Wine $\sqsubseteq$ $\exists$locatedIn.Region 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.

In the case of $\exists P.D \sqsubseteq C$, each individual x associated in P with a y, which is a member of D, becomes a member of C. The trigger is executed after an insertion of tuple $\langle x, y\rangle$ in P.

someValuesFrom: $\exists P.D \sqsubseteq C \rightsquigarrow$ **when** $^{+}P(x, y)$ **if** $D(y)$ **then** $C(x)$
when $^{+}D(y)$ **if** $P(x, y)$ **then** $C(x)$

The OWL constructs cardinality, minCardinality, maxCardinality restrict the number of tuples $\langle x, y\rangle$ in P an individual x of C can have. As in the previous restriction we could create a rule that adds tuples $\langle x, \mathsf{null}\rangle$ so that the cardinality restriction is satisfied. However, these tuples cannot be used for inference therefore the body of the rule is empty:

cardinality: $C \sqsubseteq {=}nP \rightsquigarrow -$

maxCardinality: $C \sqsubseteq {<}nP \rightsquigarrow -$

minCardinality: $C \sqsubseteq {>}nP \rightsquigarrow -$

In the case of ${=}nP \sqsubseteq C$, the restriction specifies that for any x which has n tuples $\langle x, y_1\rangle, \ldots, \langle x, y_n\rangle$ in P, then x is a member of C. However, due to the open world assumption of the DL, we cannot be certain about the cardinality of a property (except if the property is functional). For example, even if n tuples exist for x in P, this does not exclude the fact that there might be other tuples for x which are not yet known. Thus, the rule's body for this restriction is empty. Similarly for the case of ${<}nP \sqsubseteq C$.

cardinality: ${=}nP \sqsubseteq C \rightsquigarrow -$

maxCardinality: ${<}nP \sqsubseteq C \rightsquigarrow -$

However, in the case of the minCardinality construct ${>}nP \sqsubseteq C$, if x is associated with n tuples $\langle x, y_1\rangle, \ldots, \langle x, y_n\rangle$ in P, then the restriction is satisfied and x becomes a member of C. The trigger implementing this restriction is illustrated below and it uses function $count(P(x, \_))$ which returns the number of tuples $\langle x, y\rangle$ in P for individual x.

minCardinality: ${>}nP \sqsubseteq C \rightsquigarrow$ **when** $^{+}P(x, y)$ **if** $count(P(x, \_)) > n$ **then** $C(x)$

The DL hasValue construct $C \sqsubseteq \exists P{:}a$ specifies that each individual x of C has a tuple $\langle x, a\rangle$ in P. This restriction translates into a trigger which is executed after an insertion of instance x into C. The trigger inserts the tuple $\langle x, a\rangle$ into table P .

hasValue: $C \sqsubseteq \exists P{:}a \rightsquigarrow$ **when** $^{+}C(x)$ **if** true **then** $P(x, a)$

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`. In the case of $\exists P{:}a \sqsubseteq C$, for each x with $\langle x, a\rangle$ a tuple of P, x is inserted into C.

hasValue: $\exists P{:}a \sqsubseteq C \rightsquigarrow$ **when** $^{+}P(x, y)$ **if** $P(x, a)$ **then** $C(x)$

3.4 Inferences from intersections

When an intersection appears on the left of a subclass relationship there is an additional inference that can be performed. For example, from TBox rule (2), we have

Loire $\sqcap$ WhiteWine $\sqsubseteq$ WhiteLoire

and hence if there is an individual x inserted which is both a member of Loire and a member of WhiteWine, then it can be inferred to be a member of WhiteLoire. To achieve this we would need a trigger which is executed after an instance x is inserted in table `Loire` or `WhiteWine`, and if the value is present in the other of those two tables, inserts x into `WhiteLoire`:

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

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

In general, if $E_1 \sqcap \ldots \sqcap E_n \sqsubseteq C$, then we require a trigger on each E_i that will check to see if $E_1 \sqcap \ldots \sqcap E_n$ now holds. Thus the implementation of intersection is:

intersection: $E_1 \sqcap \ldots \sqcap E_n \sqsubseteq C \rightsquigarrow$

$\forall_{i=1}^{n}$**when** $^{+}trigger(E_i)$ **if** $holds(E_1 \sqcap \ldots \sqcap E_n)$ **then** $C(x)$

where the $trigger$ function identifies the tables to trigger on as follows:

$trigger(> nP) := P(x, y) \qquad trigger(E_i) := E_i(x)$
$trigger(\exists P.D) := P(x, y) \qquad trigger(P{:}\{a\}) := P(x, y)$
$trigger(\forall P.D) := P(x, y)$

and the *holds* function maps the OWL-DL intersection into predicate logic (and hence SQL) as follows:

$holds(D \sqcap E) := holds(D), holds(E) \qquad holds(D) := D(x)$
$holds(>n\ P) := count(P(x, \_)) > n \qquad holds(P{:}\{a\}) := P(x, a)$
$holds(\exists P.D) := P(x, y), D(y) \qquad holds(\forall P.D) := \mathsf{false}$

Note that we say the $\forall P.D$ is assumed to be false, simply because with open world semantics, there is no simple query that can determine if it is true. As an example of using the above rules, if we apply them to the TBox rule (2), we have that $E_1 \equiv \mathsf{Loire}$ and $E_2 \equiv \mathsf{WhiteWine}$, and that expands out to:

when $^{+}trigger(\mathsf{Loire})$ **if** $holds(\mathsf{Loire} \sqcap \mathsf{WhiteWine})$ **then** `WhiteLoire`(x)
when $^{+}trigger(\mathsf{WhiteWine})$ **if** $holds(\mathsf{Loire} \sqcap \mathsf{WhiteWine})$ **then** `WhiteLoire`(x)

and expanding the *trigger* and *holds* function gives:

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

If we remove the redundant check on `Loire` in the condition of the first rule, and the redundant check on `WhiteWine` in the condition of the second rule, we have the two ECA rules we talked about in the beginning of this section, and have provided a solution to problem **P2** in the introduction.

3.5 Restrictions on and between OWL-DL Properties

Properties can be **functional** and/or **inverse functional**. A functional property $P(D, R)$ can have only one tuple $\langle x, y\rangle$ for each x in D, and an inverse functional can have only one tuple $\langle x, y\rangle$ for each y in R. We translate these restrictions into SQL as key/unique constraints:

FunctionalProperty: $P \rightsquigarrow$ `ALTER TABLE` P `ADD PRIMARY KEY (domain)`
InverseFunctionalProperty: $P \rightsquigarrow$ `ALTER TABLE` P `ADD CONSTRAINT UNIQUE (range)`

In the Wine ontology there is a functional property definition $\top \sqsubseteq \leqslant 1\ \mathsf{hasFlavor}$ which adds 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. Note that it would be possible in an implementation to normalise a functional property table (such as `hasFlavor`) with the class of its domain (such as `Wine`) to make the range a column in the class table (*i.e.* to have a column `grape` in `Wine`, instead of the separate `hasFlavor` property table). All triggers that were on the property table would instead be on the relevant columns of the class table. However this would make our presentation more complex, and so we do not use such an optimisation in this paper.

Properties can also be **transitive** and/or **symmetric**. For example, the locatedIn property in the Wine ontology is transitive, and hence if we know:

locatedIn(ChateauChevalBlancStEmilion, BordeauxRegion)
locatedIn(BordeauxRegion, FrenchRegion)

then we can infer:

locatedIn(ChateauChevalBlancStEmilion, FrenchRegion)

The production rule for a transitive property P needs to define a trigger to be executed after each insert of tuple $\langle x, y\rangle$ in P. The rule will insert for each $\langle y, z\rangle$ existing in P the tuple $\langle x, z\rangle$ and for each $\langle z, x\rangle$ in P the tuple $\langle z, y\rangle$ 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: $P \in \mathbf{P}_+ \rightsquigarrow \mathbf{when}\ {}^+P(x, y)\ \mathbf{if}\ \mathsf{true}\ \mathbf{then}$
$foreach(z, P(y, z), P(x, z)), foreach(z, P(z, x), P(z, y))$

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

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

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

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

A property P might be declared as the inverse of another property Q. This declaration asserts that for each tuple $\langle x, y\rangle$ in P, the inverse tuple $\langle y, x\rangle$ 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: $P \equiv Q^- \rightsquigarrow \mathbf{when}\ {}^+P(x, y)\ \mathbf{if}\ \neg Q(y, x)\ \mathbf{then}\ Q(y, x)$

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 $\langle x, y\rangle$ in P the trigger inserts the tuple on table Q, and vice versa. The production rule is:

equivalentProperty: $P \equiv Q \rightsquigarrow \mathbf{when}\ {}^+P(x, y)\ \mathbf{if}\ \mathsf{true}\ \mathbf{then}\ Q(x, y)$
$\mathbf{when}\ {}^+Q(x, y)\ \mathbf{if}\ \mathsf{true}\ \mathbf{then}\ P(x, y)$

4 Related Work

DL reasoners come in a number of forms [2]. The most common type are Tableaux based reasoners like Racer, FacT++ and Pellet. These 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 numbers of individuals, due to the use of a refutation procedure rather than a query answering algorithm [5].

Rule based reasoners provide an alternative to the tableaux based approach that is more promising for handling large datasets. O-DEVICE [11] translates OWL rules into an in-memory representation, and can process all of. It can process all of OWL-

DL except 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 [6] is similar in both features and problems, but supports a smaller subset of OWL-DL than O-DEVICE.

KAON2 [13] does reasoning by means of theorem proving. The TBox is translated into first-order clauses, which are executed on a disjunctive Datalog engine to compute the inferred closure. KAON2 has fast load and query times, but is unable to handle **nominals** (*i.e.* hasValue and oneOf, *i.e.* misses the O in $\mathcal{SHOIN}(\mathbf{D})$).

DLDB2 [14] and SOR [17] (previously called Minerva) are most similar to SQOWL, since they use an RDBMS as their rule engine. DLDB2 stores the rules inside the database as non-materialised views. Tables are created for each atomic property and class, populated with 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 into 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. DLDB2 is not able to perform type inference based on allValuesFrom.

SOR [17] 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 to materialise the inference results. This makes query processing faster. Because the rules are kept outside the database, any additions to the database necessitate a rerun of the reasoning.

5 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. Our approach gives a complete implementation of OWL-DL in a relational database, assuming that we make the UNA. As such, we have no need to handle OWL-DL constructs differentFrom, AllDifferent, or sameAs since they are concerned with issues where the UNA does not hold.

Running the LUBM [14] and UOBM [8] benchmarks shows we are between two and 1000 times faster at query answering than other DBMS based approaches [9]. Only the DLDB2 approach has the same advantage of performing all its type inference within the DBMS, and DLDB2 is at least 30 times slower than SQOWL in query answering in these benchmarks.

The current prototype is 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. Furthermore, it does not make the obvious optimisation that all functional properties of a class can be stored as a single table, which would reduce further the number of triggers and reduce the number of joins required in query processing. A more substantial addition would be to handle deletes and updates, since we would then need to consider how a fact might be derived from more than one rule.

We have shown that the SQOWL approach offers for a certain class of semantic web application, a method of efficiently storing data, and performing type inference. We also believe that our work opens up the possibility of using ontologies expressed in OWL-DL to enhance database schemas with type inference capabilities, and will explore this theme of 'knowledge reasoning' in RDBMS applications in future work.

References

1. Resource Description Framework (RDF), 2001. http://www.w3.org/RDF/.
2. 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.
3. P. Biron and A. Malhotra. XML Schema part 2: Datatypes second edition. http://www.w3.org/TR/xmlschema-2, 2004.
4. S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. VLDB*, pages 577–589, 1991.
5. U. Hustadt and B. Motik. Description logics and disjunctive datalog the story so far. In *Description Logics*, 2005.
6. A. Kiryakov, D. Ognyanov, and D. Manov. Owlim - a pragmatic semantic repository for OWL. In *WISE Workshops*, pages 182–192, 2005.
7. 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.
8. 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.
9. P. McBrien, N. Rizopoulos, and A. Smith. SQOWL: Performing OWL-DL type inference in SQL. Technical report, AutoMed Technical Report 37, 2009.
10. D. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview, 2004. http://www.w3.org/TR/owl-features/.
11. G. Meditskos and N. Bassiliades. A rule-based object-oriented OWL reasoner. *IEEE Trans. Knowl. Data Eng.*, 20(3):397–410, 2008.
12. M.K.Smith, C. Welty, and D. McGuinness. OWL Web Ontology Language Guide, 2004. www.w3.org/TR/owl-guide/.
13. B. Motik and U. Sattler. A comparison of reasoning techniques for querying large description logic aboxes. In *LPAR*, pages 227–241, 2006.
14. Z. Pan, X. Zhang, and J. Heflin. DLDB2: A scalable multi-perspective semantic web repository. In *Web Intelligence*, pages 489–495, 2008.
15. Pellet. http://clarkparsia.com/pellet/.
16. T. Urpí and A. Olivé. A method for change computation in deductive databases. In *Proc. VLDB*, pages 225–237, 1992.
17. 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.