# 3XL

*Supporting Efficient Operations on Very Large OWL Lite Triple-stores*

Liu, Xiufeng; Thomsen, Christian; Pedersen, Torben Bach

*Published in:*
Information Systems

*DOI (link to publication from Publisher):*
10.1016/j.is.2010.12.001

*Publication date:*
2011

*Document Version*
Early version, also known as pre-print

Information Systems 36 (2011) 765–781

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

# 3XL: Supporting efficient operations on very large OWL Lite triple-stores

Xiufeng Liu *, Christian Thomsen, Torben Bach Pedersen

*Department of Computer Science, Aalborg University, Selma Lagerlöfs Vej 300, DK-9220 Aalborg Ø, Denmark*

ARTICLE INFO

*Article history:*
Received 11 May 2010
Received in revised form
28 October 2010
Accepted 3 December 2010
Recommended by: F. Korn
Available online 10 December 2010

*Keywords:*
Triple-store
OWL Lite
Bulk operations
Specialized schema

ABSTRACT

An increasing number of (semantic) web applications store a very large number of (subject, predicate, object) triples in specialized storage engines called triple-stores. Often, triple-stores are used mainly as plain data stores, i.e., for inserting and retrieving large amounts of triples, but not using more advanced features such as logical inference, etc. However, current triple-stores are not optimized for such bulk operations and/or do not support OWL Lite. Further, triple-stores can be inflexible when the data has to be integrated with other kinds of data in non-triple form, e.g., standard relational data. This paper presents 3XL, a triple-store that efficiently supports operations on very large amounts of OWL Lite triples. 3XL also provides the user with high flexibility as it stores data in an object-relational database in a schema that is easy to use and understand. It is, thus, easy to integrate 3XL data with data from other sources. The distinguishing features of 3XL include (a) flexibility as the data is stored in a database, allowing easy integration with other data, and can be queried by means of both triple queries and SQL, (b) using a specialized data-dependent schema (with intelligent partitioning) which is intuitive and efficient to use, (c) using object-relational DBMS features such as inheritance, (d) efficient loading through extensive use of bulk loading and caching, and (e) efficient triple query operations, especially in the important case when the subject and/or predicate is known. Extensive experiments with a PostgreSQL-based implementation show that 3XL performs very well for such operations and that the performance is comparable to state-of-the-art triple-stores.



## 1. Introduction

The increasing popularity of (semantic) web applications means that very large amounts of semantic web data, e.g., from ontologies, need to be managed. Most semantic web data is somehow based on the *Resource Description Framework* (RDF) [1], a family of *World Wide Web Consortium* (W3C) specifications for conceptual description/modeling of web resource information. Recently, the *Web Ontology Language* (OWL), a semantic markup language recommended by W3C for publishing and sharing ontologies on the WWW has gained popularity [2]. OWL is layered on top of RDF. Even the least expressive of the three OWL layers (OWL Lite) offers class hierarchies and constraints features, and is very useful for thesauri and other taxonomies. OWL (and RDF) data takes the form of (subject, predicate, object) triples. These triples are typically stored in specialized storage engines called *triple-stores*.

Our initial motivation for this work was the European Internet Accessibility Observatory (EIAO) project [3] where tens of millions of triples describing test results about the accessibility of web pages for people with various kinds of disabilities, e.g., blind people using a screen reader, were generated in the W3C standard EARL RDF language.

* Corresponding author.
*E-mail addresses:* xiliu@cs.aau.dk (X. Liu), chr@cs.aau.dk (C. Thomsen), tbp@cs.aau.dk (T.B. Pedersen).

0306-4379/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2010.12.001

Here, and in other projects, we have seen that the triple-stores are used mainly as specialized *bulk data stores*, i.e., for inserting and retrieving large amounts of triples (bulk operations). More advanced features such as logical inference, etc., are often not used. Additionally, for the basic storage of data about OWL instances, we found that even a subset of the OWL Lite features was enough, namely classes, subclasses, object properties, data properties, domains, ranges, restrictions, onProperty, and maxCardinality. A well-known example of such data is the data generated by the data generator for the de facto industry standard OWL data management benchmark Lehigh University Benchmark (LUBM) [4].

Similarly to many other projects, the EIAO project involved later integration of the collected EARL RDF data with other non-RDF data when a data warehouse (DW) was built to enable easy analysis of the accessibility results. To integrate data from triples with other kinds of data from relational databases, flat files, XML, etc. can be difficult. In the EIAO case, an extract-transform-load (ETL) application was hand-coded to execute triple queries, interpret triple results and do the many needed transformations to integrate the data into a relational DW. Thus, it was a design criteria for 3XL to allows easy integration with non-triple data.

In this paper, we present the 3XL triple-store that, unlike most current OWL Lite triple-stores, is specifically designed to support bulk data management operations (load and retrieval) on very large OWL Lite triple-stores and provide the user with flexibility in retrieving the data. The "3" refers to triples, and the "XL" part to "eXtra Large" and "fLeXible" ("3XL" is also the largest standard t-shirt size in Europe). 3XL's approach has a number of unique characteristics. First, 3XL stores data in a relational database meaning that the user has flexibility as queries can be expressed either in triples or in SQL. Second, for the database schema, 3XL uses a *specialized data-dependent schema* derived from the OWL ontology for the data to store, meaning that the schema is easy to navigate and understand and that an "intelligent partitioning" of data is performed. Third, 3XL uses advanced object-relational features of the underlying ORDBMS, namely table inheritance and the possibility to have arrays as in-lined attribute values. Fourth, 3XL makes extensive use of a number of *bulk-loading* techniques and caching to speed up bulk insertions significantly. Finally, 3XL is specifically designed to support efficient bulk retrieval for queries where the subject and/or the predicate is known, as such queries are the most important for most bulk data management applications. 3XL is implemented on top of the PostgreSQL ORDBMS.

Extensive performance experiments with both real-world and synthetic data show that 3XL has load and query performance comparable to the best (file-based) triple-stores, and that 3XL outperforms other DBMS-based triple-stores. We believe this positions 3XL in a unique spot: performance comparable to the best file-based triple-stores combined with the high degree of flexibility in accessing and integrating non-triple data offered by a DBMS-based triple-store.

The rest of the paper is structured as follows. Section 2 introduces the 3XL system in general, explains how to generate a 3XL database schema from an input OWL ontology, and finally describes triple addition and queries. Section 3 presents experiments. Section 4 presents related work. The last section concludes and provides ideas for future work.

## 2. The 3XL system

### 2.1. Overview

First, we informally describe the general idea about generating a specialized database schema for an OWL ontology in PostgreSQL. The descriptions give an intuition about how 3XL works before this is described in detail.

To build the database to store the data in, an OWL ontology is read. This ontology should define all classes, their parent–child relationships and their properties (including domains and ranges). In the database, a *class table* is created for each class. The class table for the class *C* directly inherits from any class tables for the parent classes of *C*. This means that if the class table for *C*'s parent *P* has the attributes *a*,*b*,*c* then the class table for *C* has at least the attributes *a*,*b*,*c*.

Two attributes are needed for each instance of any class: an ID and a URI. To have these available in all tables, all class tables – directly or indirectly – inherit from a single root class table that represents the OWL class `owl:Thing` that all other OWL classes inherit from. The class table for `owl:Thing` has the columns ID and URI. All the ID values are for convenience unique integers drawn from the same database sequence (this is explained later).

If the class *C* has a `DataProperty` *d* with `maxCardinality` 1, the table for *C* has a column *d* with a proper datatype. For a *multiproperty* without a `maxCardinality`,[1] there is a special *multiproperty table*. This multiproperty table has a column that holds the attribute values and a column that holds the IDs for the instances the property values apply to. The ID attribute acts like a foreign key, but it is not declared (this is explained below). We here denote this as a *loose foreign key*. Note that a multiproperty table does not inherit from the class table for `owl:Thing` since multiproperty tables are not intended to represent instances, but only *values* for instances. A multiproperty has only one multiproperty table for a class *C* and not one for each subclass of *C*.

Instead of using multiproperty tables, the class tables can have columns that hold *arrays*. In this way it is possible to represent several property values for an instance in the single row that represents the instance in question.

An `owl:ObjectProperty` is handled similarly to how an `owl:DataProperty` is handled. If the object property has `owl:maxCardinality` 1, a column for the property is created in the appropriate class table. This column holds IDs for the referenced objects. If the property is a multiproperty, the value column in the multiproperty table holds ID values.

**Example 1.** We now introduce the running example used in the rest of the paper. To save space we do not use URIs but intuitive names for classes and properties.

Assume that there are three classes: Document, HTMLVersion, and HTMLDocument, where HTMLDocument is a subclass of Document. Document has the properties title and keyword. The property keyword is the only multiproperty in this example. HTMLVersion has the properties version and approvalDate. Apart

[1] OWL Lite only allows `maxCardinality` to be 0, 1, or unspecified.

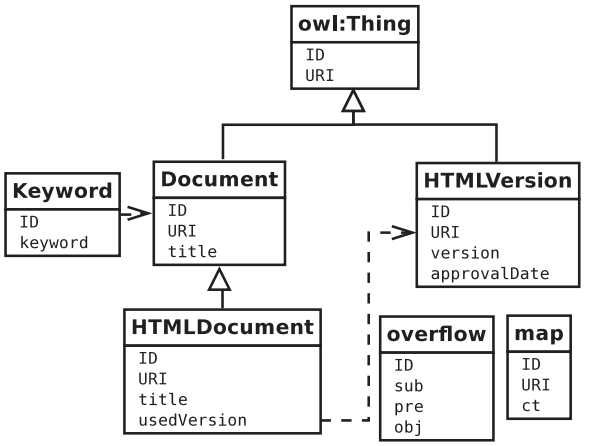


**Fig. 1.** A database schema generated by 3XL.

from the inherited properties, HTMLDocument has the property usedVersion. The property usedVersion is an `owl:ObjectProperty` with `owl:range` HTMLVersion. The remaining properties are all of kind `owl:DataProperty`.

This results in the database schema drawn in Fig. 1. Inheritance is shown with arrows as in UML. A loose foreign key is shown as a dotted arrow. For now, ignore the `map` and `overflow` tables which are explained later.

Note how easy the schema from Example 1 is to use in SQL queries. This makes it easy to integrate the data with other data. Further, the user can exploit all the advanced functionality that the underlying DBMS (PostgreSQL) provides. The user can, however, also choose to query the data by means of (single or "chained") triples as will be explained later.

When triples are being inserted, 3XL has to find out which class the subject belongs to. This decides which class table to insert the data into. If the property name of the triple is unique among classes, it is easy to decide. Otherwise, 3XL tries to deduce the *most specific class* that the instance for sure belongs to. The subject may, however, be an instance of a class that is not described by the ontology used for the schema generation. In that case, the triple is placed in the `overflow` table.

To be efficient, 3XL does not insert data from a triple into the underlying database as soon the triple is added. Instead, the data is held in a buffer until larger amounts of data can be inserted quickly into the underlying database using bulk load mechanisms. To keep data in a buffer for a while also has the advantage that the type detection described above can make a more precise guess. It is a requirement in OWL Lite that there is a triple giving the `rdf:type` for each individual. Thus, a triple revealing the type should appear sooner or later and has often appeared when the actual insertion into the database takes place.

In Example 1 it may, however, happen that an instance *i* of the class Document that has been written to the class table for Document later turns out to actually be an instance of the class HTMLDocument. In that case, it is easy to move the row representing *i* from the class table for Document to the class table for HTMLDocument. Here it is convenient only to have one multiproperty table for keyword since no rows have to moved from the multiproperty table. This also shows why the foreign key from multiproperty tables has to be loose: It is unknown which class table the referenced ID value is located in. However, when querying for a specific ID value for an instance of Document in Example 1, it is enough to use the SQL expression `SELECT ID FROM Document WHERE ...`. PostgreSQL then automatically also looks in descendant tables. This also shows why all IDs should be unique across tables and therefore are drawn from the same sequence.

A drawback of the approach where a row representing an instance is moved from one class table *T* to a class table for a subclass *S*, is that the subclass may put a `maxCardinality 1` restriction on a property *p* that in the superclass is a multiproperty. In this case, the multiproperty table for *p* is not needed to represent data for *S* instances. It is then possible to let *p* be represented by a column in *S* and not by the multiproperty table that has a loose foreign key to *T*. However, for simplicity we keep using the multiproperty table for *p* if it already exists and do not add an extra column for *p* to the class table for *S*.

When the triple-store is queried, it is done by issuing one or several combined (subject, predicate, object) triples, called *point-wise queries* and *composite queries*, respectively (in addition, the user has the possibility to use SQL queries as the data is stored in a relational database). If a property is given in a point-wise query, this can reveal which class table(s) to look into. If only a subject or object is given, it is possible to look up the URI in the `owl:Thing` class table. This is, however, potentially very expensive, so 3XL, in addition to the previously mentioned class tables, also has a `map` table that maps from a URI to the class table that holds the instance with that URI. For each query triple, the `overflow` table is also searched by 3XL. In comparison to point-wise queries, composite queries are more expressive, as they are conjunctive combinations of several point-wise query triples.

In summary, the idea is to have the data spread over many tables (with many columns). It is fast to find data when the table to look in can be identified easily ("intelligent partitioning"). The tables also have very good potential for being indexed. Indexes may be added on attributes that are often used in queries. We are now ready to give a detailed description of how the specialized database schema of 3XL is generated. After that, we describe how additions to the triple-store are handled. This is followed by a description of how queries are handled.

### 2.2. Schema generation

In the following, we describe the handling of the supported OWL constructs when the specialized database schema is generated. To generate the database schema, 3XL reads an ontology and builds a model of the classes including their properties and subclass relationships. Based on the built model, SQL DDL statements to create tables are generated and executed. Note that this SQL is not conforming to the SQL standard since it uses PostgreSQL's object-oriented extensions (see more below). In the following, we focus on the resulting schema.

Note that a database schema generated by 3XL always has the table `map(ID, URI, ct)`. As explained later, this table is used to make it fast to find the table that represents a given instance and the ID of the instance. The table

`overflow(ID, sub, pre, obj)` is also created in each generated schema. This table holds triples that do not fit in the other tables.

We are now ready to describe how the constructs of OWL Lite are handled in the generation of a specialized database schema. We assume that a database schema $D$ is being generated for the OWL ontology $O$.

**owl:Class:** An `owl:Class` in $O$ results in a table, called a *class table*, in $D$. In the following, we denote by $\mathcal{C}_X$ the class table in $D$ for the class $X$ in $O$. $\mathcal{C}_X$ is used such that for each instance of $X$ that is not an instance of a subclass of $X$ and for which data must be stored in the triple-store, there is exactly one row in $\mathcal{C}_X$. Each represented instance has a URI and is given a unique ID by 3XL.

The special class table $\mathcal{C}_{\texttt{owl:Thing}}$ for `owl:Thing` is always created in $D$. This special class table does not inherit from any other table and has two columns named ID (of type INTEGER) and URI (of type VARCHAR). Any other class table created in $D$ always inherits from one or more other class tables (see below) and always inherits – directly or indirectly – from $\mathcal{C}_{\texttt{owl:Thing}}$. This implies that the columns ID and URI are available in each class table.

For other class tables than $\mathcal{C}_{\texttt{owl:Thing}}$, other columns may also be present: A class table for a class that is in the `rdfs:domain` of some property $P$ and is a subclass of a restriction saying the `owl:maxCardinality` of the property is 1, also has a column for $P$. This column is only explicitly declared in the class table for the most general class that is the domain of the property. But class tables inheriting from that class table automatically also have the column. For an example of this, refer to Example 1 where a column for title is declared in the class table for Document.

**rdfs:subClassOf:** For classes $X$ and $Y$ in $O$ where $Y$ is a subclass of $X$ (i.e., the triple ($Y$, `rdfs:subClassOf`, $X$) exists in $O$), there exist class tables $\mathcal{C}_X$ and $\mathcal{C}_Y$ in $D$ as explained above. But $\mathcal{C}_Y$ is declared to inherit from $\mathcal{C}_X$ and thus has at least the same columns as $\mathcal{C}_X$. This resembles the fact that any instance of $Y$ is also an instance of $X$. So when rows are read from $\mathcal{C}_X$ to find data about $X$ instances, PostgreSQL also reads data from $\mathcal{C}_Y$ since the rows there represent data about $Y$ instances (and thus also $X$ instances). In Example 1, $\mathcal{C}_{\text{HTMLDocument}}$ inherits from $\mathcal{C}_{\text{Document}}$ since HTMLDocument is a subclass of Document.

Any class $X$ defined in $O$ that is not a subclass of another class implicitly becomes a subclass of `owl:Thing`. Thus, if no other parent is specified for $X$, $\mathcal{C}_X$ inherits from $\mathcal{C}_{\texttt{owl:Thing}}$ as do $\mathcal{C}_{\text{Document}}$ and $\mathcal{C}_{\text{HTMLVersion}}$ in the running example.

**owl:ObjectProperty** and **owl:DataProperty:** A property (no matter if it is an `owl:ObjectProperty` or `owl:DataProperty`) results in a column in a table. If the property is an `owl:ObjectProperty`, the column is of type INTEGER such that it can hold the ID for the referenced instance. If the property on the other hand is an `owl:DataProperty`, the column is of a type that can represent the range of the property, e.g., VARCHAR or INTEGER.

If the `owl:maxCardinality` is 1, the column is placed in the class table for the most general class in the `rdfs:domain` of the property. Since there is at most one value for each instance, this makes it efficient to find the data since no joining is needed and one look-up in the relevant class table can find many property values for one instance.

If no `owl:maxCardinality` is specified, there may be an unknown number of property values to store for each instance and the idea about storing one property value in a column in the class table breaks. Instead, a column with an array type can be used. Another solution is to create a *multiproperty table*. Each row in the multiproperty table represents one value for the property for a specific instance. In a multiproperty table there are two columns: One to hold the ID of the instance that the represented property value applies to and one for the property value itself. This approach is illustrated for the keyword property in Example 1. In 3XL, it is left as a configuration choice if multiproperty tables or array columns should be used for multiproperties.

**rdfs:domain:** The `rdfs:domain` for a property decides which class table to place the column for the property in case it has a `owl:maxCardinality` of 1 or in case that array columns are used instead of multiproperty tables. In either case, the column to hold the property values is placed in $\mathcal{C}_T$ where $T$ is the domain.

If multiproperty tables are used and no `owl:maxCardinality` is given, the `rdfs:domain` decides which class table holds (directly or indirectly in a descendant table) the instances for which the property values are given. In other words, this decides where one of the IDs referenced by the multiproperty table exists. Note that no foreign key is declared in $D$. To understand this, recall that since there is only one multiproperty table for the given property, the most specific type of an instance that has this property may be different from the most general. So although the property has domain $X$, another class $Y$ may be a subclass of $X$, and $Y$ instances can be referenced by a property with range $X$. An example of this is seen in the running example, where keyword is defined to have the domain Document, but an HTMLDocument can also have keyword values. So in general there is not only one class table representing the range. Therefore we use a *loose foreign key*. A loose foreign key $LFK_\ell$ from $\mathcal{C}_X$ to $\mathcal{C}_Y$ is a column $\ell_X$ in $\mathcal{C}_X$ and a column $\ell_Y$ in $\mathcal{C}_Y$ with the constraint that if a row in $\mathcal{C}_X$ has the value $v$ for $\ell_X$, then at least one row in $\mathcal{C}_Y$ or a descendant table of $\mathcal{C}_Y$ has the value $v$ in the column $\ell_Y$. The crucial point here compared to a normal foreign key, is that the referenced value does not have to be in $\mathcal{C}_Y$, but can instead be in one of $\mathcal{C}_Y$'s descendants. Note that a loose foreign key is not enforced by the DBMS; this is left to 3XL to do. If no domain is given in $O$ for the property, it is implicitly assumed to be `owl:Thing`.

**rdfs:range:** The `rdfs:range` is used to decide where to find referenced instances for an `owl:ObjectProperty` and to decide the data type of the column holding values for an `owl:DataProperty`. So, similarly to the case explained above, the range decides which table the other ID of a multiproperty table for an object property references by a loose foreign key. Further, when the range of a property $p$ is known, the object of a triple where the predicate is $p$, can have its `rdf:type` inferred (although in OWL Lite, it also has to be given explicitly).

**owl:Restriction** (including **owl:onProperty** and **owl:maxCardinality**): In OWL, the way to say that a class $C$ satisfies a certain condition, is to say that $C$ is a subclass of $C'$ where $C'$ is the class of all objects that satisfy the condition [5]. The $C'$ class can be an anonymous class. To construct an anonymous class for which conditions can be specified, the

owl:Restriction construct is used. For an owl:Restriction, a number of things such as owl:maxCardinality can be specified.

Following the previous explanations about classes and subclasses this would lead to generating class tables for anonymous restrictions when 3XL generates the database schema. But since all instances of $C'$ (which is actually anonymous) would also be instances of the non-anonymous class $C$ and $\mathcal{C}_{C'}$ would thus be empty, this is more complex than needed. Instead, when 3XL generates the database schema, supported restrictions are "pulled down" to the non-anonymous subclass. So if the restriction $C'$ of which $C$ is a subclass, defines the owl:maxCardinality to be 1 for the property $P$ by means of owl:onProperty, this means that $P$ can be represented by a column in $\mathcal{C}_C$ and that no class table is generated for $C'$.

Currently, 3XL's restriction support is limited as only cardinality constraints are handled. As previously described, an owl:maxCardinality of 1 results in a column in a class table. Thus we assume that a property with max:Cardinality 1 only occurs once for a given subject. This deviates from the OWL semantics where it for a property $p$ with owl:maxCardinality 1 can be deduced that $o_1$ and $o_2$ are equivalent if the both the triples $(s,p,o_1)$ and $(s,p,o_2)$ are present.

The following table summarizes how OWL constructs from the ontology $O$ are mapped into the database schema $D$.

| **The construct** … | **results in** … |
|---|---|
| owl:Class | a class table |
| rdfs:subClass | the class table for the subclass inherits from the class table for the superclass |
| owl:ObjectProperty or owl:DataProperty | a column in a class table if the owl:maxCardinality is 1 and in a multiproperty table otherwise |
| rdfs:domain | a column for the property in the class table for the domain if the owl:maxCardinality is 1 and a loose foreign key from a multiproperty table to a class table otherwise. |
| rdfs:range | a type for the column representing the property |

3XL thus supports a subset of OWL Lite. This subset is enough to represent the real-life semantic data from the EIAO project which served as our initial motivation for 3XL. Later, support for more OWL Lite constructs can be added. For example, we envision that support for owl:sameAs could be implemented by representing sameAs-relationships explicitly in a table (not a class table, but a table managed by 3XL similarly to the map table). Queries would, however, then have to be rewritten if they involve an instance which is the sameAs another instance. Also, the construct owl:equivalentClass could be supported by letting 3XL maintain a mapping (likely in memory for efficiency) between classes and the classes that are physically represented by a class table.

### 2.3. Addition of triples

We now describe how 3XL handles triples that are inserted into a specific *model M* which is a database. $M$ has the database schema $D$ which has been generated as described above from the ontology $O_S$ with schematic data. We assume that the triples to insert are taken from an ontology $O_I$ which only contains data about instances, and not schematic data about classes, etc. Note that $O_I$ can be split up into several smaller sets such that $O_I = O_{I_1} \cup \cdots \cup O_{I_n}$ where each $O_{I_i}$, $i = 1, \ldots, n$, is added at a different time. In other words, unlike schema generation which happens only once, addition of triples can happen many times.

First, we focus on the state of $M$ after the addition of the triples in $O_I$ to give an intuition for the algorithms that handle this. Then, we present pseudocode in Algorithms 1–3 and explain the handling of triple additions in more details.

If the subject of a triple is an instance of a class that is not described by $O_S$, the triple is represented in the overflow table. Assume in the following that the subjects of the triples to insert are instances of classes described by $O_S$.

When a triple $(s, p, o)$ is added to $M$, 3XL has to decide in which class table and/or multiproperty table to put the data from the triple. Typically, the data in a triple becomes part of a row to be inserted into $M$. For each different $s$ for which a triple $(s,\texttt{rdf:type},t)$ exists[2] in $O_I$ and no triple $(s,\texttt{rdf:type},t')$ where $t'$ is more specific than $t$ exists in $O_I$, a row $\mathcal{R}_s$ is inserted into $\mathcal{C}_t$.

We now consider the effects of adding a triple $(s, p, o)$ where $p$ is a property defined in $O_S$. First, assume that $p$ is declared to have owl:maxCardinality 1. Then $\mathcal{R}_s$'s column for $p$ in $\mathcal{C}_t$ gets the value $v(p,o)$ which equals $o$ if $p$ is an owl:DataProperty or equals the value of the ID attribute in $\mathcal{R}_o$ if $p$ is an owl:ObjectProperty. In other words, the value of a data property is stored directly whereas the value of an object property is not stored as a URI but as the (more efficient) integer ID of the referenced object.

Now assume that no owl:maxCardinality is given for $p$. As previously mentioned, such properties can be handled in two ways. If array columns are used, the situation resembles that of a property with a maximal cardinality of 1. The only difference is that the column for $p$ in $\mathcal{R}_s$ does not get its value set to $v(p,o)$. Instead the value of $v(p,o)$ is added to the array in the column for $p$ in $\mathcal{R}_s$. If multiproperty tables are used, the row $(\iota, v(p,o))$ where $\iota$ is the value of the ID attribute in $\mathcal{R}_s$ is added to the multiproperty table for $p$. In other words, the row that is inserted into the multiproperty table has a reference (by means of a loose foreign key) to the row $\mathcal{R}_s$. Further, it has a reference to the row for the referenced object if $p$ is an owl:ObjectProperty and otherwise the value of the property.

So for properties defined in $O_S$, the values they take in $O_I$ are stored explicitly in columns in class tables and multiproperty tables. For other triples, information is not stored explicitly by adding a row. If the predicate $p$ of a triple $(s, p, o)$ is rdf:type, this information is stored implicitly since this triple does not result in a row being added to $M$, but decides in which class table $\mathcal{R}_s$ is put.

The pseudocode listed in Algorithms 1–3 shows how addition of triples is handled. For a so-called *value holder vh*, we denote by $vh[x]$ the value that $vh$ holds for $x$. We let the

[2] Recall that the type must be explicitly given.

value holders hold lists for multiproperties and denote by $\circ$ the concatenation operator for a list.

**Algorithm 1.** AddTriple

```
Require: A triple (s,p,o)
1: vh←GetValueHolder(s)
2: if p is defined in O_S then
3: if domain(p) is more specific than vh[rdf:type] then
4: vh[rdf:type]←domain(p)
5: if maxCardinality(p) = 1 then
6: vh[p]←Value(p,o)
7: else
8: vh[p]←vh[p]∘Value(p,o)
9: else if p = rdf:type and o is more specific than vh[rdf:type] and
   o is described by O_S then
10:vh[rdf:type]←o
11:else
12:Insert the triple into overflow
```

**Algorithm 2.** GetValueHolder

```
Require A URI u for an instance
1:  if the data buffer holds a value holder vh for u then
2:    return vh
3:  else
4:    table← The class table holding u (found from map)
5:    if table is not NULL then
6:      /* Read values from the database */
7:      vh←new ValueHolder()
8:      Read all values for u from table and assign them to vh.
9:      Delete the row with URI u from table
10:     for all multiproperty tables mp referencing table do
11:       Read all property values in rows referencing the row for u
          in table and assign these values to vh
12:       Delete from mp the rows referencing the row with URI u in
          table
13:     Add vh to the data buffer
14:     return vh
15:   else
16:     /* Create a new value holder */
17:     vh←new ValueHolder()
18:     vh[URI]←u
19:     vh[ID]← a unique ID
20:     vh[rdf:type]← owl:Thing
21:     Add vh to the data buffer
22:     return vh
```

**Algorithm 3.** Value

```
Require: A property p and an object o
1:  if p is an owl:ObjectProperty then
2:      res← the ID of the instance with URI o (found from map)
3:      if res is NULL then
4:                          res←(GetValueHolder(o))[ID]
5:      return res
6:  else
7:      /* It is an owl : DataProperty*/
8:      return o
```

When triples are being added to $M$, 3XL may not immediately be able to figure out which table to place the data of the triple in. For this reason, and to exploit the speed of bulk loading, data to add is temporarily held in a *data buffer*. Data from the data buffer is then, when needed, flushed into the database. This is illustrated in Fig. 2.

The data buffer does not hold triples. Instead it holds *value holders* (see Algorithm 1, line 1 and Algorithm 2). So for each subject $s$ of triples that have data in the data buffer, there is a value holder associated with it. In this value holder, an associative array maps between property names and values for these properties. In other words, the associative array for $s$ reflects the mapping $p \mapsto v(p,o)$. Note that if the predicate $p$ of a triple $(s,p,o)$ is `rdf:type`, $p \mapsto o$ is also inserted into the associative array in the value holder for $s$ unless the associative array already maps `rdf:type` to a more specific type than $o$. Actually, 3XL infers triples of the form $(s,$`rdf:type`$,o)$ based on predicate names, but only the most specialized type is stored (Algorithm 1 lines 3–4). This type information is later used to determine where to place the values held by the value holder. For a multiproperty $p$, the associative array maps $p$ to a list of values (Algorithm 1, line 8) but for a property $q$ with a maximal cardinality of 1, the associative array maps $q$ to a scalar value (Algorithm 1, line 6). Further, 3XL assigns a unique ID to each subject which is also held by the value holder (Algorithm 2, line 19 when the value holder is created).

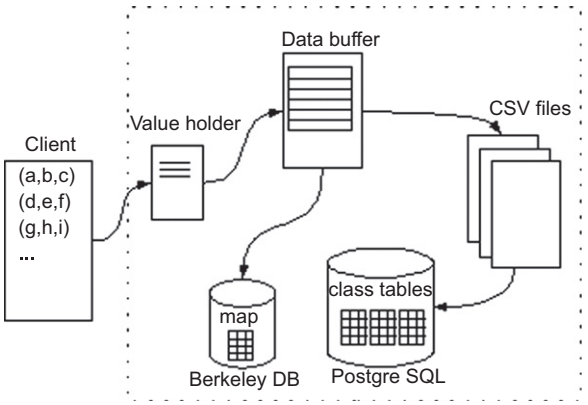


**Fig. 2.** Data flow in 3XL.

**Example 2** (*Data buffer*). Assume that the following triples are added to an empty 3XL model $M$ for the running example:

- (http://example.org/HTML-4.0, version, "4.0");
- (http://example.org/HTML-4.0, approvalDate, "1997-12-18");
- (http://example.org/programming.html, title, "How to Code?");
- (http://example.org/programming.html, keyword, "Java");
- (http://example.org/programming.html, keyword, "programming").

Before the triples are inserted into the underlying database by 3XL, the data buffer has the following state.

| http://example.org/HTML-4.0 | | |
|---|---|---|
| ID | ↦ | 1 |
| rdf:type | ↦ | HTMLVersion |
| version | ↦ | 4.0 |
| approvalDate | ↦ | 1997-12-18 |

| http://example.org/programming.html | | |
|---|---|---|
| ID | ↦ | 2 |
| rdf:type | ↦ | Document |
| title | ↦ | How to Code? |
| keyword | ↦ | [programming, Java] |

Here the top row of a table shows which subject, the value holder holds values for. The following rows show the associative array. Note that the type for http://example.org/programming.html is assumed to be Document since this is the most general class in the domains of title and keyword.

Now assume that the triple (http://example.org/programming.html, usedVersion, http://example.org/HTML-4.0) is added to $M$. Then the type detection finds that http://example.org/programming.html must be of type HTMLDocument, so its value holder gets the following state.

| http://example.org/programming.html | | |
|---|---|---|
| ID | ↦ | 2 |
| rdf:type | ↦ | HTMLDocument |
| title | ↦ | How to Code? |
| keyword | ↦ | [programming, Java] |
| usedVersion | ↦ | 1 |

Note how the value holder maps usedVersion to the ID value for http://example.org/HTML-4.0, not to the URI directly. If the required `rdf:type` triples now are inserted, this does not change anything since the type detection has already deduced the types.

Due to the definition of $v$ described above, the value holders and eventually the columns in the database hold IDs of the referenced instances for object properties. But when triples are added, the instances are referred to by URIs. So on the addition of the triple $(s,p,o)$ where $p$ is an object property, 3XL has to find an ID for $o$, i.e., $v(p,o)$. If $o$ is not already represented in $M$, a new value holder for $o$ is created (Algorithm 3, line 4). Depending on the range of $p$, type information about $o$ may be inferred. If $o$ on the other hand is already represented in $M$, its existing ID should of course be used. It is possible to search for the ID by using the query `SELECT id FROM` $\mathcal{C}_{\texttt{owl:Thing}}$ `WHERE uri =` $o$. However, for a large model with many class tables and many rows (i.e., data about many instances) this can be an expensive query. To make this faster, 3XL maintains a table `map(uri, id, ct)` where `uri` and `id` are self-descriptive and `ct` is a reference to the class table where the instance is represented. Whenever an instance is inserted into a class table $\mathcal{C}_X$, the instance's URI and ID and a reference to $\mathcal{C}_X$ are inserted into `map`. By searching the data buffer and the `map` table, it is fast to look up if an instance is already represented and to get its ID if it is. The `map` table exists in the PostgreSQL database, but for performance reasons 3XL does not query/update the `map` table in the database while adding triples. Instead, 3XL only extracts all rows in the table once when starting a load of triples and places them in a temporary BerkeleyDB database [6] which acts like a cache. With BerkeleyDB it is possible to keep a configurable amount of the data in memory and efficiently and transparently write the rest to disk-based storage.

Similarly, 3XL also needs to determine if the instance $s$ is already represented when adding a triple $(s,p,o)$. Again the `map` table is used. If $s$ is not already represented, a new value holder is created and added to the data buffer. If $s$ on the other hand is represented, a value holder is created in the data buffer and given the values that can be read from the class table referenced from `map` and then $\mathcal{R}_s$ and all rows referencing it from multiproperty tables are deleted. In this way, it is easy to get the new and old data for $s$ written to the database as data for $s$ is just written as if it was all newly inserted. This also helps, if it due to newly added data becomes evident that $s$ has a more specialized type than known before. In our implementation, the deletions are not done immediately as shown in the pseudocode. For a better performance, we invoke one operation deleting several rows before inserting new data.

When the data buffer gets full, a part of data in the data buffer is inserted into the database. This is done in a bulk operation where PostgreSQL's very efficient COPY mechanism is used instead of INSERT SQL statements. So the data gets dumped from the data buffer to temporary files in comma-separated values (CSV) format and the temporary files are then read by PostgreSQL. The `rdf:type`s read from the value holders are used to decide which tables to insert the data into. In case, no type is known, `owl:Thing` is assumed. For unknown property values, NULL is inserted. If multiproperty tables are used, values from a multiproperty are inserted into these instead of a class table.

To exploit that the data might have *locality* such that triples describing the same instance appear close to each other, a *partial-commit* mechanism is employed, in which the least recently used $m$% of the data buffer's content is moved to the database when the data buffer gets full (the percentage $m$ is user-configurable). This is illustrated in Fig. 3. In this way, the system can in many cases avoid reading in the data just written out to the database.

### 2.4. Triple queries

In this section, we describe the two types of queries, point-wise queries and composite queries, which are implemented in 3XL.

#### 2.4.1. Point-wise queries

A point-wise query is a triple, i.e., $Q=(s,p,o)$. Any of the elements in the query triple can take the special value $*$ to match anything. We consider how 3XL handles point-wise queries on the triples in a model $M$. Since schematic information given in $O_S$ (for which the specialized schema

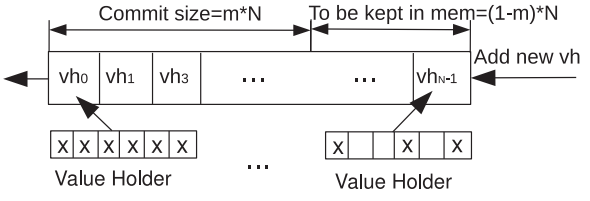


**Fig. 3.** The data buffer.

was generated) is fixed, we do not consider queries for schematic information here. Instead we focus on queries for instance data inserted into $M$, i.e., queries for data in $O_I$. The result of a query consists of those triples in $M$ where all elements *match* their corresponding elements in the query triple. The special $*$ value matches anything, but for all elements in $Q$ different from $*$, all corresponding elements in a triple $T \in M$ must be identical for $T$ to be included in the result.

**Example 3** (*Point-wise query*). Consider again the triples that were inserted in Example 2 and assume that only those (and the required triples explicitly giving the `rdf:type`) were inserted into $M$. The result of the query $(*,\text{keyword},*)$ is the set holding the following triples:

- (http://example.org/programming.html, keyword, Java);
- (http://example.org/programming.html, keyword, programming).

The result of the query (http://example.org/HTML-4.0, $*$,$*$) is the set holding the following triples:

- (http://example.org/HTML-4.0, rdf:type, owl:Thing);
- (http://example.org/HTML-4.0, rdf:type, HTMLVersion);
- (http://example.org/HTML-4.0, approvalDate, 1997-12-18);
- (http://example.org/HTML-4.0, version, 4.0);

i.e., the set containing all the knowledge about http://example.org/HTML-4.0, including all its known types.

For each query, the `overflow` table is searched and the result set of this is unioned with the results of searching the class and multiproperty tables. The `overflow` table is considered with a single SQL statement where all overflow triples with matching values are found. In the remaining descriptions, we focus on how the class and multiproperty tables are used to find the remaining triples of the result set.

As there are three elements in the query triple $Q$ and each of these can take an ordinary value or the special value $*$, there are $2^3=8$ generic cases to consider. We go through each of them in the following. $s$, $p$, and $o$ are all values different from $*$. When we for a subject $s$ say that the class table that holds $s$ is found, it is implicitly assumed that some class table actually holds $s$. If this is not the case, the result is of course just the empty set. Further, we assume that all data (including `map`'s data) is inserted into the database before the queries are executed.

**Case** $(s, p, o)$: In this case, the query is for the query triple itself, i.e., the result set is either empty or consists exactly of the query triple. If $p$ equals `rdf:type`, the result is found by looking in the `map` table to see if the class table holding $s$ is $\mathcal{C}_o$ or a descendant of $\mathcal{C}_o$. This is done by using the single SQL query `SELECT ct FROM map WHERE uri` $= s$ which can be performed fast if there is an index on `map(uri, ct)`. If $s$ is held by $\mathcal{C}_o$ or a descendant of $\mathcal{C}_o$, $Q$ is returned and otherwise an empty result is returned.

If $p$ is different from `rdf:type`, the result is found by finding the ID for $s$ (from now called $s$ id) and the class table where $s$ is inserted (by means of `map`). If that class table has a column or a multiproperty table for $p$, it is determined if the property $p$ takes the value $o$ for $s$. To determine this, it is necessary to look for $v(p,o)$ in the database as an ID is stored instead of a URI for an `owl:ObjectProperty`. If $p$ takes the value $o$ for $s$, $Q$ is returned, otherwise the empty result is returned. So this requires an SQL query selecting the class table (if $p$ is represented by a column) or the ID (if $p$ is represented by a multiproperty table) from `map` and either the query `SELECT true FROM classtable WHERE id` $=$ $s$`id AND` $p$`column` $= v(p,o)$ (if $p$ is not a multiproperty), the query `SELECT true FROM classtable where id` $=$ $s$`id AND` $v(p,o) =$ `ANY(`$p$`column)` (if $p$ is a multiproperty represented by an array column), or the query `SELECT true FROM` $p$`table WHERE id` $= s$`id AND value` $= v(p,o)$ (if $p$ is a multiproperty represented by a multiproperty table). In any case, only 2 SQL SELECT queries are needed and – except when $p$ is represented by an array column – indexes on the ID and $p$ columns can help to speed up these queries.

**Example 4** (*Finding a specific triple*). Let $Q$=(http://example.org/programming.html, keyword, programming) be a query given in the running example. To answer this query, 3XL executes the following SQL queries since keyword is represented by a multiproperty table.

```
SELECT id FROM map WHERE uri= 'http://example.org/programming.html'
SELECT true FROM keywordTable WHERE id=$id AND value='programming'
```

The result from the database is `true` so the triple exists in the model and 3XL returns $Q$ itself as the result.

*Case* $(s,p,*)$: Also in this case, there is special handling of the situation where $p =$ `rdf:type`. Then, the `map` table is used to determine the class table $\mathcal{C}_X$ where $s$ is located. The result set consists of all triples $(s, p, C)$ where $C$ is the class $X$ or an ancestor class of $X$. So the only needed SQL query is `SELECT ct FROM map WHERE uri=`$s$. Based on the result of this and its knowledge about class inheritance, 3XL generates the triples for the result.

If $p$ is an `owl:DataProperty`, the class table holding $s$ is found. From this, the row representing $s$ is found and each value for $p$ is read. If $p$ is a multiproperty and multiproperty tables are used, the values for $p$ are found in the multiproperty table instead by using the ID for $s$ as a search criterion. The result set consists of all triples $(s, p, V)$ where $V$ is a $p$ value for $s$. Again, only 2 SQL SELECTs are needed: One querying `map` and one querying for the value(s) for $p$ from either the class table or the multiproperty table for $p$. Indexes on `(uri, ct)` and `(uri, id)` in `map` and on the `id`s in the class table/multiproperty table will help to speed up these queries.

If $p$ is an `owl:ObjectProperty`, special care has to be taken as the URIs of the referenced objects should be found, not their IDs. The first step is to find the class table $\mathcal{C}_X$ holding $s$ and the ID of $s$ by means of single SELECT on the `map` table. Assume WLOG that the range of $p$ is $R$. If $p$ is represented by the column $p$`column` in $\mathcal{C}_X$, the query `SELECT` $\mathcal{C}_R$.uri `FROM` $\mathcal{C}_X$, $\mathcal{C}_R$ `WHERE` $\mathcal{C}_X.p$`column` $= \mathcal{C}_R$.`id AND` $\mathcal{C}_X$.`id` $= s$`id` is used. If $p$ is

represented by a multiproperty table mp, $\mathcal{C}_R$ is joined with mp instead of $\mathcal{C}_X$. If $p$ is a multiproperty represented by an array column, $\mathcal{C}_X$ and $\mathcal{C}_R$ are still joined, but the condition to use is WHERE $\mathcal{C}_R$.id = ANY($\mathcal{C}_X$.$p$column) AND $\mathcal{C}_X$.id = $s$id. The result set holds all triples ($s$,$p$,$U$) where $U$ ranges over the selected URIs.

*Case* ($s$,∗,$o$): In this case, the class table holding $s$ is found. Then all property values (including values in multiproperty tables) are searched. The result set consists of all triples ($s$, $P$, $o$) where $P$ is a property that takes the value $o$ for $s$. So by iterating over the properties defined for the class that $s$ belongs to, the previous ($s$,$p$,$o$) case can be used to find the triples to include. Note that also the special case ($s$,rdf:type,$o$) should be considered for inclusion in the result set. So for this query type, an SQL query selecting the class table and the ID from map is needed. Further, the SQL query SELECT true FROM mp WHERE ID = $s$id AND value = $v(p,o)$ is needed for each multiproperty table mp representing a property $p$ defined for $s$'s class as is the SQL query SELECT true FROM classtable WHERE id = $s$id AND pc = $v(p,o)$ for each column pc representing a property $p$ for $s$ in the class table holding $s$.

*Case* ($s$,∗,∗) In this case, the class table holding $s$ is found by using map. For each property $P$ defined in $O_S$, each of its values $V$ for $s$ is found. The result set consists of all triples ($s$, $P$, $V$) unioned with the triples in the result set of the query ($s$,rdf:type,∗).

In this case the following SQL queries are needed: One selecting the class table and ID from map, the query SELECT $p_1$column, ..., $p_n$column FROM classtable WHERE id = $s$id if there are columns representing data properties $p_1, \ldots, p_n$ in the class table holding $s$, and a query SELECT value FROM mp WHERE id=$s$id for each multiproperty table mp representing a data property defined for $s$'s class. Again, indexes on the id attributes in the class tables and multiproperty tables speed up the queries. Further, SQL to find the URIs for the values of object properties is needed. So for each object property $q$ defined for the class that $s$ belongs to and which is not represented by an array column, the following query is used: SELECT $\mathcal{C}_R$.uri FROM $\mathcal{C}_R$, $\Phi$ WHERE $\mathcal{C}_R$.id = $\Phi$.$q$column AND $\Phi$.id = $s$id. Here $\Phi$ is a multiproperty table for $q$ or the class table holding $s$ and $R$ is the range of $q$. Indexes on the id attributes will again speed up the queries. If $q$ is represented by an array column, $\mathcal{C}_R$.id = ANY($\Phi$.$q$column) should hold instead of $\mathcal{C}_R$.id = $\Phi$.$q$column.

*Case* (∗,$p$,$o$): If $p$ equals rdf:type, the class table $\mathcal{C}_o$ is found and all URIs are selected from it (including those in descendant tables). The result set consists of all triples ($U$, $p$, $o$) where $U$ ranges over the found URIs. This requires only 1 SQL query: SELECT uri FROM $\mathcal{C}_o$.

If $p$ is different from rdf:type, 3XL must find the most general class $G$ for which $p$ is defined. If $p$ is represented by a multiproperty table $X$, the tables $X$ and $\mathcal{C}_G$ are joined and restricted to consider the rows where the column for $p$ takes the value $v(p,o)$ and the URIs for these rows are selected by the query SELECT uri FROM X, $\mathcal{C}_G$ WHERE X.id = $\mathcal{C}_G$.id AND value = $v(p,o)$. If $p$ is represented in a column in $\mathcal{C}_G$, all URIs for rows that have the value $v(p,o)$ in the column for $p$ (either as an element in case $p$ is a multiproperty represented by an array column or as the only value in case $p$ is not a multiproperty) are selected. This is done by using either the query SELECT uri FROM $\mathcal{C}_G$ WHERE $p$column = $v(p,o)$ or the query SELECT uri FROM $\mathcal{C}_G$ WHERE $v(p,o)$ = ANY ($p$column). The result set consists of all triples ($U$, $p$, $o$) where $U$ ranges over the selected URIs. The first of these queries benefits from an index on the column holding data for $p$, but for the latter a scan is needed as we are only looking for a particular value inside an array.

**Example 5** (*Find subjects from a* (∗,$p$,$o$) *query*). Consider the running example and assume that 3XL is given the query $Q$=(∗, keyword, programming). The most general class for which keyword is specified is Document so the SQL query SELECT uri FROM keywordTable, $\mathcal{C}_{\text{Document}}$ WHERE keywordTable.id = $\mathcal{C}_{\text{Document}}$.id AND value = 'programming' is executed. One URI is found by the query, so the triple (http://example.org/programming.html, keyword, programming) is returned by 3XL.

*Case* (∗,$p$,∗): If $p$ in this case equals rdf:type, the result set contains all triples describing types for all subjects in the model. So for each class table $\mathcal{C}_X$, all its URIs (including those in subtables) are found with the SQL query SELECT uri FROM $\mathcal{C}_X$ which performs a scan of $\mathcal{C}_X$ and its descendants. The result set consists of all triples ($U$, $p$, $X$) where $U$ ranges over the URIs selected from $\mathcal{C}_X$.

If $p$ is a data property, 3XL handles this similarly to the (∗,$p$,$o$) case described above with the exception that no restrictions are made for the object (i.e., the parts concerning $v(p,o)$ are not included in the SQL) and the values in the column representing $p$ are also selected. Again special care has to be taken if $p$ is an object property. It is then needed to join the class table or multiproperty table holding $p$ values to the class table for the range $R$ of $P$. Further, the column $\mathcal{C}_R$.uri should be selected instead of the column representing $p$ (this is similar to the already described ($s$,$p$,∗) case). For each row ($U$,$O$) in the SQL query's result, a triple ($U$,$p$,$O$) is included in 3XL's result set.

*Case* (∗,∗,$o$): In this case, all triples with the given $o$ as object should be returned. Consider that $o$ could be the name of a class in which case type information must be returned (note that we can ignore the possibility that $o$ is, e.g., owl:ObjectProperty as we have assumed that there are no queries for schematic data given in $O_S$). We handle this part as in the (∗,$p$,$o$) case (with $p$ = rdf:type). But $o$ could also be any other kind of value that some property defined in $O_S$ takes for some instance. To detect if $o$ is another instance, we use the query SELECT ID FROM map WHERE URI=$o$. If the result is empty, we execute the query $q$ = SELECT * FROM $\mathcal{C}_X$ WHERE dp$_1$ = o OR ... OR dp$_n$ = o for each class $X$ (the dp$_j$'s are the columns holding X's data properties). If the result of the query towards the map table, on the other hand, found an ID $i$, we append OR op=$i$ for each column op representing an object property in $\mathcal{C}_X$.

*Case* (∗,∗,∗): In this case, all triples in $M$ should be returned. This can also be done by reusing some of the previously described cases. More concretely the result set for this query consists of a union of all type information triples and the union of all result sets for the queries (∗,$p$,∗) where $p$ is a property defined in $O_S$. Formally, the result set is given by the following where $\Omega(a,b,c)$ denotes the result set for the query ($a$,$b$,$c$) and $P$ is the set of properties defined in

$O_S$ : $\Omega(*,\texttt{rdf:type},*) \cup (\bigcup_{p \in P} \Omega(*,p,*))$. In other words, this is handled similarly to how the $(*,*,o)$ case is handled.

#### 2.4.2. Composite queries

Composite queries are composed of a conjunction of several query triples, each of the form (subject, predicate, object). Unknown *variables* used for *linking* triples together, are specified using a string starting with a question mark, while known *constants* are expressed using their full URIs or in an abbreviated form where prefixes can be replaced with shorter predefined values. There are three different patterns for linked query triples:

The first pattern is query triples ordered in a "chain" in which the object of a triple is the subject of the next triple (see Fig. 4). The second pattern is query triples in a "star" which share a common subject (see Fig. 5). The third is the combination of the two others. Each query pattern is characterized by *paths* which connect query triples together. A node in the paths is a subject or an object, and an edge connecting two nodes is a predicate. When the query engine constructs SQL statements needed to answer a composite query, a table join is produced for two adjacent nodes if the property linking them is an object property (if the property is a multiproperty, the multiproperty table is also included in the join). If the property is a data property, it can be processed similarly to the ways we have discussed above for the point-wise queries whose predicates are known.

After the SQL statement is generated, it is directly issued to the underlying DMBS. The advantage of this approach is that, by exploiting the DBMS, we can take advantage of its sophisticated query evaluation and optimization mechanisms for free. Note that there are table joins between different class tables and between class tables and multiproperty tables. As all class tables and multiproperty tables are indexed – typically, there are indices on all the ID columns and loose foreign key columns – the joins are not expensive.

In the following, we give an example to illustrate how a composite query is converted into SQL by the 3XL query engine. The query is used to find all HTML documents with the keyword Java and their corresponding versions.

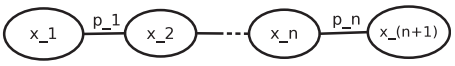


**Fig. 4.** "Chain" pattern.

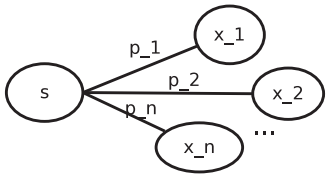


**Fig. 5.** "Star" pattern.

**Example 6** (*Composite query*). Consider the composite query (?x, rdf:type, HTMLDocument) (?y, rdf:type, HTMLVersion) (?x, keyword, Java) (?x, usedVersion, ?y) in which keyword is a multiproperty of owl:DataProperty type, and usedVersion is of owl:ObjectProperty type. When the multiproperty is implemented as an array, the composite query gets translated into:

SELECT $\mathcal{C}_{\text{HTMLDocument}}$.uri AS x, $\mathcal{C}_{\text{HTMLVersion}}$.uri AS y FROM $\mathcal{C}_{\text{HTMLDocument}}$, $\mathcal{C}_{\text{HTMLVersion}}$ WHERE $\mathcal{C}_{\text{HTMLDocument}}$.usedVersion = $\mathcal{C}_{\text{HTMLVersion}}$.ID AND 'Java' = ANY($\mathcal{C}_{\text{HTMLDocument}}$.keyword).

When the multiproperty is implemented as a multiproperty table, the composite query gets translated into:

SELECT $\mathcal{C}_{\text{HTMLDocument}}$.uri AS x, $\mathcal{C}_{\text{HTMLVersion}}$.uri AS y FROM $\mathcal{C}_{\text{HTMLDocument}}$, $\mathcal{C}_{\text{HTMLVersion}}$, keywordTable WHERE $\mathcal{C}_{\text{HTMLDocument}}$.usedVersion = $\mathcal{C}_{\text{HTMLVersion}}$.ID AND $\mathcal{C}_{\text{Document}}$.id = keywordTable.id AND keywordTable.keyword = 'Java'.

Fig. 6 shows the result of this composite query.

## 3. Performance evaluation

### 3.1. Experiment settings

We first conduct an experimental study to analyze the effectiveness of the various optimizations we made in the implementation. Then, we evaluate the loading and query performance of 3XL in comparison with the two state-of-the-art high performance triple-stores, BigOWLIM [7] and RDF-3X [8]. BigOWLIM is a commercial tool which is implemented in Java as a Storage and Inference Layer (SAIL) for the Sesame RDF database. It supports full RDFS, different OWL variants including most of OWL Lite. RDF-3X is an open source RISC-style engine with streamlined indexing and query processing which provides schema-free RDF data storage and retrieval. Unlike 3XL, the reference systems both use file-based storage. We note that it is not our goal to necessarily be strictly faster than the reference systems, but instead to provide comparable performance in combination with the flexibility of a DBMS-based solution. Finally, we compare the performance of 3XL to that of other DBMS-based triple-stores.

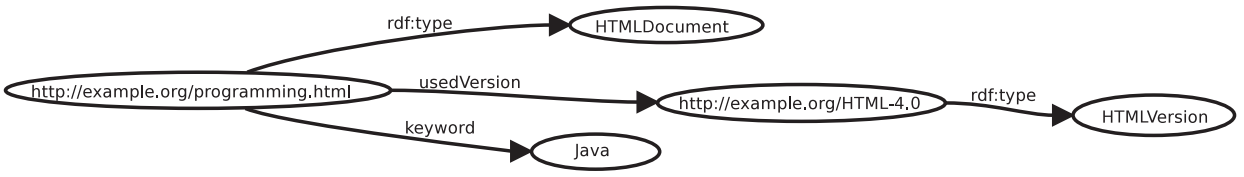


**Fig. 6.** The result of the composite query.

In the experiments, the following two datasets are used:

- *EIAO dataset*: This is a real-world dataset from the European Internet Accessibility Observatory (EIAO) project [3] which developed a tool for performing automatic evaluation of accessibility of web sites. This project serves as the design inspiration for the 3XL triple-store. The EIAO dataset conforms to an OWL Lite ontology [9] which contains 16 classes and 75 properties. Among the properties, 18 are of type `OWL:ObjectProperty`, 57 are of type `OWL:DataProperty`, and 29 are multiproperties.
- *LUBM dataset:* This is a synthetic dataset describing fictitious universities from the Lehigh University Benchmark (LUBM) [4] which is the de facto industry benchmark for OWL repository scalability. For 3XL, we use an ontology which is based on a subset of the published LUBM ontology, but only uses the 3XL-supported constructs and makes implicit subclass relationships explicit. Our ontology covers 20 classes and 20 properties. Among the properties, 13 are of type `OWL:ObjectProperty`, seven are of type `OWL:DataProperty`, and four are multiproperties. The ontology allows datasets generated by the (unmodified) LUBM generator to be loaded into 3XL. It is available from www.cs.aau.dk/∼xiliu/3xlsystem/.

We benchmark the performance when loading up to 100 million triples from each dataset (corresponding to 724 universities in the generated LUBM dataset). Before the loading, the original datasets are converted to N-triples format by the Redland RDF parser. All compared systems read input data from the N-triples format. The time spent on parsing is included in the overall loading time. In the query performance study, 14 queries are studied on the LUBM dataset, and 10 queries on the EIAO dataset. Both datasets contain 25 million triples. To reduce caching effects, a query is run 10 times with randomly generated query condition values and the average time is calculated. For example, in the performance study of the query $(s,p,*)$, 10 different values of $s$ are used. Between each query execution, the DBMS is restarted and all caches are cleared.

All experiments are conducted on a DELL D630 notebook with a 2.2 GHz Intel(R) Core(TM)2 Duo processor, 3 GB main memory, Ubuntu 10.04 with 64-bit Linux 2.6.32-22 kernel and java-6-sun-1.6.0.20. All the experiments are done under console mode, which all the unnecessary services are disabled including Linux X server. The JVM options "-Xms1024m -Xmx2500m -XX:-UseGCOverheadLimit -XX:+UseParallelGC" are used for both 3XL and BigOWLIM. PostgreSQL 8.3.5 is used as the RDBMS for 3XL with the settings "shared_buffers=512 MB, temp_buffers=128 MB, work_mem=56 MB, checkpoint_segments=20" and default values for other configuration parameters. BigOWLIM 3.3 is configured with Sesame 2.3.2 as its database, and with the following runtime settings: "owlim:ruleset empty; owlim:entity-index-size 5000,000; owlim:cache-memory 200M". This means that no reasoning is done during data loading. The cache memory is calculated by using a configuration spreadsheet included with the BIGOWLIM distribution, and the other settings are as referenced in [7]. For RDF-3X, we follow the setup from [8].

The source code for 3XL, the used datasets and queries, instructions, etc. are available from www.cs.aau.dk/∼xiliu/3xlsystem/.

### 3.2. Loading time

We first study the effect of the various optimizations in our implementation. To find the performance contribution of each optimization, we measure the loading time of each by using the LUBM dataset and compare with the non-optimized result. We focus on four aspects: (1) bulk-loading, (2) partial-commit, (3) using BDB to cache the `map` table, and (4) their combination. Fig. 7 shows the results. First, we see that the loading times grow linearly (with the slight exception of the two middle ones) with increasing dataset size, but at very different rates. When loading without any optimization, data is inserted into the `map` and class tables using SQL INSERTs through JDBC. Before INSERTing, triples with the same subject already existing in the database are removed using DELETEs. Here, it takes a staggering 252 h to load 100 M triples (average speed = 65 triples/s). Using BDB (next line) has little effect. Switching to buffering triples (by means of value holders) and using JDBC batch INSERTs (next line, in the lower part of the figure) is almost 30 times faster, showing the effectiveness of the 3XL buffering. Adding the partial commit (PC) optimization (line Insert, VH, PC.) is about 13% faster (8.95 vs. 10.24 h) than using normal "full" commit. This is because partial commit takes advantage of data locality to improve the buffer hit rate and thus reduces the number of database accesses when generating value holders (see Section 2).

The next big jump in performance comes from using bulkload rather than batch INSERTs (line Bulkload, VH), yielding a further 4–5 fold performance improvement. The bulkload results are also shown in Fig. 8 for better readability. Combining bulkload with a Berkeley DB cache for the `map` table further contributes about 50% performance improvement (line Bulkload, VH, BDB). This is due to two reasons: first, the BDB `map` accelerates the identification of the class table during value holder generation, and second, 3XL saves the load time of `map` table data. In our initial implementation, we maintained the `map` table

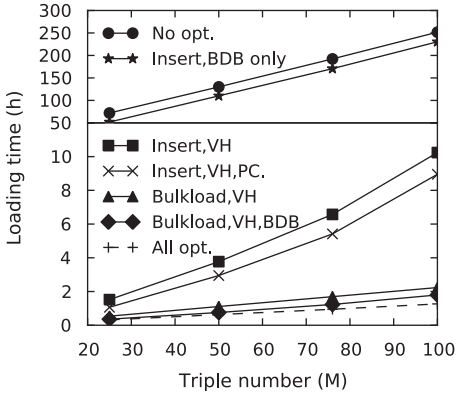


**Fig. 7.** Effect of optimizations.

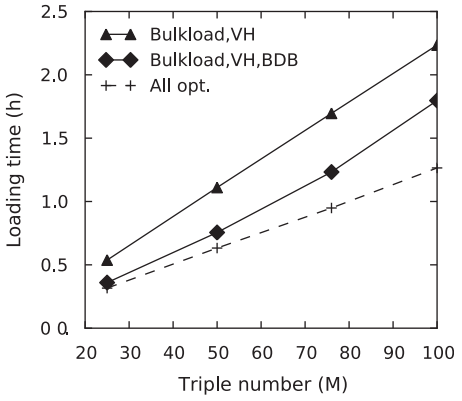


**Fig. 8.** Bulkload optimizations.

in the database like the class tables: *buffer→CSV→copy to database*. In this scheme, loading 100 M triples required 16 commit operations and used 200 min to load the `map` table alone. In comparison, the current scheme only needs 10 min to load the `map` table, so the optimized scheme of maintaining separate `map` tables for loading and querying is obviously better. Finally, adding partial commit (line All opt.) yields another 15.5% improvement. In summary, the optimization yielded a 202-fold performance improvement, from 255 to 1.26 h, thus demonstrating the effectiveness of the 3XL design and implementation choices.

We have performed experiments to find the optimal configuration parameters, including (a) using two different buffer schemes: caching class instances in class-specific buffers vs. caching all instances in one common buffer, (b) varying the value holder buffer size, (c) varying the Berkeley DB buffer size, (d) varying the partial commit value, and (e) using different cache algorithms including *first in, first out* (FIFO), *least recently used* (LRU) and *least frequently used* (LFU). We found that using one common buffer and the LRU cache algorithm is best for loading performance. The partial commit value depends on the data locality of the dataset, i.e., a lower partial-commit value should be set for a higher data locality, and vice versa. The best sizes for the value holder buffer and BDB cache also depend on the characteristics of the data and the hardware configuration. In addition, using a memory-based file system (tmpfs) [10] to cache the CSV files was tested, but the built-in OS file caching works so well that explicitly using *tmpfs* did not improve performance.

Using the full set of the optimizations, we now compare 3XL with the reference systems by loading 100 M triples from each dataset. The sizes of EIAO and LUBM datasets in N-Triples format are 18.5 and 16.6 GB respectively. However, a characteristic of the two datasets is that EIAO contains many duplicated triples while LUBM contains only distinct triples. Therefore, based on the above optimization study, we set the partial-commit value to be 0.4 for the EIAO dataset and 0.8 for the LUBM dataset. The value holder buffer size is set to 150,000 and 300,000, respectively. The BDB cache size is 300 MB for both datasets.

The loading results are shown in Table 1. For both datasets, 3XL with multiproperties represented in arrays ("3XL-Array") and 3XL with multiproperty tables ("3XL-MP") rank between

**Table 1**
The comparison on loading 100 M triples.

| | EIAO | | LUBM | |
|---|---|---|---|---|
| | Load time (min) | DB size (GB) | Load time (min) | DB size (GB) |
| 3XL-Array | 94.3 | 6.2 | 67.3 | 6.1 |
| 3XL-MP | 90.0 | 6.2 | 75.9 | 6.1 |
| BigOWLIM | 86.5 | 13.0 | 55.5 | 13.0 |
| RDF-3X | 158.7 | 5.3 | 155.7 | 5.1 |

BigOWLIM and RDF-3X. For the real-world EIAO dataset, 3XL-MP only uses 4% more time than the state-of-the-art triple-store BigOWLIM while 3XL-Array only uses 9% more time. For the synthetic LUBM datasets, 3XL-MP uses 36% more time than BigOWLIM and 3XL-Array uses 21% more time. However, BigOWLIM is using a file-based data store, which is reported [11] to have higher loading performance than relational database-based stores in general. RDF-3X has the slowest load performance, using 83% and 180% more time than BIGOWLIM for the EIAO and LUBM datasets, respectively. For both datasets, the 3XL variants consume less than half of the disk space BigOWLIM requires. The RDF-3X DB size is the smallest, about 15% smaller than the 3XL DB size. Further, 3XL offers a high degree of flexibility in integrating the data with other (non-triple) datasets as 3XL stores data in an intuitive relational database. With this design goal in mind, it is thus very satisfying to achieve a load-performance which is comparable to the state-of-the-art triple-store BigOWLIM.

3XL takes longer time to load the EIAO dataset than the LUBM dataset, although the datasets occupy nearly the same amount of space when loaded. There are two reasons for this. First, as the EIAO ontology has more classes and properties, the loading process needs to operate on more class tables and attributes in the database which takes more time. Second, when processing the duplicated EIAO data, there is a higher possibility that 3XL needs to fetch already inserted data from the database. However, this possibility has been reduced by using a *least recently used* (LRU) cache algorithm and a lower partial-commit value. With the current settings, loading 100 M EIAO triples still involves 207,183 database visits compared to zero when loading the LUBM triples.

In addition to the comparisons with BigOWLIM and RDF-3X, we have also compared with the popular Jena2 systems (filed-based and DBMS-based) [12] by using the LUBM dataset. However, the loading performance of Jena2 systems was slow, and it took 32.6 and 20.4 h to import 100 M triples to Jena2(DBMS-based) and Jena2(file-based), respectively. In an earlier experiment, we also tried this testing on 3store, which uses a traditional giant *triple table*, but it was not able to scale to load 100 M triples as it did not finish in a reasonable amount of time.

RDF-3X does not exploit an OWL schema for loading the data (as it is based on RDF, not OWL). Unlike 3XL, RDF-3X does not support repeated loads. If data is loaded into an existing database, the previous data will be overwritten. BigOWLIM requires a known schema when doing reasoning during the load. However, no schema is used in this

**Table 2**
Load performance comparison (repositories) [7,11].

| RDF data (#Triples) | RATE (triples/s) | Configuration | Tool (year) |
|---|---|---|---|
| Native 2,023,606 | 400 db-based | Sun dual UltraSPARC-II 450 MHz 1G RAM | RSSDB 2001 |
| Native 279,337 | 418 db-based | Apple Powerbook G4 1.125 GHz 1G RAM | Jena(Mysql) 2004 |
| Native 279,337 | 4170 file-based | Apple Powerbook G4 1.25 GHz 1G RAM | Jena(file) 2004 |
| LUBM 6,890,933 | 151 db-based | P4 1.80 GHz 256M RAM | DLDB 2005 |
| LUBM 1.06 Bill. | 12,389 db-based | 2xXeon 5130 2 GHz, 8 GB RAM, RAID 4xSATA | OpenLink Virtuoso 2006 |
| LUBM 1.06 Bill. | 13,100 file-based | AMD 64 2 GHz 16G RAM | AllegroGraph 2007 |
| LUBM 100 Mill. | 10,750 db-based | P4 3.0 GHz 2G RAM | Oracle bulk-load scheme 2008 |
| Synthetic RDF 235 Mill. | 3840 file-based | AMD Opteron 1 GHz, 64-bit JDK, no other info. | KOWARI 2006 |
| LUBM 70 Mill. | 6481 file-based | P4 2.8 GHz, 1 GB, Xmx800, JDK 1.5 | Sesame's Native Store 2008 |
| Uniprot 262 Mill. | 758 db-based | No info. | RDF Gateway |

experiment as no reasoning is done. 3XL, on the other hand, exploits the schema for creating a specialized and intuitive database schema. It is, however, still possible to load data that is not described by the OWL schema by means of 3XL's `overflow` table, so 3XL supports the standard open world assumption.

### 3.3. Comparison with other bulk-loading systems

In order to compare with DBMS-based triple-stores, we refer to a bulk-loading study published by [7,11]. Table 2 (reproduced from [7,11]) summarizes the bulk-loading speed rates of a number of tools.

It is of course difficult to compare the results exactly as the datasets as well as the hardware configurations used vary significantly. However, certain things can be deduced. First of all, all the most recent results (2005 and onwards) are based on the *LUBM benchmark* which is also used for 3XL. There are both file-based (which have to implement their own transaction handling etc.) and DBMS-based (called "db-based") triple-stores in the table. As far as the performance of db-based triple-stores is concerned, the Virtuoso triple-store and the Oracle bulk-loading scheme (described in detail in [11]) have the best performance, with loading speeds of 12,692 and 10,750 triples/s, respectively. However, the Virtuoso scheme is run on much more powerful hardware, meaning that the Oracle scheme is in fact the fastest of the two. The file-based AllegroGraph stores 13,100 triples/s. Virtuoso's implementation has employed parallel loading techniques and storage optimization like bitmap indexes, etc., while the Oracle scheme has used the high-performance commercial DBMS Oracle and made use of its bulk-loading utility *SQL*Loader*. The paper [11] thus establishes the Oracle scheme as the leading DBMS-based bulk-loading scheme. It is thus natural to compare it with 3XL. However, the Oracle license explicitly disallows us to publish performance figures without the consent of Oracle, meaning that open and transparent comparisons are impossible.

We can see that the hardware used for the two setups is almost identical: our CPU is slower, but we have a little more main memory. However, on this almost identical hardware, we can see that 3XL is significantly faster than the Oracle scheme. When loading the LUBM data, 3XL-Array and 3XL-MP handle 24,765 and 21,959 triples/s, respectively. The difference is so profound that we think we can safely claim that 3XL outperforms the Oracle scheme (which handles 10,750 triples/s) for bulk-loading.

### 3.4. Query response time

We conduct the query testing on the EIAO dataset using 10 queries (Q1–Q10), and on the LUBM dataset using its standard 14 queries[3] (LQ1–LQ14). Each dataset contains 25 M triples. The queries are expressed in the form (*subject*, *predicate*, *object*) (with possible "∗"-values) for 3XL, and in SPARQL [13] for BigOWLIM and RDF-3X.[4] For example, a point-wise query $(s,p,*)$ with a given subject $s$ and predicate $p$, can be converted into SPARQL: `select ?o where {`$\langle s \rangle\ \langle p \rangle$ `?o.}`.

In 3XL, we have a specialized database schema for classes, properties and different kinds of restrictions. It is thus interesting to study the query performance for: (a) different properties, i.e., `owl:ObjectProperty` and `owl:DatatypeProperty`, (b) storing multiproperties in arrays vs. in tables, and (c) the difference between 3XL and the reference systems. We study these by doing the queries on the EIAO dataset, and present the results in Table 3. The queries Q1–Q4 are all of the form $(s,p,*)$, but with different types of the predicates, namely single-valued object property, single-valued data property, multi-valued object property, and multi-valued data property, respectively. Overall, in 3XL the queries on data properties are faster than queries on object properties, e.g., Q2 vs. Q1 and queries on single-valued properties are faster than on multiproperties, e.g., Q1 vs. Q3 and Q2 vs. Q4. No significant difference is observed between 3XL-Array and 3XL-MP except for Q3. We use the queries Q5–Q9 to study the performance of point-wise queries different from the $(s,p,*)$ form. Q10 is used to study the performance of a composite query. As shown in the results, 3XL outperforms the two reference systems for Q1–Q4 and Q7–Q8 where the subjects $s$ are given. This is mainly due to 3XL's "intelligent partitioning", where, given a particular subject $s$, 3XL can very quickly locate the class table holding the relevant data. For the composite query Q10, and the point-wise queries Q5–Q6 with wildcard "∗" in the subject but with a given predicate $p$, all the systems take a longer time than for the other queries as more results are returned. For these three queries, 3XL ranks in the middle. In the case of Q9 with only a given object $o$ and with the subject and the predicate using "∗" to match anything, 3XL takes a longer

[3] The LUBM queries are available from swat.cse.lehigh.edu/projects/lubm/query.htm.

[4] The SPARQL queries are available from www.cs.aau.dk/∼xiliu/3xlsystem.

**Table 3**
Query response time for the EIAO dataset with 25 M triples (ms).

| | Q1 ($s_1$,$p_1$,∗) | Q2 ($s_2$,$p_2$,∗) | Q3 ($s_3$,$p_3$,∗) | Q4 ($s_4$,$p_4$,∗) | Q5 (∗,$p_5$,∗) |
|---|---|---|---|---|---|
| 3XL-Array | 53 | 48 | 216 | 48 | 16,943 |
| 3XL-MP | 69 | 26 | 78 | 58 | 17,255 |
| BigOWLIM | 87 | 85 | 321 | 85 | 2002 |
| RDF-3X | 59 | 81 | 591 | 91 | 99,369 |
| | Q6 (∗,$p_6$,$o_1$) | Q7 ($s_5$,∗,∗) | Q8 ($s_6$,∗,$o_2$) | Q9 (∗,∗,$o_3$) | Q10 (Comp.) |
| 3XL-Array | 8984 | 23 | 38 | 6333 | 38,243 |
| 3XL-MP | 8934 | 68 | 75 | 3227 | 29,022 |
| BigOWLIM | 898 | 121 | 146 | 104 | 7398 |
| RDF-3X | 33466 | 46 | 38 | 225 | 139,951 |

**Table 4**
Query response time for the LUBM dataset with 25 M triples (ms).

| | LQ1 | LQ2 | LQ3 | LQ4 | LQ5 | LQ6 | LQ7 |
|---|---|---|---|---|---|---|---|
| 3XL-Array | 531 | 627 | 1376 | 127 | N/A | (11,526) | 2623 |
| 3XL-MP | 139 | 576 | 661 | 114 | N/A | (11,459) | 126 |
| BigOWLIM | 185 | 75,219 | 130 | 183 | 90 | 29,830 | 179 |
| RDF-3X | 35 | 36,096 | 74 | N/A | N/A | N/A | N/A |
| | LQ8 | LQ9 | LQ10 | LQ11 | LQ12 | LQ13 | LQ14 |
| 3XL-Array | 1675 | 56,835 | (1390) | N/A | N/A | N/A | 8764 |
| 3XL-MP | 1612 | 15,083 | (137) | N/A | N/A | N/A | 8745 |
| BigOWLIM | 676 | 100,037 | 2 | 154 | 674 | 11,796 | 46,625 |
| RDF-3X | N/A | N/A | N/A | N/A | N/A | N/A | 17,006 |

time as this query has to traverse all predicates $p_i$. Here, we note that 3XL is in fact specifically *designed* to be efficient for queries *with* a subject.

We now proceed to make an evaluation by using the LUBM dataset and its queries (LQ1–LQ14). These queries are all composite queries which are more expressive and complex than the point-wise queries we discussed above. Table 4 describes the test results of 3XL and the reference systems. Overall, BigOWLIM has the highest completeness and supports all 14 queries, while RDF-3X only supports four queries which is due to its lack of OWL inference. 3XL supports 10 of the LUBM queries. Because of its use of an inheritance database schema, 3XL has some semantic abilities and can reason on the instances of a class and its subclasses. When querying on a class table, all of its subclass tables are queried as well. Therefore, 3XL does support queries that, e.q., query a class and its subclasses. With regard to the query performance, 3XL-MP, in general, outperforms 3XL-Array since the multiproperty table is indexed. Neither of the systems is able to outperform all other systems for all queries. For LQ1, which selects instances of a given class which reference a certain instance of another class, RDF-3X has the least query response time while the times used by BigOWLIM and 3XL-MP are quite similar. For LQ2, which selects instances of three classes with a triangular pattern of relationships between the involved instances, both the 3XL variants are more than two orders of magnitude faster than the two reference systems. LQ3 is similar to LQ1 in both query characteristics and results. LQ4 selects instances and three property values from a class (with many subclasses) based on an object property linking to another class and is highly selective. 3XL-MP takes the least time for this query closely followed by 3XL-Array. RDF-3X does not support this query. LQ5 depends on `rdfs:subPropertyOf` which is not supported by 3XL. LQ6 selects all instances from a given class and its subclasses (an implicit subclass relationship was made explicit in the modified LUBM ontology and the timings for 3XL are therefore shown in parentheses). All the systems take a longer time on this query, but the 3XL variants both outperform BigOWLIM. LQ7 involves more classes and properties, and is more selective than LQ6. 3XL-MP is the fastest followed by BigOWLIM. LQ8 is based on LQ7 but adds one more property to increase the query complexity. The 3XL variants almost have equal performance which is lower than BigOWLIM's. LQ9 involves a triangular pattern of relationships between three classes. This query takes much longer time than all the other queries in all systems, but 3XL-MP has considerably better performance. LQ10 selects instances from a class but depends on the same implicit subclass relationship as does LQ6. The numbers shown in parentheses show the time spent by 3XL when the implicit relationship is given explicitly. LQ11, LQ12, and LQ13 depend on inference not supported by 3XL or RDF-3X. The last query LQ14 selects all instances of a given class (without subclasses). This query is similar to LQ6, but uses a subclass of the class used by LQ6. The 3XL variants are both faster than the two reference systems.

In summary of the performance results on the EIAO dataset, 3XL-Array and BigOWLIM both show the best performance in 4 out of the 10 queries, 3XL-MP shows the best performance in two queries, and RDF-3X shows the best performance (actually a tie) for a single query. For the LUBM

dataset, 3XL-MP shows the best performance for 6 out of the 14 queries while BigOWLIM has the best performance for seven queries and RDF-3X has the best performance for the remaining query.

In particular when querying triples with a shared subject, 3XL shows very good performance. 3XL has inference capabilities on instances with an inheritance relationship by means of its database schema. RDF-3X cannot answer many of the considered LUBM queries as it has no schema support and cannot do any inference.

In summary, the performance of 3XL is comparable to that of the state-of-the-art file-based triple-stores. This holds both for loading and querying. The performance of 3XL exceeds that of other DBMS-based triple-stores. 3XL is designed to use a database schema which is flexible and easy to use, and it is thus very satisfying that the solution achieves a very good performance, while offering the flexibility of the DBMS-based triple-store.

## 4. Related work

Different RDF and OWL stores have been described before. In this section we describe the most relevant ones. Note that terminology is used with different meanings in different solutions. For example, "class table" is not meaning the same in *RDFSuite* described below and in 3XL.

An early example of an RDF store can be found in *RDFSuite* [14,15]. In the part of the work focusing on storing RDF data, two different representations are considered: *GenRepr* which is a generic representation that uses the same database schema for all RDF schemas and *SpecRepr* which creates a specialized database schema for each RDF schema. It is found that the specialized representation performs better than the generic representation.

In the generic representation, two tables are used. One for resources and one for triples. In a specialized representation, RDFSuite represents the core RDFS model by means of four tables. Further, a specialized representation has a so-called *class table* for each class defined in the RDFS. In contrast to the class tables used by 3XL, RDFSuite's class tables only store the URIs of individuals belonging to the represented class. Both RDFSuite and 3XL use the table inheritance features of PostgreSQL for class tables. RDFSuite's specialized representation also has a so-called *property table* for each property. This is different from 3XL's approach where multiproperty tables only are used if the cardinality for the represented property is greater than 1. In RDFSuite, property tables store URIs for the source and target of each represented property value. Alexaki et al. [15] also suggest (but do not implement) a representation where single-valued properties with literal types as ranges are represented as attributes in the relevant class tables. This is similar to the approach taken by 3XL. In 3XL this is taken a step further and also done for attributes with object values.

In Broekstra et al.'s solution for storing RDF and RDFS, *Sesame* [16], different schemas can be used. Sesame is implemented such that code for data handling is isolated in a so-called *Storage and Inference Layer* (SAIL). It is then possible to plug-in new SAILs. A generic SAIL for SQL92 compatible DBMSes only uses a single table with columns for the subjects, predicates and objects. In a SAIL for PostgreSQL, the schema is inspired by the schema for RDFSuite and is dynamically generated. Again, a table is created for each class to represent. Such a table has one column for the URI. A table created for a class inherits from the tables created for the parents of the class. Likewise, a table is created for each property. Such a table for a property inherits from the tables that represent the parents of the property if it is a subproperty. This SAIL is reported [16] to have a good query performance but disappointing insert performance when tables are created.

In Wilkinson et al.'s [12] tool for RDF storage, *Jena2*, all statements can be stored in a single table. In the statement table, both URIs and literal values are stored directly. Further, Jena2 allows so-called *property tables* that store pairs of subjects and values. It is possible to cluster multiple properties that have maximum cardinality 1 together in one property table such that a given row in the table stores many property values for a single subject. These can be compared to 3XL's class tables. An important difference is, however, 3XL's use of table inheritance to reflect the class hierarchy.

Harris and Gibbins [17] suggest a schema with fixed tables for their RDF triple-store, 3store. One table with columns for subject, predicate and object holds all triples. To normalize the schema, there are also tables for representing models, resources, and literals. Each of these has two columns: one for holding an integer hash value and one for holding a text string. The triple table then references the integer values in these three tables. This approach where all triples are stored in one table is different from the approach taken by 3XL where the data to store is held in many different tables.

Pan and Heflin [18] suggest the tool *DLDB*. The schema for DLDB's underlying database is similar to RDFSuite's. DLDB also defines views over classes. A class's view contains data from the class's table as well as data from the views of any subclasses. Instead of views, 3XL uses table-inheritance. A DLDB version for OWL also exists.

Neumann and Weikum [8,19] suggest a scalable and general solution for storing and querying RDF triples. The system, called *RDF-3X*, does not use a DBMS, but a specialized storage system which applies intensive indexing to enable fast querying. A major difference between RDF-3X and 3XL is that 3XL uses (a subset of) OWL Lite and supports OWL classes, object and data properties, etc. and thus, unlike RDF-3X, can answer OWL queries as most of those in the LUBM benchmark.

Abadi et al. [20] propose to use a two-column table `property(subject, object)` for each unique property in an RDF data set. This is implemented in both a *column-store* which stores data by columns and in a more traditional *row-store* which stores data by rows. The proposed schema is reported to be about three times faster than a traditional triple-store in a row-store while it is around 30 times faster in a column-store. In a later evaluation paper, Sidirourgos et al. [21], however, find that in a row-store, the simple triple-store performs as well as the two-column approach if the right indexes are in place. They also find that a column-store provides good performance but that scalability becomes a problem when there are many properties (leading to many tables). 3XL is designed to be fast for queries where the subject and/or predicate is known and where many/all properties should be retrieved. Further, it exploits the object-relational capabilities of the row-store

PostgreSQL. We therefore believe that the best choice is to group the single-valued properties of a class together in a class table and allow multi-valued properties in special property tables or arrays in the class table.

Zhou et al. [22] implement the *Minerva* OWL semantic repository integrated with a DL reasoner and a rule inference engine. The imported data is inferred based on a set of rules, and the results are materialized to a DB2 database which has an inference-based schema containing atomic tables, TBox axiom tables, ABox fact tables and class constructor tables.

*IBM SHER* [23] is a reasoner that allows for efficient retrieval of ABoxes stored in databases. It achieves the efficiency by grouping the instances of a same class into a dramatically simplified summary ABox, and doing queries upon this ABox.

Storage of RDF data has also found its way into commercial database products. *Oracle 10g* and *11g* manage storage of RDF in a central, fixed schema [24]. This schema has a number of tables, including one that has an entry for each unique part of all the triples (i.e., up to three entries are made for one triple) and a table with one entry for each triple to link between the parts in the mentioned table. In a recent paper [11], it describes how Oracle supports efficient *bulk loading* by extensive use of SQL and a hash-based scheme for mapping between values and IDs.

Unlike Minerva's inference-based schema, SHER's summary ABox, and Oracle's use of a variant generic schema representation, 3XL uses PostgreSQL's object-oriented functionalities to support its specialized data-dependent schema.

Other repositories designed for OWL also exist. *DBOWL* [25] creates a specialized schema based on the ontology like 3XL does. There is a single table with three attributes to store triples. For each class and property in the OWL ontology, a view is created. 3XL differs from this, as data is partitioned over several physical tables that may have many attributes. *OWLIM* [26] is another solution. It is implemented as a SAIL for Sesame. Two versions of OWLIM exist: (1) The free SwiftOWLIM which is the fastest but performs querying and reasoning in-memory and (2) the commercial BigOWLIM which is file-based and scales to billions of triples. This is different from 3XL that has its data in an underlying PostgreSQL and exploits the results of decades of research and development in the database community such as atomicity, concurrency control, and abstraction.

## 5. Conclusion and future work

In this paper, we present the 3XL triple-store. Unlike most current triple-stores, 3XL is specifically designed to support easy integration with non-RDF data and at the same time support efficient data management operations (load and retrieval) on very large OWL Lite triple-stores. 3XL's approach has a number of notable characteristics. First, 3XL is DBMS-based and uses a *specialized data-dependent schema* derived from an OWL Lite ontology. In other words, 3XL performs an "intelligent partitioning" of the data which is efficiently used by the system when answering triple queries and at the same time intuitive to use when the user queries the data directly in SQL. Second, 3XL uses advanced object-relational features of the underlying ORDBMS (in this case PostgreSQL), such as table inheritance and arrays as "in-lined" attribute values. The table inheritance represents subclass relationships in a natural way to a user. Third, 3XL is designed to be efficient for bulk insertions. It makes extensive use of a number of bulk loading techniques that speed up bulk operations significantly and is designed to use the available main memory very efficiently, using specialized caching schemes for triples and the `map` table. Fourth, 3XL supports very efficient bulk retrieval for point-wise queries where the subject and/or the predicate is known, as we have found such queries to be the most important for most bulk data management applications. 3XL also supports efficient retrieval for composite queries. 3XL is motivated by our own experiences from a project using very large amounts of triples. Extensive experiments based on the real-world EIAO dataset and the industry standard LUBM benchmark show that 3XL has loading and query performance comparable to the best file-based solutions, and outperforms other DBMS-based solutions. At the same time, 3XL provides flexibility as it is DBMS-based and uses a specialized and intuitive schema to represent the data. 3XL thus bridges the gap between efficient representations and flexible and intuitive representations of the data. 3XL thus places itself in a unique spot in the design space for triple-stores.

The overall lessons learnt can be summarized as follows: (1) Using a specialized schema generated from an OWL ontology is very effective. With this schema, it is fast to find the relevant data which is intelligently partitioned into class tables (and possibly also multiproperty tables). This results in very good query performance. (2) An ORDBMS is a very strong foundation for building such a specialized schema for an OWL ontology. It provides the needed functionality (i.e., table inheritance) to represent class relationships and efficient storage of multi-valued variables in arrays. It also provides a query optimizer, index support, etc. for free. Finally, it provides the user flexibility as it is easy to combine the data with non-RDF data. (3) The right choice of caching mechanisms is very important for the performance. In particular, the often used `map` table should be cached outside the DBMS. For a `map` table too big to fit in memory, an external caching system based on BerkeleyDB is better than using the DBMS. (4) When loading OWL data, using bulk-loading in clever ways has a huge effect. In particular, instead of bulk-loading all available data, only the oldest parts should be loaded keeping the freshest data in memory.

There are a number of interesting directions for future work. First of all, optimization will be continued, focusing on performance improvement of data transferred from memory to database, and supporting queries of form $(*,*,o)$ better. Further, 3XL will be extended to support more of the OWL features, in the first case all of OWL Lite. Finally, 3XL will be integrated with a reasoner running on top of 3XL to allow more reasoning than the class-subclass reasoning on instances currently supported.

## Acknowledgements

This work was partly supported by the Agile & Open Business Intelligence (AOBI) project co-funded by the Regional ICT Initiative under the Danish Council for Technology and Innovation under Grant no. 07-024511, the European Internet Accessibility Observatory (EIAO) project

funded by the European Commission under Contract no. 004526, and the eGovMon project co-funded by the Research Council of Norway under the VERDIKT program (project no. Verdikt 183392/S10).

## References

[1] O. Lassila, R. Swick, Resource Description Framework (RDF) Model and Syntax Specification, W3C Recommendation, 1999, Available at: 〈w3.org/TR/REC-rdf-syntax〉 as of 2010-10-18.
[2] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. McGuinness, P. Patel-Schneider, L. Stein, OWL Web Ontology Language Reference, W3C Recommendation, 2004, Available at: 〈w3.org/TR/REC-rdf-syntax〉 as of 2010-10-18.
[3] C. Thomsen, T.B. Pedersen, Building a web warehouse for accessibility data, in: Proceedings of DOLAP, 2006, pp. 43–50.
[4] Y. Guo, J. Heflin, Z. Pan, LUBM: a benchmark for OWL knowledge base systems, J. Web Sem. 3 (2) (2005) 158–182.
[5] G. Antoniou, F. van Harmelen, A Semantic Web Primer, MIT Press, 2004.
[6] D.B. Berkeley, Oracle Embedded Database, Available at: 〈oracle.com/us/products/database/berkeley-db〉 as of 2010-10-18.
[7] BigOWLIM—Semantic Repository for RDF(S) and OWL, Available at: 〈www.ontotext.com/owlim/OWLIM_primer.pdf〉 as of 2010-10-18.
[8] T. Neumann, G. Weikum, RDF-3X: a RISC-style engine for RDF, in: Proceedings of the VLDB Endow, 2008, pp. 647–659.
[9] EIAO Ontology, Available at: 〈www.cs.aau.dk/~xiliu/3xlsystem/experiment/ontology/eiao.owl〉 as of 2010-10-18.
[10] P. Snyder, tmpfs: a virtual memory file system, in: Proceedings of EUUG, 1990, pp. 241–248.
[11] S. Das, E. Chong, W. Zhe, M. Annamalai, J. Srinivasan, A scalable scheme for bulk loading large RDF graphs into oracle, in: Proceedings of ICDE, 2008, pp. 1297–1306.
[12] K. Wilkinson, C. Sayers, H. Kuno, D. Reynolds, Efficient RDF storage and retrieval in Jena2, in: Proceedings of SWDB, 2003, pp. 131–150.
[13] G. Prud' Hommeaux, A. Seaborne, et al. SPARQL query language for RDF, in: J. of W3C working draft, 2006.
[14] S. Alexaki, V. Chrisophides, G. Karvounarakis, D. Plexousakis, K. Tolle, The ICS-FORTH RDFSuite: managing voluminous RDF description bases, in: Proceedings of ISWC, 2001, pp. 1–13.
[15] S. Alexaki, V. Chrisophides, G. Karvounarakis, D. Plexousakis, On storing voluminous RDF descriptions: the case of web portal catalogs, in: Proceedings of WebDB, 2001, pp. 43–48.
[16] J. Broekstra, A. Kampman, F. van Harmelen, Sesame: a generic architecture for storing and querying RDF and RDF schema, in: Proceedings of ISWC, 2002, pp. 54–68.
[17] S. Harris, N. Gibbins, 3Store: efficient bulk RDF storage, in: Proceedings of PSSS, 2003, pp. 1–15.
[18] Z. Pan, J. Heflin, DLDB: extending relational databases to support semantic web queries, in: Proceedings of PSSS, 2003, pp. 109–113.
[19] T. Neumann, G. Weikum, Scalable join processing on very large RDF graphs, in: Proceedings of SIGMOD, 2009, pp. 627–640.
[20] D.J. Abadi, A. Marcus, S.R. Madden, K. Hollenbach, Scalable semantic web data management using vertical partitioning, in: Proceedings of VLDB Endow, 2007, pp. 411–422.
[21] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, S. Manegold, Column-store support for RDF data management: not all swans are white, in: Proceedings of the VLDB Endow, 2008, pp. 1553–1563.
[22] J. Zhou, L. Ma, Q. Liu, L. Zhang, Y. Yu, Y. Pan, Minerva: a scalable OWL ontology storage and inference system, in: Proceedings of ASWC, 2006, pp. 429–443.
[23] J. Dolby, A. Fokoue, A. Kalyanpur, A. Kershenbaum, E. Schonberg, K. Srinivas, L. Ma, Scalable semantic retrieval through summarization and refinement, in: Proceedings of AAAI, 2007, p. 299.
[24] Oracle Semantic Technologies Center, Available at: 〈oracle.com/technology/tech/semantic_technologies〉 as of 2010-10-18.
[25] J.S.S. Narayanan, T. Kurc. DBOWL: Towards Extensional Queries on a Billion Statements Using Relational Databases, Technical Report, 2006, Available at: 〈bmi.osu.edu/resources/techreports/osubmi.tr.2006.n3.pdf〉 as of 2010-10-18.
[26] A. Kiryakov, D. Ognyanov, D. Manov. OWLIM—a pragmatic semantic repository for OWL, in: Proceedings of SSWS, 2005, pp. 182–192.