



time to avoid storing large scale inference data. Our aim is to improve performance of semantic repository and provide the possibility of managing large scale semantic data.

## 2 Relate Work

Along with more and more popular research of semantic web, many semantic repositories have been developed. All of them can be divided into three categories based on the persistent strategy they use: RDB (Relational Database)-based, File system-based and Memory-based. Because RDB has been fully studied these years, RDB-based systems are in the majority, like Sesame [1], DLDB-OWL [6], RStar [5] and so on. Sesame provides a general storage interface and implements storage method on MySQL, Oracle and so on. File system-based and memory-based storage methods have also been implemented. Sesame provides two logical storage models: RDF schema and RDFS schema. No inference is supported for RDF schema. For RDFS schema, user can use default inference function defined by Sesame, but this is limited to inference rules defined in RDFS. Moreover, user can also use self-defined inference rules, which makes Sesame have good extensibility. From RDF(S) to OWL, only the self-defined inference rules change. But from experiment, we can observe that large number of rules is needed to express complete OWL semantics and when loading data, performance is very bad for doing complete inference based on such rules. Therefore, it can't be used to manage large scale OWL data. DLDB-OWL uses MS Access as its persistent platform and uses inference engine FaCT. It declares high performance for large scale OWL data, but has limited inference ability. From experiment we can observe that DLDB-OWL cannot get any answers for some queries. OWLim [3] is a typical memory-based system. It supports more semantic rules than any other systems. OWLim uses Sesame's general storage interface and it has higher performance than Sesame's own memory-based storage module. To support persistent storage of semantic data, OWLim uses a simple file format, named "N-triples" and provides backup function. But when do query and inference processing, all data will be read from hard disk into memory. From experiment, we can observe that OWLim cannot handle OWL documents which size is larger than 100MB on general computer hardware. Because OWLim supports most of the semantic rules in OWL Lite, we use it as benchmark of query completeness in our experiment. To support large scale semantic data management, HStar is built on file system and do query and inference processing on physical storage model. Very small part of inference data is materialized and almost the same semantic rules are supported as OWLim. Only one query has less answers than OWLim when do test queries of Lehigh University Benchmark.

## 3 Data Model

To give better description of HStar's functions, we formalized data model of OWL supported by HStar. This data model has summarized most of the characters of OWL Lite. Our storage, inference and query processing strategies are all based on the data model.

D: all data in OWL document as format <subject property object>

$$L = \{C, P, I, R_C, R_P, R_{CP}, R_I, R_{CI}, T_P\};$$

$$C = \{URI_i | \exists \langle URI_i \text{ rdf:type owl:Class } \rangle \in D\};$$

$$P = \{URI_i | \exists \langle URI_i \text{ rdf:type owl:ObjectProperty } \rangle \vee \exists \langle URI_i \text{ rdf:type owl:DatatypeProperty } \rangle \in D\};$$

$$I = \{URI_i | URI_i \notin C \text{ and } URI_i \notin P\};$$

$$R_C = \{C_i \prec C_j | C_i, C_j \in C \wedge \exists \langle C_j \text{ rdfs:subClassOf } C_i \rangle \in D\} \cup \{C_i \equiv C_j | \exists \langle C_i \text{ owl:equivalentClass } C_j \rangle \in D\};$$

$$R_P = \{P_i \prec P_j | P_i, P_j \in P \wedge \langle P_j \text{ rdfs:subPropertyOf } P_i \rangle \in D\} \cup \{P_i \equiv P_j | \exists \langle P_i \text{ owl:equivalentProperty } P_j \rangle \in D\} \cup \{P_i \leftrightarrow P_j | \exists \langle P_i \text{ owl:inverseOf } P_j \rangle \in D\};$$

$$R_{CP} = \{[P_i, C_j] | \exists \langle P_i \text{ rdfs:domain } C_j \rangle \in D \vee \exists \langle P_i \text{ rdfs:range } C_j \rangle \in D\};$$

$$R_I = \{[URI_i, URI_j] | \exists \langle URI_i \text{ } P_x \text{ } URI_j \rangle, P_x \in P \in D\} \cup \{URI_i \equiv URI_j | \exists \langle URI_i \text{ owl:sameAs } URI_j \rangle \in D\};$$

$$R_{CI} = \{[URI_i, C_j] | \exists \langle URI_i \text{ rdf:type } C_j \rangle \in D\};$$

$$T_P = P_T \cup P_S \cup P_F \cup P_{IF};$$

$$P_T = \{P_i | P_i \in P \wedge \langle P_i \text{ rdf:type owl:TransitiveProperty } \rangle \in D\};$$

$$P_S = \{P_i | P_i \in P \wedge \langle P_i \text{ rdf:type owl:SymmetricProperty } \rangle \in D\};$$

$$P_F = \{P_i | P_i \in P \wedge \langle P_i \text{ rdf:type owl:FunctionalProperty } \rangle \in D\};$$

$$P_{IF} = \{P_i | P_i \in P \wedge \langle P_i \text{ rdf:type owl:InverseFunctionalProperty } \rangle \in D\};$$

OWL data has been divided into three categories by this data model: one consists of C, P, I, which respectively represent OWL Class, OWL Property and Individual Resource; one consists of  $R_C$ ,  $R_P$ ,  $R_I$ ,  $R_{CP}$ ,  $R_{CI}$ , which respectively represent relation of elements in C, relation of elements in P, relation of elements in I, relation between elements in C and elements in P, relation between elements in C and elements in I; the last one is  $T_P$ , which represents characters defined on OWL Property, including transitive  $P_T$ , symmetric  $P_S$ , functional  $P_F$  and inverse functional  $P_{IF}$ .  $C$ ,  $P$ ,  $R_C$ ,  $R_P$ ,  $R_{CP}$  and  $T_P$  are used to define Ontology and they are always stable. Most of OWL data focus on  $R_I$ , which use Ontology to describe type and relation information of elements in I. Completeness of inference includes two aspects: one is to get complete relation of  $R_C$  and  $R_P$ , the other is to get complete relation of  $R_I$  and  $R_{CI}$ . The former represents complete ontology and the latter represents complete ontology instances.

## 4 Analysis of Inference Completeness

We mentioned above that inference completeness is to get complete  $R_C$ ,  $R_P$ ,  $R_I$  and  $R_{CI}$ . Below we give a detailed discussion for them respectively:

### 4.1 Completeness Analysis of $R_C$ , $R_P$

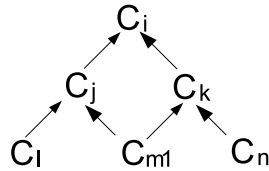
Elements in  $R_C$  have two relations:  $C_i \prec C_j$  represents inheritance;  $C_i \equiv C_j$  represents equivalence. Inheritance is transitive. Equivalence is transitive and symmetric. Because equivalence affects inheritance, complete equivalence relation should be computed first. Complete  $R_C$  should satisfy:

- (a)  $\forall C_i \in C$ , can get all  $\{C_j | C_j \in C \wedge \exists (\text{explicit or implicit}) C_i \equiv C_j\}$ ;
- (b)  $\forall C_i \in C$ , can get all  $\{C_j | C_j \in C \wedge \exists (\text{explicit or implicit}) C_j \prec C_i\}$ ;
- (c)  $\forall C_i \in C$ , can get all  $\{C_j | C_j \in C \wedge \exists (\text{explicit or implicit}) C_i \prec C_j\}$ ;

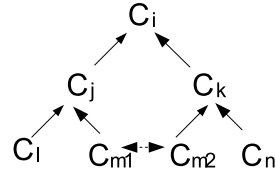
There are three methods to guarantee requirement (a): The first is to store all explicit  $C_i \equiv C_j$  and get all relevant  $\{C_i \equiv C_j\}$  to construct equivalent set when query; The second is to store all explicit and implicit  $C_i \equiv C_j$ . There will be no implicit data left and equivalent set does not need to be built. The third is to store equivalent set directly on hard disk. The second method uses redundancy data to improve search performance but adds maintenance cost. The third method not only avoids redundancy data, but also can get equivalent set directly. It is suitable for managing large equivalence relation. In general, equivalence relation in OWL is quite few. So HStar adopts the first method.

For inheritance relation  $C_i \prec C_j$ , transitive character makes  $C_i \prec C_j \prec C_k \Rightarrow C_i \prec C_k$ . Requirement (b) and (c) are all related with it. There are also two methods for these two requirements: One is for every transitive chain, compute all implicit inheritance relation and put them into storage system. E.g. if use  $C_i \leftarrow C_j$  represents  $C_i \prec C_j$  and suppose there are inheritance relations in fig.1.

There are four transitive chains in fig.1:  $C_i \prec C_j \prec C_l$ ,  $C_i \prec C_j \prec C_m$ ,  $C_i \prec C_k \prec C_m$ ,  $C_i \prec C_k \prec C_n$ . We can compute three implicit inheritance relations from these chains:  $C_i \prec C_l$ ,  $C_i \prec C_m$ ,  $C_i \prec C_n$ . For large inheritance relation, such method will produce too much redundancy data. Computation complexity is  $O(n^2)$  ( $n$  is the number of elements in inheritance relations) and it will be a hard work to maintain the redundancy data. The other method is using tree storage structure to represent inheritance relations.



**Fig. 1.** An example of inheritance relation



**Fig. 2.** Tree structure for Fig.1

Node  $C_m$  in fig.1 splits into nodes  $C_{m1}$  and  $C_{m2}$ . Node  $C_{m1}$  copies all information of  $C_m$  and node  $C_{m2}$  is a reference of  $C_{m1}$ . If it is required to find all  $C_x$ , which satisfy  $C_x \prec C_m$ , first locate node  $C_{m1}$  in Fig.2, get all ancestors of  $C_{m1}$ , i.e.  $\{C_i, C_j\}$ , and then get all ancestors of  $C_{m2}$ , i.e.  $\{C_i, C_k\}$ , union the two result sets and remove the duplicate, we can get  $\{C_i, C_j, C_k\}$ . Such method avoids computing redundancy data, but needs native tree storage on hard disk. HStar adopts this method.

Inheritance relation and equivalence relation defined in  $R_P$  are same as those in  $R_C$  in essence. HStar uses same method to deal with them. Besides these, there is another relation defined, i.e.  $\{P_i \leftrightarrow P_j | \langle P_i \text{ owl:inverseOf } P_j \rangle \in D\}$ . This relation will only

bring implicit data in  $R_I$  according to OWL semantic definition. So we will discuss it later.

## 4.2 Completeness Analysis of $R_I$

From definition of  $R_I$ , we can see there are two sub-relations in it. We give their definitions below:

$$\begin{aligned} R_{I1} &= \{[URI_i, URI_j] | \exists \langle URI_i P_x URI_j \rangle \in D\} \\ R_{I2} &= \{URI_i \equiv URI_j | \exists \langle URI_i owl:sameAs URI_j \rangle \in D\} \end{aligned}$$

$R_{I2}$  defines equivalence relation which affects completeness of  $R_{I1}$ , just as equivalence relation in  $R_C$  does. Besides that user can directly define  $R_{I2}$ , property that is element of  $P_F$  or  $P_{IF}$  can also infer  $R_{I2}$  relation. The inference rules are defined below:

$$\begin{aligned} \langle URI_i P_x URI_k \rangle \wedge \langle URI_j P_x URI_k \rangle \wedge P_x \in P_{IF} &\Rightarrow URI_i \equiv URI_j \\ \langle URI_k P_x URI_i \rangle \wedge \langle URI_k P_x URI_j \rangle \wedge P_x \in P_F &\Rightarrow URI_i \equiv URI_j \end{aligned}$$

So complete  $R_{I2}$  needs to apply rules above to every element in  $P_F$  and  $P_{IF}$ . And the process needs to do iteratively. E.g. suppose  $P_x \in P_F$ ,  $a, b, c, d$  respectively represent an URI and  $\exists \{\langle a P_x b \rangle, \langle a P_x c \rangle, \langle b P_x d \rangle, \langle c P_x a \rangle\} \in D$ . According to rules above, we can get  $\langle a P_x b \rangle \wedge \langle a P_x c \rangle \Rightarrow b \equiv c$ . But the process cannot terminate now, because  $b \equiv c$  also affects existed data. With this consideration, we can get  $\langle b P_x d \rangle \wedge \langle c P_x a \rangle \Rightarrow d \equiv a$ . The process needs to do iteratively until no new equivalence relations are generated. Storage method of  $R_{I2}$  is the same as equivalence relation of  $R_C$ .

Completeness of  $R_{I1}$  is mainly determined by characteristic of  $P_x$ . If  $P_x \in P_T$  or  $P_x \in P_S$  or  $P_x$  has inheritance or equivalence relation in  $R_p$ , it will bring implicit data into  $R_{I1}$ . If there exist  $P_x$  satisfying  $P_x \in P_T \wedge P_x \in P_S$ , we treat such  $P_x$  as an equivalent relation.

There is a condition that is not defined definitely in OWL semantic. If  $\{P_x \prec P_y \in R_p \text{ or } P_y \prec P_x \in R_p\}$  and  $\{P_x \in P_T \text{ or } P_x \in P_S \text{ or } P_x \in P_F \text{ or } P_x \in P_{IF}\}$ , whether  $P_y \in P_T$  or  $P_y \in P_S$  or  $P_y \in P_F$  or  $P_y \in P_{IF}$  is not defined. So HStar does not consider the interaction effect between  $R_p$  and  $T_p$ .

Under the precondition above, completeness of  $R_{I1}$  can be considered from  $P_T$ ,  $P_S$  and  $R_P$  respectively:

1.  $P_T$  defines transitive character which is equivalent to inheritance relation of  $R_C$  in essence. HStar adopts the same method to deal with  $P_T$ .
2.  $P_S$  defines symmetric relation and related rule in OWL is  $P_x \in P_S \wedge \langle URI_i P_x URI_j \rangle \Rightarrow \langle URI_j P_x URI_i \rangle$ . Two methods can guarantee the completeness of  $P_S$ : One is to store all implicit data brought by  $P_S$ . E.g. when user inserts  $\langle URI_i P_x URI_j \rangle$ , both  $\langle URI_i P_x URI_j \rangle$  and  $\langle URI_j P_x URI_i \rangle$  will be stored. There is no need to consider  $P_S$  character when

query with this method. But the volume of such data will be doubled. The other method only stores the explicit data and use query rewriting to satisfy  $P_S$  requirement. E.g. suppose  $P_x \in P_S$ , query  $\langle URI_i P_x ? \rangle$  should be rewritten as  $\langle URI_i P_x ? \rangle$  and  $\langle ? P_x URI_i \rangle$ . When data volume that has  $P_S$  character become larger, performance of the second method will be better than the first one.

3. Rule  $P_i \prec P_j \wedge \langle URI_x P_j URI_y \rangle \Rightarrow \langle URI_x P_i URI_y \rangle$  makes  $R_p$  may bring implicit data. Considering query  $\langle URI_x P_i ? \rangle$ , if there is only  $\langle URI_x P_j URI_y \rangle$  in  $R_I$ , no result will be returned if don't use rule above. As we have mentioned in section 4.1, relation  $P_i \prec P_j$  in  $R_p$  is stored as a tree structure in HStar. For any  $P_i$  in this structure, all its descendants can be accessed directly. So when processing query  $\langle URI_x P_i ? \rangle$ , HStar will search all data in  $D$  which have  $P_i$  or  $P_i$ 's descendants as their Property. Special storage design in HStar makes such operation can be processed efficiently. We will give detailed analysis in section 5. Relation  $\{P_i \leftrightarrow P_j \mid \langle P_i \text{ owl:inverseOf } P_j \rangle \in D\}$ , which is defined in  $R_p$ , has rule  $\langle P_i \text{ owl:inverseOf } P_j \rangle \wedge \langle URI_x P_i URI_y \rangle \Rightarrow \langle URI_y P_j URI_x \rangle$  defined in OWL semantic. Like  $P_s$ , completely materializing implicit data brought by this rule will double such data volume. Query rewriting can also be used here and its performance will be better when data volume is larger.

### 4.3 Completeness Analysis of $R_{CI}$

$R_{CI}$  describes type information of URI and it is the most complex part of OWL data. Both  $R_C$  and  $R_{CP}$  affect completeness of  $R_{CI}$  and the related rules are list below:

1.  $\langle URI_x \text{ rdf:type } C_j \rangle \wedge C_i \prec C_j \Rightarrow \langle URI_x \text{ rdf:type } C_i \rangle$
2.  $\langle P_x \text{ rdfs:domain } C_y \rangle \wedge \langle URI_i P_x URI_j \rangle \Rightarrow \langle URI_i \text{ rdf:type } C_y \rangle$
3.  $\langle P_x \text{ rdfs:range } C_y \rangle \wedge \langle URI_i P_x URI_j \rangle \Rightarrow \langle URI_j \text{ rdf:type } C_y \rangle$

As we have mentioned in section 4.1, relation  $C_i \prec C_j$  is stored as a tree structure in HStar. When processing query  $\langle URI_x \text{ rdf:type } ? \rangle$ , we first get  $C_i$  if there is explicit data  $\langle URI_x \text{ rdf:type } C_i \rangle$  in  $D$ ; then get all ancestors of  $C_i$  and return them as the result. For rules 2 and 3, if we don't get complete  $R_{CI}$  relation when loading OWL documents, the whole data space search will be required when query processing. HStar materializes all implicit data brought by rules 2 and 3.

From discussion above, we can observe that HStar only materializes implicit data brought by  $P_F$  and  $P_{IF}$ , implicit data in  $R_{CI}$  brought by Property's domain and range.

## 5 Storage Design

From the third section, we can see that the main part of OWL data is five kinds of relations,  $R_C$ ,  $R_P$ ,  $R_{CP}$ ,  $R_I$  and  $R_{CI}$ . How to organize these relations on hard disk is the task of storage design. Considering the characteristics of both OWL data and queries against it, we designed a special storage model for OWL data, which is built on file system rather than RDB, ORDB and etc. In the rest of this section, we will first describe the inner identifier of entities, and then present the storage method of different relations.

### 5.1 Inner Identifier for Entities: OID

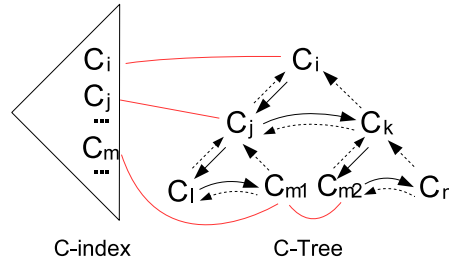
In OWL data, entities are identified by URI, which is usually a long string. Storing original URI takes considerable space; therefore we use inner identifier OID to replace URI in storage. OID consists of two members: *id*, which occupies four bytes, *flag*, which occupies one byte, indicates whether entity has equivalent resources. Thus an OID totally occupies five bytes, which is much smaller than a URI. The relationship between OID and URI is one to one and is saved in two global hash tables.

### 5.2 Storage of $R_C$ and $R_{CI}$

Inheritance relation in  $R_C$  is stored in tree structure. We named it C-Tree. E.g. the relation in fig.2 is stored as C-Tree structure in fig.3. Each tree node keeps addresses (represented by page number and offset in physical page) of its first child, parent, left and right siblings. It is easy to access the ancestors and descendants of a tree node by these addresses.

Non-tree nodes in inheritance relation graph split into multiple copies. One is primary (P-Node), and the others are references. E.g. node  $C_m$  has been divided into  $C_{m1}, C_{m2}$ . They are linked in the Same Entity List (SE-List), with the primary one as head. Only primary node stores the address of child and Individual List.

Locating arbitrary  $C_x$  in C-Tree structure is an indispensable operation for inheritance relation query. C-index is built to improve the performance of this operation, which is a B+ tree structure and uses identifiers of P-Nodes as keys. As showed in fig.3, C-index record addresses of nodes in C-Tree. Using C-index and SE-List, all the nodes responding to  $C_x$  in C-Tree can be accessed quickly.



**Fig. 3.** C-Tree and C-Index for  $R_C$  Storage

Equivalence relation in  $R_C$  is stored in B+ tree. E.g. suppose  $C_i$  is equivalent to  $C_j$  and the id of  $C_i$ 's OID is smaller than the id of  $C_j$ 's OID, and then take  $C_i$  as key and  $C_j$  as value. Only explicit equivalence relations are stored. Equivalence sets are built in memory to facilitate query processing, each set corresponding to a memory list. Updating equivalence relation needs to maintain both B+ tree and lists in memory.

Individuals related to the same  $C_x$  are stored in one Individual List (I-List), whose start address is saved in  $C_x$ 's P-Node of C-Tree. E.g. in fig.4, individuals  $I_i$  and  $I_j$  have

type of  $C_m$ . They are stored in an I-List, with the start address kept in node  $C_{m1}$  of C-Tree. This structure is to facilitate querying individuals of given Class, which is the most frequent query about  $R_{CI}$ .

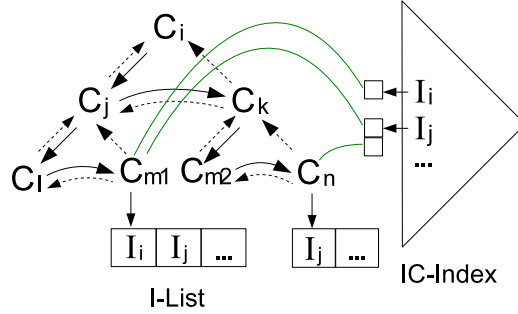


Fig. 4. I-List and IC-index for  $R_{CI}$  Storage

Queries for type of given individual are less frequent but necessary. IC-index is built to facilitate these queries. It is a B+ tree index, which uses OID of individual as key. Leaf node contains all the Classes to which the individual belongs. E.g. IC-index in fig.4 records that individual  $I_i$  belongs to  $C_m$  and  $I_j$  belongs to  $C_m$  and  $C_n$ .

Only explicit  $R_{CI}$  relations are stored in I-List and IC-index. To guarantee the inference completeness, we need to combine I-List and IC-index with C-Tree structure. That is the reason why we store addresses of I-Lists in P-Nodes of C-Tree. E.g. in fig.4, to find individuals of  $C_i$ , I-List of both  $C_i$  and its descendants need to be returned. Here,  $I_i, I_j$  is the result. To query type of  $I_i$ , we find  $C_{m1}$  through IC-index, then  $C_m$  and its ancestors are returned. Here,  $C_i, C_j, C_k, C_m$  is the result.

### 5.3 Storage of $R_p, R_{CP}, R_I$ and $T_p$

Inheritance relation and equivalence relation in  $R_p$  are stored in the same way as those relations in  $R_C$ . P-Tree, and P-index are built as C-Tree and C-index. Inverse relations in  $R_p$  are stored as data members of P-Nodes in P-Tree (Property Tree); for  $P_i \leftrightarrow P_j$ , store  $P_j$  in  $P_i$ 's P-Node, and store  $P_i$  in  $P_j$ 's P-Node.

$T_p$  and  $R_{CP}$  are also stored as data members of P-Nodes.  $T_p$  is represented by a byte and the first four bits are used to indicate whether  $P_x$  has transitive, symmetric, function and inverse-function characters.  $R_{CP}$  is stored as two arrays, which store entities having  $rdfs: domain$  or  $rdfs: range$  relation with  $P_x$ .

Equivalence relation in  $R_I$  (namely  $R_{I2}$  in section 4.2) is stored as same as that relation in  $R_C$ . Individual pairs of  $R_{I1}$ , which are related to same transitive  $P_x$ , are stored in one Individual Tree (I-Tree). I-Tree adopts the same structure as C-Tree, including I-index and SE-List structures. E.g. in fig.5,  $P_n$  is transitive. Pairs  $(I_k, I_m), (I_k, I_n)$  relate to  $P_n$ , and are stored in its I-Tree. Individual pairs related to same non-transitive  $P_x$  are stored in two Individual B+ trees (IB-Tree). One is SB-Tree (S-Key B+ tree),



taking subject as key. The other is OB-Tree (O-Key B+ tree), taking object as key. E.g. in fig.5,  $P_m$  is not transitive. Pair  $(I_i, I_j)$  relates to  $P_m$  and is stored in its IB-Trees. SB-Tree takes  $I_i$  as key and OB-Tree takes  $I_j$  as key. The root addresses of I-Tree and IB-Trees are kept in  $P_x$ 's P-Node.

IP-index is built similarly to IC-index. The difference is that IP-index records how an individual relates to different properties (as subject or object). E.g. the IP-index in fig.5 records that  $I_i$  relates to  $P_m$  as subject (represented by solid lines),  $I_j$  relates to  $P_m$  as object (represented by dashed line), and so on.

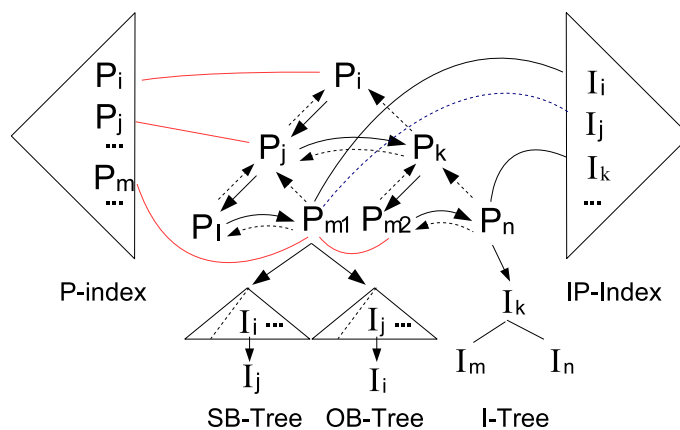


Fig. 5. Storage of  $R_p$  and  $R_{I1}$

Queries against  $R_{I1}$  can be processed in a similar way with  $R_{CI}$ . The difference is that queries against  $R_{I1}$  may need further search in  $P_x$ 's I-Tree or IB-Tree. For transitive  $P_x$ , search in I-Tree in the same way as in C-Tree. For non-transitive  $P_x$ , search in SB-Tree with given subject, or in OB-Tree with given object.

## 6 Query Processing

HStar supports queries in SPARQL language, which is proposed by W3C and likely to be the standard query language for OWL. When we mention "OWL query" later, it means SPARQL query. Here we give a query example, which queries all the facts related to "students take courses".

```

PREFIX p:<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl>
SELECT ?x ?y
WHERE {
    ?y rdf:type p:Course.
    ?x rdf:type p:Student.
    ?x p:takesCourse ?y
}
    
```

We call triple with variable(s) "Query Triple", QT for short. E.g. "?y rdf:type p:Course" in the query example is a QT, in which "?y" represents variable to be evaluated during

query processing. From query example above, we can observe that QT is the basic unit in OWL query. Query processor first evaluate all QTs to get middle results and then choose some order to join all middle results to get final results. Different join orders produce different sizes of middle results and this affects query performance. Such problem has also been encountered in SQL query processing. For OWL data is different from data in relational database, new solution needs to be proposed. In the next section, we give our intuition for this problem and describe several possible solutions that can be used for OWL query optimization.

## 6.1 Query Optimization

### 1. Remove possible redundant QTs based on Ontology.

$R_{CP}$  is part of Ontology and it defines domain and range of a Property. Rules

$$\begin{aligned} &\langle P_x.rdf : domainC_y \rangle \wedge \langle URI_i P_x URI_j \rangle \Rightarrow \langle URI_i rdf:type C_y \rangle \text{ and} \\ &\langle P_x.rdf : rangeC_y \rangle \wedge \langle URI_i P_x URI_j \rangle \Rightarrow \langle URI_j rdf:type C_y \rangle \end{aligned}$$

have been mentioned in section 4.3. These rules not only bring implicit data, but also define restrictions. That means if there is  $\langle URI_i P_x URI_j \rangle$  in  $D$  and  $P_x$  has domain  $C_m$ , has range  $C_n$ , then  $URI_i$  must be an instance of Class  $C_m$  and  $URI_j$  must be an instance of Class  $C_n$ . We can make full use of such restrictions to optimize some type of queries. E.g. suppose there are properties “StudentNumber”, “Teach” and two disjoint classes “Student”, “Teacher”. We know only Class “Student” can have “StudentNumber” and only Class “Teacher” can do “Teach” in real world. These facts will be defined by  $R_{cp}$ . Now if user issues query  $\langle ?s Student ?n \rangle \langle ?s Teach ?c \rangle$ , we can immediately judge that such query has no result because “Student” can not “Teach” and “Teacher” has no “StudentNumber”. Another example is that if user issues query  $\langle ?s rdf:type Student \rangle \langle ?s StudentNumber ?n \rangle$ , we can remove QT  $\langle ?s rdf:type Student \rangle$  because only “Student” has “StudentNumber”.

### 2. Choose Join order based on statistic data.

Choosing join order needs a method to estimate mid-result size of two QTs’ join. E.g. query  $\langle s p_1 ?x \rangle, \langle ?x p_2 ?y \rangle, \langle ?y p_3 o \rangle$  contains three QTs. There are two possible join orders:  $(\langle s p_1 ?x \rangle \text{ join } \langle ?x p_2 ?y \rangle) \text{ join } \langle ?y p_3 o \rangle$  or  $\langle s p_1 ?x \rangle \text{ join } (\langle ?x p_2 ?y \rangle \text{ join } \langle ?y p_3 o \rangle)$ . If we can estimate middle results’ size of  $(\langle s p_1 ?x \rangle \text{ join } \langle ?x p_2 ?y \rangle)$  and  $(\langle ?x p_2 ?y \rangle \text{ join } \langle ?y p_3 o \rangle)$ , then we can choose the join order which has smaller middle result size. Here we suggest borrowing idea for such problem from relational database. When loading data into HStar, we can compute how many triples there are for every Property, we named this number as  $N_{tp}$ ; and compute how many different instances there are for every Property’s subject and object, we named the two numbers as  $N_{sp}$  and  $N_{op}$ , then the middle result size of  $\langle s p_1 ?x \rangle \text{ join } \langle ?x p_2 ?y \rangle$  can be computed by  $\min\{N_{tp1}/N_{op1}, N_{tp2}/N_{sp2}\}$ .

## 7 Experiment

Experiments in [2] give detailed compare among semantic repositories, DLDB-OWL [6], Sesame-DB [1], Sesame-Memory [1] and OWLJessKB [4]. The experiments test performance of data loading, query processing and query completeness. In our experiment, we do test on systems DLDB-OWL, Sesame-DB, OWLim [2] and HStar. OWLim is a memory-based system, implemented under Sesame general architecture and has better performance on data loading, query processing and query completeness than Sesame’s original memory-based system. OWLJessKB [4] is also a memory-based system. [2] points out that it has implemented incorrect inference strategy. So OWLim can be treated as the best memory-based system and we ignore Sesame-Memory and OWLJessKB system in experiment.

Our experiment uses an extension of Lehigh University Benchmark, which has been described in [2]. Four test data sets are generated by tool provided by [2]. They are univer1, univer5, univer10 and univer20. The smallest data set is 8MB including 15 OWL documents. The largest is 218MB including 402 OWL documents. We get "Out-OfMemory" error when loading univer10 into OWLim system. Sesame-DB uses user-defined inference rules and costs about 13 hours to load univer5. "OutOfMemory" error occurred when loading univer20 into HStar for a memory-based hash map is used. This will be improved in the next version. DLDB-OWL costs more than 13 hours to load univer10, but it still doesn’t finish loading work, which is different from that discussed in [2]. So we just give out the test result for first three data sets.

### 7.1 Experiment Environment

Hardware: CPU P4.3G, 512MB of RAM, 40GB of hard disk; Software: Windows XP, Java JDK1.5, MySQL4.1.4, MS Access2003, DLDB-OWL(04-03-29 release), Sesame(1.2.2), OWLim(2.8). For all test systems, we set maximum heap size as 256MB.

	Data set	Instance number	Load time(ms)	Repository size(KB)
OWLim	LUBM(1, 0)	103,074	2,985	17,311
Sesame-DB			1,206,141	48,333
HStar			98,641	19,922
DLDB-OWL			183,937	15,876
OWLim	LUBM(5, 0)	645,649	47,578	107,809
Sesame-DB			47,131,655	283,967
HStar			982,875	77,082
DLDB-OWL			994,157	89,156
OWLim	LUBM(10,0)	1,316,322	-	-
Sesame-DB			-	-
HStar			2,135,453	154,656
DLDB-OWL			-	-

**Table 1.** Description of test data sets and data loading performance

From left of fig.6, we can observe that OWLim has the best data loading performance for the first two data sets. HStar has almost the same performance with DLDB-OWL. Sesame-DB has the worst performance. [2] points out that Sesame-DB constructs dependent relation among OWL data elements when loading data. This is very time consumed but is very useful for update performance. DLDB-OWL doesn't consider update problem. HStar just materialize a little part of inference data and it's easy to maintain their relation.

[2] gives 14 query test cases. They are used to test query performance and query completeness. In our experiment, OWLim supports the most semantic rules and we use OWLim query answers as benchmark to evaluate other systems' query completeness.

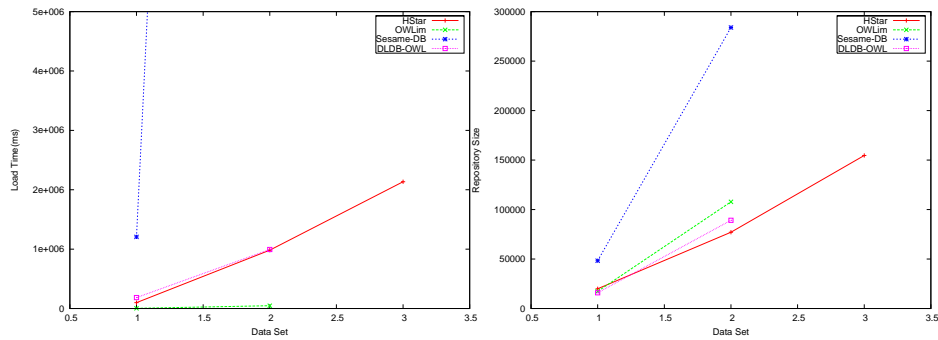


Fig. 6. Data loading performance and repository size

From fig.7, we can observe that HStar has different answers with OWLim only for the 12th query. DLDB-OWL has no answers for the 11, 12, 13th queries. Sesame-DB has incompleteness answers for the 6, 7, 8, 9th queries and has no answers for the 10, 12th queries. We can sort them by answer completeness as below: OWLim > HStar > DLDB-OWL > Sesame-DB.

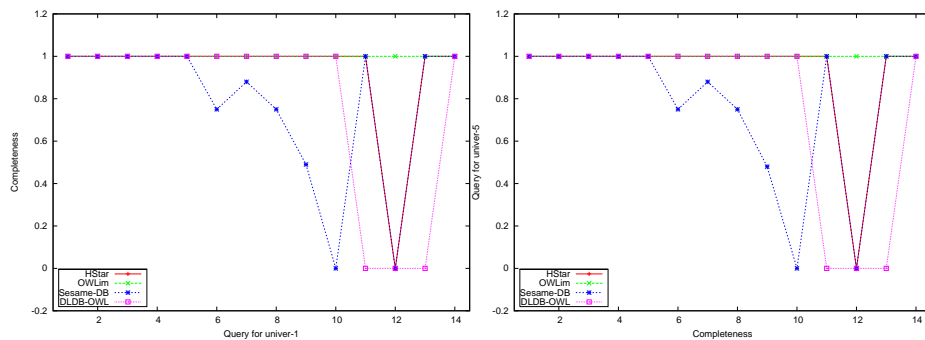


Fig. 7. Query completeness



