Query Processing in a Multidatabase System with an integrated view based on Description Logics

Goñi A. , Illarramendi A. , Mena E. , Blanco J.M.

Facultad de Informática, Universidad del País Vasco.
Apdo. 649, 20.080 San Sebastián. SPAIN
e-mail: jibgosaa@si.ehu.es

Abstract

It is widely accepted the increase need of organizations to manage data stored in distributed heterogeneous information sources. For this new situation flexible query answering systems are required. In this paper we present the main features of a query answering component defined for a multidatabase system. Queries can be formulated over an integrated schema, defined using a system based on Description Logic, and their answers must be found on the underlying databases. The query answering component provides new approaches for semantic and caching optimization techniques and the opportunity of giving intensional as well as extensional answers.

1 Introduction

The integration of heterogeneous and autonomous information sources is a requirement for the new type of cooperative information systems. Multidatabase systems have been proposed as a solution to work with different pre-existing autonomous databases. Federated database systems are a special type of multidatabase systems where an integrated schema is provided. Three different types of problems are involved when building a Federated Database System (FDBS): *translation* of the underlying database schemata into schemata expressed in a canonical model, *integration* of the translated schemata into an integrated schema and *query processing* of the user-formulated queries over the integrated schema by accessing the underlying databases. Although there has been a lot of research about the problems of translation and integration of schemata to obtain integrated ones, the problem of query processing against these integrated schemata has not been treated so much.

There exist many different approaches for building a FDBS, namely the Entity-Relationship model approach [?, ?], the Object-Oriented approach [?, ?], and the Knowledge Representation Systems (KRS) approach [?, ?]. In our case we have built a FDBS that integrates several heterogeneous relational databases[1] by using a particular type of KRS based on Description Logics[2](DL system). The integrated schema is represented by a knowledge base built upon the different relational database schemata, and the extension of the knowledge base (the instances of the classes and attribute values) is in fact in the underlying databases. Therefore, the answer for the queries formulated over the integrated schema must be found in the underlying databases. In this context,

[1] Although they all use the same data model, semantic heterogeneity still can remain.
[2] Also known as Terminological Logics or based on KL-ONE [?, ?].

it is necessary to use optimization techniques in order to improve the performance of the query processing task. DL systems provide interesting features for developing *semantic and caching query optimization* techniques and also for providing *intensional answers*.

In the rest of the paper we present first an introduction to DL systems, then we introduce briefly the work's framework with the system architecture of our FDBS, the main features of *the mapping information* and an example of one integrated schema. Finally we explain with more detail the query processing aspects of our FDBS.

2 A brief introduction to DL systems

The family of DL systems has its origin on the semantic network research area, however they concentrate more on the description of objects than on the representation of sentences. In our case, we use a DL system for building a FDBS. The integrated schema will be represented as a hierarchy of classes and attributes. Two types of class descriptions can appear in the hierarchy: *primitive* classes that are phrased in terms of necessary conditions that the instances verify and *defined* classes that express not only necessary conditions but also sufficient. The types of conditions are value restrictions and cardinality restrictions over attributes and other classes. As in an object-oriented system, every instance of a class has its own object identifier (OID).

For example:

The class *person* with attributes *name*, *age* and *children* can be expressed as a primitive class:

person :< *anything and exactly(1,name) and exactly(1,age)*

meaning that any instance of *person* is an instance of *anything*[3] with one and only one value for the attributes *name* and *age*.

The classes *youth* and *parent* can be expressed as defined classes in the following way:

youth := *person and all(age,lt(30))*

meaning that an instance of *youth* is an instance of *person* with age less than 30, and that any person aged less than 30 is a youth,

parent := *person and atleast(1,children)*

meaning that parents are persons with at least one children and persons with at least one children are parents.

Moreover, two important features in DL systems are the notions of *subsumption* and *classification*. One class *subsumes* another one if in all possible circumstances, any instance of the second one must be in the first one. In a DL system, it is possible to know whether one class *is subsumed* by another one simply by looking at the definition of the classes, without accessing to the instances. The *classification* mechanism consists of discovering the subsumption relationships between classes when a new class is declared, i.e., the new class is automatically located into the hierarchy of classes, therefore classes viewed as composite descriptions, can be reasoned with and are the source of inferences.

For example, if the following defined class *teenager* is described:

teenager := *person and all(age,gt(14)) and all(age,lt(19))*

meaning that teenagers are persons aged between 14 and 19 and nothing more, one could reason that a teenager must be a youth although it has not said in its description. In fact a DL system detects that *youth* subsumes *teenager* and therefore *teenager* is classified under *youth* in the class hierarchy.

Furthermore, since a query is just a definition of the required properties to be satisfied by the instances listed in the answer, class descriptions can be used for information re-

[3] *Anything* is a special class such that any instance can belong to it.

trieval as queries [?]. The general form of a query has a part with projection of attributes *[rf(attribute1),...,rf(attributeN)]* that may be empty, meaning that only OIDs are desired, and another part that is the description of a class *getall class_description*, that is an intersection of class names and cardinality and value restrictions of attributes.

For example, the query

[self,rf(name)] for getall person and all(age,le(17))

is asking for the instances and names of all the persons with age less or equal than 17.

Therefore the notion of subsumption between *classes* can also be used with queries: *some queries or classes subsume other queries or classes*. For example, the class description of the previous query is subsumed by the class *youth*. This feature is used to verify whether data are cached or not when a query is formulated.

For the query processing task, DL systems provide some interesting features for developing *semantic and caching query optimization* techniques and also for providing *intensional answers* as it will be shown in section 4.

3 Work's framework

Before focusing on the query processing we show the system architecture with its different components and explain what the mapping information is. This is necessary to understand how the Query Processor can answer the user queries over the integrated schema by accessing different underlying relational databases. Moreover, we also present an example of an integrated schema that will be used throughout section 4 to illustrate query processing aspects.

3.1 System Architecture

For the implementation of our FDBS we use the Client/Server approach in such a way that there exists a Client application dealing with the knowledge base that represents the integrated schema built upon the different relational databases and several Server applications, one for each database that participates in the multidatabase system. This implementation allows for a parallel processing over the different databases. Usually the Client and Server applications will be on geographically dispersed nodes (however, this is not mandatory). In figure 1 the architecture of the FDBS is shown.

In this architecture four main components can be distinguished: Translator, Integrator, Monitor and Query Processor.

1. The *Translator* component produces a knowledge base schema from a conceptual schema (or a subset of it, called exported schema) of a component database. The resultant knowledge base schema will be semantically richer than the source schema, therefore this component has to capture, with the Person Responsible for the Integration (PRI)'s help, semantics that are not expressed explicitly.

2. The *Integrator* component produces a integrated schema by integrating a set of knowledge base schemata previously obtained by the Translator component. During the integration process a set of correspondences between data elements of the knowledge base schemata that must be integrated will be defined by the PRI and new ones can also be deduced by the system.

3. The *Monitor* component responds automatically, i.e., without user intervention, to design changes made in the schema of a component database that affect the integrated schema

Figure 1: Architecture of the FDBS

[?]. This component is formed by three kind of processors: the Modifications Detector, the Modifications Manager and the Consistency Re-establisher and by the System Consistency Catalog.

4. The *Query Processor* component obtains the answer to the user formulated queries over the integrated schema by accessing the databases. This component has two kind of modules: the Global Query Processor and the Local Query Processor.

3.2 Mapping Information

The *mapping information* is the linking information that relates the DL objects in the integrated schema (classes and attributes) with the relational objects in the underlying databases. This mapping information has to be generated by the Translator and the Integrator components. A formal definition of the mapping information is given in [?] but in a simplified way it can be stated as follows:

- The mapping information of a class C is a pair <attr,rel> where *rel* is a derived relation expressed in the *extended relational algebra* (*ERA expression*) and *attr* is some attribute of *rel*. There is an instance with its own OID for each different value that the attribute/s *attr* takes in the ERA expression *rel*.

  For example. Let us suppose that there exists the table `student(id,name,age)`. The mapping information for the class *student* obtained from the previous relation could be <*id,student*> meaning that there exists an instance of the class *student* for each different value that the attribute `id` takes in the table `student`.

- The mapping information of an attribute A is a triple <attr_inst,attr_attr,rel> where *rel* is also an ERA expression and *attr_attr* contains the different values that the attribute A takes for each instance represented by *attr_inst*.

  For example. Supposing now that there exist the tables `student(id,name,age)` and `has_child(idp,idc)`, an attribute *children* for the class *student* can be defined. The mapping information for the attribute *children* could be <*idp,idc,has_child*> meaning that the different values taken by `idc` for the same value of `idp` are the *children* corresponding to the *student* represented by that `idp` value.

3.3 Example of an integrated schema

Let us suppose that there are two very simple exported schemata, namely *db1* and *db2*, from two databases with information about teachers, students and their children.

db1	db2
student(id,name,age)	teacher(id,name,age)
has_child(idp,idc)	has_child(idp,idc)

Figure 2: Simplified schemata

The Translator and the Integrator have obtained the primitive classes *person, teacher* and *student* with attributes *name, age* and *children*. And also the defined classes *youth, teaching_assistant, parent* and *super_parent*.

person :< *anything*
student :< *person*
teacher :< *person*
youth := *person and all(age,lt(30))*
teaching_assistant := *teacher and student*
parent := *person and atleast(1,children)*
super_parent := *person and atleast(5,children)*

Figure 3: Integrated schema

The mapping information associated to some of these classes and attributes appears in the next one:

CLASS	$<attr,rel>$
person	$<idp,db1.student \cup_{(id)} db2.teacher>$
youth	$<idp,\sigma_{age<30}\ (db1.student \cup_{(id)} db2.teacher)>$
teaching_assistant	$<idp,db1.student \cap_{(id)} db2.teacher>$
super_parent	$<idp,\sigma_{new\_attr\geq 5}\ ((idp\ \mathcal{F}_{count(idc)}\ (db1.has\_child \cup_{(idp)} db2.has\_child)))>$
ATTRIBUTE	$<attr\_inst,attr\_attr,rel>$
children	$<idp,idc,db1.has\_child \cup_{(idp)} db2.has\_child>$

As it can be shown, the derived relations that appear in these mapping informations are multidatabase ERA expressions because they contain aggregate functions ($\mathcal{F}_{count()}$) and attributes and relations are from different databases (i.e. *db1.student, db2.teacher*).

4 Query Processing

In general, the query processing task is carried out with two different kinds of processors: the Global Query processor and the Local Query processor. The subgoals of the former one are to make a global query optimization; to decompose a query into subqueries that will run over different databases and to generate an optimal plan to build the answer. The subgoals of the last one are to make local optimizations; to find the answers for the subqueries; and last, to send the answers to the Global Query process when is needed.

The different tasks made by the Global Query Processor are (see figure 4):

1. parsing of the query,
2. semantic optimization, that is, obtaining of the *most immediate superclasses (MIS)* for the classes and restrictions that form the query. These MIS are used to detect inconsistent queries, to transform the query and in some cases to give intensional answers;
3. identification of the cached parts of the query and answering of the query in the cache memory, if it is possible. In other case, obtaining of a set of DL queries to be cached;
4. generation of an optimal plan to answer these non-cached DL queries from the underlying databases.

The Local Query Processors have to receive the plans sent by the Global Query Processor in its last task. In those plans it is explicitly said what to execute and where to send the answer.

4.1 Semantic optimization

In the database area, semantic query optimization methods exploit domain knowledge such as that expressed by integrity constraints, hierarchies, etc. to detect inconsistent queries or to transform a user formulated query into another one with the same answer, that is semantically equivalent, but that can be processed more efficiently. These semantic optimization methods are external to the database systems and are defined as a special purpose mechanism. Using a DL system, it is possible to do semantic query optimization using the reasoning capabilities of these systems. The classification mechanism allows for obtaining the *most immediate superclasses (MIS)* of the class description of the query. The *most immediate superclasses (MIS)* of a class description C are classes that subsume C and that are not subsumed among them, they are the most specific subsumers of C. A detailed definition of MIS is given in the appendix.

4.1.1 Transformation or Detection

With these MIS, it is possible to detect if the query is *inconsistent* (when the special class *nothing* is a MIS for the query) and it is also possible to reformulate the query by adding some MIS to the query or by deleting some class from the class description of the query.

For example: for the query *getall student and youth and all(age,35)* the set of MIS is {*nothing*} meaning that the query is inconsistent because any instance of *youth* has age less than 30 and cannot be 35. This detection of inconsistent queries avoids searching the answer in the underlying databases or in the cache memory because it is known that the answer is empty.

For example: for the query *getall student and all(age,lt(30))* then the set of MIS is {*student, youth*} (see appendix). Therefore *youth* is a class that can be used to answer the query instead of the attribute restriction *all(age,lt(30))*.

Unfortunately, it is not always better to use the MIS to answer the queries. It depends whether the MIS has a good mapping information associated to it or if it is cached.

In the last case, if *youth* were in the cache memory then it would be worth to use it instead of *all(age,lt(30))* but not in other case because the mapping information associated to *youth* ($<idp, \sigma_{age<30}$ *(db1.student* $\cup_{(id)}$ *db2.teacher)*$>$) tells that it is necessary to scan the relations *student* and *teacher* in two different databases in order to retrieve all the youths, but only students have been asked for.

4.1.2 Intensional answers

User queries are usually answered by giving the set of instances that satisfy the conditions in the query. They are considered as extensional answers. However, when working with DL systems it is also possible to give answers in terms of descriptions that satisfy the instances. In this case they are considered as intensional answers. This can be done in two ways:

1. By giving the Most Specific Formulation (MSF) of the query, that is, the *most immediate superclasses* of the query. For example:

   Query: *getall teacher and student and atleast(1,children)*

   Intensional answer (MSF): *getall teaching_assistant and parent*

2. By giving the Extended Query Formulation (EQF). This is possible by using the class definitions instead of class names.

   For example:

   Query: *getall youth and super_parent*

   Intensional answer (EQF): *getall person and all(age,lt(30)) and atleast(5,children)*

   This extended query formulation could be interesting for example for inconsistent queries in order to know the reason for this.

   Query: *getall parent and atmost(0,children)*

   Intensional answer (MSF): *getall nothing*

   Intensional answer (EQF): *getall person and* **atleast(1,children)** *and* **atmost(0,children)**

4.2 Cache optimization

When using a Client/Server architecture to implement a FDBS, it is worth having some data cached in the extension of the knowledge base in order to avoid accessing the underlying databases each time a user formulated a query. Communication cost involved in transferring intermediate results among the nodes and the final reconstruction of the answer can be avoided. Using this type of architecture to implement distributed database systems is relatively less costly than using database machines or very expensive processors. This is particularly true now that workstations are getting faster and cheaper. In [?] three different Client/Server architectures are compared where there is a Server node with a shared database and several Client nodes that want to access that shared database.

Unfortunately, it is not possible to have all the data cached for several reasons:

1. Due to the autonomy of the underlying relational databases, their extensions can be updated so the cached data would become inconsistent very often. When this happens there are two possibilities a) to reestablish the cache memory after every update, but in this case dynamic relations would have to be continuously sent from the Server to the Client or b) not to reestablish the cache memory after every update occurs but any time a query that affected inconsistent cached data were made then it should be answered by accessing the underlying databases, the cache memory would have useless data.

2. The size of the cache memory would be obviously huge because it would be the sum of the size of several databases. This would produce space problems, most of the times it is not possible to have such a huge cache memory and time response problems because frequently asked queries could be answered slower if some not so frequently asked queries were cached.

When working with a DL system there are two different types of values associated with every instance: the *object identifier (OID)*, that is the unique value that identifies it, and the particular values taken by its attributes. DL queries ask for OIDs or for attribute values. For example:

- *getall person*

  It asks for the OIDs of the instances of the class *person*;

- *getall person and atleast(2,children)*

  It asks for the OIDs of the instances of the class *person* that have at least two values for the attribute *children*;

- *[self,rf(age)] for getall person*

  It asks for the value of the attribute *age* for each instance of the class *person*.

In the first two previous queries the answer is a set of OIDs that correspond to the class description. These identifiers do not usually give much information because, they are given automatically by the system. In the third one the result of the query is the set of pairs <OID,value of the attribute *age*>.

Therefore it is possible to cache the OIDs of the instances of a class description (hereafter to *cache a class description* to which a class name is given) and to cache the attribute values for each instance of a class description (hereafter to *cache an attribute* for a class).

For example, it is possible to cache the class description *getall person and atleast(2,children)* instead of caching the entire class *getall person*. A new class name can be given to the cached

class description. And also it is possible to cache the attribute values for each instance of a class description instead of for the entire class. For example, to cache *[rf(name)] for getall person and atleast(2,children)* instead of caching *[rf(name)] for getall person* This is different from object-oriented systems where entire classes have to be cached.

During the query processing task it is necessary to detect if the query can be answered with the data stored in the cache memory, that is, if the query *is contained* in the cache memory. As database queries are descriptions of data, it has to be proved that any data that verifies the description is in the cache.

For example, suppose that in the cache memory there is information about all the persons older than 18 and the next query is made: *obtain all the persons older than 30.* This query may be answered from the cache because all the persons older than 30 are older than 18 and therefore, they are cached.

In general, to verify if a query is in the cache it is not easy and it depends on the query language and on the representation of the cached data. And what is more, that verification should be as fast as possible because much time cannot be spent to verify that finally the query is not cached. When the query language is the relational algebra it is possible to create a graph structure with all the cached views. To verify if a query is cached a matching process between the graph query and the graph cache can be made to know if the query is contained in the cache. In [**?**] it is proposed a structure call *logical access path schema* and an integration algorithm that could be used to detect if a view is cached. Luckily, when working with a DL system, the classification mechanism of classes can be used to verify if queries are cached. If a query class is subsumed by the cached classes then it is true that the instances of the query class are in the cache. However, that does not mean that they can be identified and that the query can be answered directly from the cache.

For example, suppose that all the instances of the class *person* are cached and that the next query is made: obtain all the persons with at least five children (*getall person and atleast(5,children)*). In fact, it is true that all the instances of *person and atleast(5,children)* are in the cache because all the instances of *person* are cached. However it is not possible to answer the query unless the attribute *children* is also cached because it is not possible to distinguish which persons have at least 5 children if the children are not known (another possibility would be that the class *person and atleast(5,children)* were cached).

We can say that the query *[rf(r_1),...,rf(r_N)] for getall C_1 and ... and C_M* is cached if all the class names that appear in C_i and all the attributes that appear in C_i and r_1,...,r_N are cached. But this is a too strong restriction because although two classes were not cached, the intersection of both could be cached and the same query would be also cached. In fact, a query is cached if the MIS of the class description of the query C_1 *and ... and* C_M are cached and also the projected attributes r_1,...,r_N.

For example, suppose that in the previous integrated schema, only the classes *teaching_assistant* and *parent* are cached and that the attribute *name* is cached for both classes.

If the query *[rf(name)] for getall student and teacher and atleast(1,children)* is formulated, it must be verified that it is cached because although neither *student* nor *teacher* nor *children* are cached, the *MIS* of the class description, *teaching_assistant* and *parent*, are cached and the projected attribute *name* is also cached for them.

It is obvious that if the query is cached then it is answered from the cache memory, but if the query is not completely cached then it has to be answered by accessing the underlying databases. However not all the class names and attributes have to be retrieved from the databases because part of the query may be already cached.

For example, suppose that now only the class *parent* is cached and that the attribute *name* is

also cached for *parent*. When the next query is formulated

[rf(name)] for getall student and teacher and atleast(3,children)

then it is verified that the query is not cached because not all the MIS are cached.
MIS = {*teaching_assistant, parent, atleast(3,children)*}
neither *teaching_assistant* nor *children* are cached.

At this point there are several possibilities of queries to cache. Some of them are the following ones:

- to cache the whole query;

  [rf(name)] for getall student and teacher and atleast(3,children)

  It will be more complex to execute this query in the underlying databases but the answer will be the smallest possible one. After that, a future query like *getall student* will not be able to be answered from the cache.

- to cache the classes *student* and *teacher* and their attribute *children*;

  [rf(children)] for getall student

  and *[rf(children)] for getall teacher*

  These two queries will occupy more in the cache so more communication cost will be involved. However, these queries can be independently executed in both database nodes. As an advantage a future query like *getall student* will be able to be answered from the cache.

- to cache the class *teaching_assistant* and its attribute *children*;

  [rf(children)] for getall teaching_assistant

  This is an intermediate solution where more space in the cache is needed than in the first case but less than the second one. Of course, there are some other intermediate possibilities.

After the cache optimization, a set of DL queries to be cached is obtained in order to answer the user query. That set of queries is obtained depending on the space left in the cache and other criteria like probabilities of asking the queries, probabilities of updates in cached queries and response times in the underlying databases.

4.3 Generation of a plan to answer from the underlying databases.

Each one of the DL queries to be cached obtained in the cache optimization step has to be retrieved by accessing the underlying databases. For each DL query two are the steps to be executed:

- translation of the DL query into a multidatabase ERA expression. This is possible because there exists a mapping information associated to any class and attribute of the knowledge base schema. Furthermore, the mapping information for any class description can be expressed in terms of the mapping information of their class names and attributes ([?]).

  For example:

  Query: *getall teaching_assistant and atleast(3,children)*

  Mapping for *teaching_assistant*: $<idp, db1.student \cap_{(id)} db2.teacher>$

  Mapping for *children*: $<idp, idc, db1.has\_child \cup_{(idp)} db2.has\_child>$

  The mapping for the query would be:

  $<idp, \sigma_{new\_attr \geq 3} ((_{idp}\mathcal{F}_{count(idc)} (db1.student \cap_{(id)} db2.teacher \cap_{(idp=idc)} (db1.has\_child \cup_{(idp,idc)} db2.has\_child))))>$

- translation of a multidatabase ERA expression into a set of SQL sentences. For each SQL sentence it has to be said: a) in which node to execute the SQL sentence, b) where to send the answer and c) the prerequisites needed to execute it (if it has to wait for other intermediate results to be sent). Many of the techniques studied about query processing in distributed database systems can be applied in this last point such as optimal execution of joins, use of semi-joins, selection of the nodes where intermediate results must be sent, etc. ([?, ?]).

  For example, a set of SQL sentences to execute the previous query would be:

  Execute in the node of *db1* the next SQL sentence:

```
select idp, idc
from student, has_child
where id=idp
```

  and send the result (called X) to the node of *db2*.

  Execute in the node of *db2* the next SQL sentences:

```
create view Y as
        select idp, idc
        from teacher, has_child, X
        where teacher.idp=X.idp
        union
        select idp,idc
        from X, teacher
        where X.idp=teacher.idp

select idp
from Y
group by idp
having count(idc) >= 3
```

and send the result to the Client node to load the answer in the cache. Notice that the SQL sentences to be executed in the node of *db2* have to wait for the intermediate result X to arrive (X is a prerequisite for the SQL sentence to execute in *db2* node). And also notice that this is one of the strategies because it could also be possible to execute the join first in the *db2* node, send the result to *db1* node and then to execute the final join. In this case, it has been supposed that the join between *student* and *has_child* has a lower selectivity and therefore the result is smaller than the other one.

5 Conclusions

The integration of heterogeneous and autonomous information sources is a requirement for the new type of cooperative information systems. Multidatabase systems have been proposed as a solution to work with different pre-existing autonomous databases because they allow users to query different autonomous databases with a single request.

Although there has been a lot of research about the problems of translation and integration of schemata to obtain integrated ones, the problem of query processing has not been treated so

much. We have built a FDBS that integrates several heterogeneous relational databases by using a DL system. DL systems provide interesting features for developing *semantic and caching query optimization* techniques and also for providing *intensional answers*.

Appendix: the Most Immediate Superclasses

Let $\mathcal{T}$ be the set of classes that form the schema $\mathcal{T} = \{T_1,\ldots,T_P\}$, $\mathcal{Q}$ the set of classes that form the query class $\mathcal{Q} = \{D_1,\ldots,D_M\}$ then the set of *inmediate superclasses* corresponding to the query $\mathcal{MIS}$ is:

$\mathcal{MIS} = \{C \mid (C \in \mathcal{T} \vee C \in \mathcal{Q}) \wedge (C \textit{ subsumes } (D_1 \textit{ and } \ldots \textit{ and } D_M)$
$\wedge \forall E(E \in \mathcal{MIS} \wedge E \neq C \rightarrow E \textit{ does not subsume } C))$

This set of *inmediate superclasses* $\mathcal{MIS} = \{C_1,\ldots,C_N\}$ verifies:

1. $\forall i((C_i \in \mathcal{T}) \vee (C_i \in \mathcal{Q})$

   Every inmediate superclass of the knowledge base is in the query.

2. $\forall i \forall j\ (i \neq j \rightarrow C_i$ does not subsume $C_j)$

   There is no an inmediate superclass that subsumes another one.

3. $\forall C\ [(C \in \mathcal{T} \vee C \in \mathcal{Q}) \rightarrow$

   $(C$ subsumes $(D_1$ and $\ldots$ and $D_M) \rightarrow ((C \in \mathcal{MIS}) \vee (\exists E(E \in \mathcal{MIS} \wedge (E$ subsumes $C)))]$

   Every class of the knowledge base or included in the query that subsumes the query class is an inmediate superclass except if there is already another inmediate superclass that subsumes it.

4. The query class is semantically equivalent to the intersection of all the inmediate superclasses.

   $D_1 \textit{ and } \ldots \textit{ and } D_M$ is semantically equivalent to $C_1 \textit{ and } \ldots \textit{ and } C_N$

   Demonstration:

   - $C_1 \textit{ and } \ldots \textit{ and } C_N$ subsumes $D_1 \textit{ and } \ldots \textit{ and } D_M$
     $\forall i(C_i \in \mathcal{MIS}) \Rightarrow$
     $\forall i(C_i$ subsumes $D_1 \textit{ and } \ldots \textit{ and } D_M) \Rightarrow$
     $\forall i\ (D_1 \textit{ and } \ldots \textit{ and } D_M \subseteq C_i) \Rightarrow$
     $D_1 \textit{ and } \ldots \textit{ and } D_M \subseteq C_1 \textit{ and } \ldots \textit{ and } C_N \Rightarrow$
     $C_1 \textit{ and } \ldots \textit{ and } C_N$ subsumes $D_1 \textit{ and } \ldots \textit{ and } D_M$
   - $D_1 \textit{ and } \ldots \textit{ and } D_M$ subsumes $C_1 \textit{ and } \ldots \textit{ and } C_N$
     By reductio ad absurdum:
     Suppose that $D_1 \textit{ and } \ldots \textit{ and } D_M$ does not subsume $C_1 \textit{ and } \ldots \textit{ and } C_N \Rightarrow$
     $C_1 \textit{ and } \ldots \textit{ and } C_N \not\subseteq D_1 \textit{ and } \ldots \textit{ and } D_M \Rightarrow$
     $\exists i(C_i \not\subseteq D_1 \textit{ and } \ldots \textit{ and } D_M) \Rightarrow$
     $\exists i(C_i$ does not subsume $D_1 \textit{ and } \ldots \textit{ and } D_M)$ # because $C_i \in \mathcal{MIS}$

For example, in the integrated schema of section 3.3, the set of MIS for the query *getall student and all(age,lt(30))* is {*student, youth*} because 1) *student* and *youth* belong to the schema; 2) *student* does not subsume *young* and vice verse; and 3) the set of subsumers of *student and youth* is {*person, student, youth, all(age,lt(30))*}. The only two subsumers that do not belong to MIS, namely *person* and *all(age,lt(30))*, subsume *youth*.

Figure 4: Global Query Processor.