Semantic Query Optimization and Data Caching for a Multidatabase System

Goñi A. , Illarramendi A. , Mena E.

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

Abstract

Cooperative Information Systems (CISs) have to manage with data located in different large information repositories distributed over communication networks. Multidatabase systems have been proposed as a solution to work with different pre-existing autonomous databases. Query processing in multidatabase systems requires a particular study because of the autonomy and heterogeneity of the component databases. In this paper we present the query processing of a multidatabase system with a global view expressed in Description Logics (DL). Query optimization techniques like semantic and caching optimization in this system are also explained.

1 Introduction

Cooperative Information Systems (CISs) have to manage with data located in different large information repositories distributed over communication networks. The integration of heterogeneous and autonomous information sources is a requirement for these CISs. 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, ([BNPS89, SK92, SPD92, LNE89, CHS91, NGG89, QFG92]) for the problem of query processing against these integrated schemata only some particular solutions have been proposed. Moreover, the solutions studied for query processing in distributed database systems ([CP84, OV91]) are not the same as the solutions needed for multidatabase systems as several authors claim ([OV91, Day85, LOG92]) because of the autonomy and heterogeneity of the component databases.

Nowadays, there are some commercial multidatabase languages that can access data from different databases with some restrictions such as the types of DBMS they work with (i.e. ORACLE7^{TM}). By using these languages directly, users need to express from where the data must be obtained and therefore few techniques of query optimization are applied during the query processing. However, when working with an integrated schema, the queries over this integrated schema may be translated in some multidatabase queries or others taking into account information about *replication of data* and information about *cached data* in the integrated schema.

In general, the integration of different information sources is made in terms of the Real World States (RWS) of objects belonging to these information sources. Four different types of correspondences can be established between the RWS of the objects, namely *equivalent*, one *included* in another one, *overlapping* or *disjoint*. Once an integrated schema is generated using these types of correspondences it can be observed that objects that belong to this schema can have *alternative supports or mapping informations*, that is, their extensions can be found indistinctly in different sources. In fact there exist replicated data.

On the other hand, Client/Server architectures have been shown as appropriated for building and supporting multidatabase systems. Using this type of architecture a Client application can be defined that deals with the integrated schema 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 this context, 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.

Therefore, dealing with a multidatabase system implemented using a Client/Server architecture and that has some data cached for the integrated schema, the query processor has to take into account the existence of *replicated data* in the underlying databases and also the existence of *cached data* in the integrated schema node in order to perform some query optimization such as *semantic query optimization*. It is important to notice that in this context where some data are cached the user queries can be answered from the cached memory but if that is not possible, they have to be answered from the underlying databases. But once the underlying databases have to be accessed, more data can be brought than just the data needed to answer the query. But on the other hand, the more data is brought and cached, the more possibilities of answering queries from the cache but also the bigger is the cache memory. It may be possible to ask the user to *cooperate* with the system and tell the system if he is interested in bringing more information or not, depending on if he is going to ask for that data later.

Finally, although different models have been used for building FDBS (Entity-Relationship, Object-Oriented and different Knowledge Representation Systems (KRS)), we use a KRS based on Description Logics (DL) that has some advantages to do *caching* and *semantic optimization* and to offer *intensional answers*. This last feature is very useful in modern cooperative systems where an interaction with the user is preferred when for example inconsistent queries are asked.

There are several related works where a DL system is used in connection to information systems but only a few of them talk about query processing aspects. In particular, in [Dev93] Devanbu explains how translators from DL queries to database queries can be built. Borgida and Brachman [BB93] present an efficient translator from DL queries to SQL queries and also presents some problems when loading data into the DL knowledge base. In the two previous cases only one database is connected to the DL knowledge base, and therefore there is no replication of data in more than one database and the whole DL knowledge base can be loaded at the beginning of the session. There is no a caching problem because everything is cached since the beginning. In [ACHK93] Arens et al. show the SIMS system that integrates data from several information sources (databases and Loom knowledge bases). From the query processing point of view they select the appropriate information sources, create plans to answer the query and reformulate these plans by exploiting knowledge about the domain and the information sources. They also point that the retrieved data can be cached but they do not say which data is interesting to cache and how the plans vary depending on if the data are cached or not.

In this paper, we present first the work's framework with a brief introduction to DL systems, the system architecture of the FDBS, the mapping information concept, the cache memory and an example of the integrated schema. Then we focus on the query processing in the FDBS and its main stages: semantic optimization and caching optimization. Finally, some graphics that show the behavior of the cache are given.

2 Work's framework

Before focusing on the query processing we give a brief introduction to systems based on Description Logics (DL) because the integrated schema has been built by using a DL system. Moreover, 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. Later we explain the general features of the cache memory and the method to define an optimal cache. Furthermore, we also present an example of an integrated schema that will be used throughout section 3 to illustrate query processing aspects.

2.1 Systems based on Description Logics (DL)

The integrated schema will be represented as a knowledge base that contains 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.

Moreover, two important features in systems based on DL 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 system based on DL, 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.

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 retrieval as queries [Bor92].

Therefore the notion of subsumption between *classes* can be used as well with queries: *some queries or classes subsume other queries or classes.* This feature is used to verify whether data are cached or not when a query is formulated.

These DL systems provide some interesting features for developing *semantic and caching query optimization* techniques and also for providing *intensional answers* to the user.

2.2 System Architecture

As said in the introduction the Client/Server architecture is used for the implementation of the FDBS (see figure 1) where four main components can be distinguished: Translator, Integrator, Monitor and Query Processor.

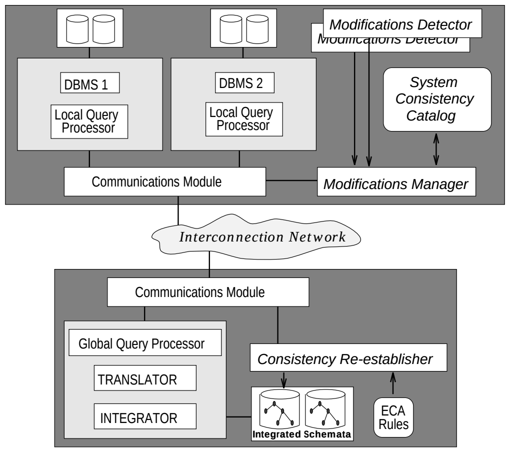

Figure 1: Architecture of the FDBS

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. 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.

The *Translator* and the *Integrator* are explained in detail in [?], the *Monitor* in [BIPG92, IBM+95] and the *Query Processor* is presented in this paper and in [?].

2.3 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 [Bla94] 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/s 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*.

  Example. Let us suppose that there exists the table `student(id,name,address)`. 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*.

  Example. Supposing now that there exist the tables `student(id,name,address)` and `studies(s_id,course#)`, an attribute *studies* for the class *student* can be defined. The mapping information for the attribute *studies* could be <*s_id,course#,studies*> meaning that the different values taken by `course#` for the same value of `s_id` are the *courses* studied by the *student* represented by that `s_id` value.

2.4 Cache memory

As was mentioned before, on one hand 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. But on the other hand it is not possible to cache all the data stored on the underlying database systems mainly for these reasons: a) the cached data could become inconsistent very often due to the autonomy of the underlying databases and b) the size of the cache memory would be obviously huge because it would be the sum of the size of several large databases.

Therefore, it is necessary to decide which data is interesting to cache. This means first, to decide the type of objects to be cached and, then to define a cost model that allows for the evaluation of the cost and benefit provided by the cached data, and last to design an algorithm that uses the previous cost model to find the optimal set of queries to be cached.

1. *Type of objects to be 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).

2. *Cost model.*

   The idea is to make a study of those DL queries made previously to the system that are worth caching, that is, the set of optimal queries that offer more benefit when they are cached for a given maximum size of the cache memory.

   Let $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_N\}$ be the set of all the DL queries made to the system and M the maximum size for the cache memory, then what we need to find is the subset $\mathcal{Q}^* \subseteq \mathcal{Q}$ such that maximizes the following formula:

$$\sum_{Q_j \in \mathcal{Q}^*} G(Q_j) \tag{1}$$

   ensuring that the cost of the cache remains smaller than a threshold

$$\sum_{Q_j \in \mathcal{Q}^*} C(Q_j) \leq M \tag{2}$$

   where G(Q$_j$) is the *benefit* of having Q$_j$ in the cache and C(Q$_j$) its respective *cost*,

$$G(Q_j) = \alpha \times P_Q(Q_j) \times (T_{DBS}(Q_j) - T_{CACHE}(Q_j)) - \beta \times P_U(Q_j) \times T_{CACHING}(Q_j) \tag{3}$$

$$C(Q_j) = \begin{cases} 0 & \textit{if } Q_j \textit{ is implicitly cached} \\ |Q_j| & \textit{in other case} \end{cases} \tag{4}$$

   $P_Q(Q_i)$ is the probability of asking for Q_i.
   $P_U(Q_i)$ is the probability of updating the extension of Q_i.
   $T_{DBS}(Q_i)$ is the response time for Q_i in the underlying databases.
   $T_{CACHE}(Q_i)$ is the response time for Q_i in the cache.
   $T_{CACHING}(Q_i)$ is the time for storing Q_i in the cache.
   $|Q_j|$ is the size of the query Q_j.

   In the equation 3 ($T_{DBS}(Q_j) - T_{CACHE}(Q_j)$) is a factor that indicates the benefit of answering Q_j from the cache instead of from the underlying databases. The more Q_j is asked the more benefit is gained because that factor is multiplied by $P_Q(Q_j)$. However, it is possible that the query gets updated often so queries with a great probability of updating and with a great cost of caching have to decrease the previous benefit ($P_U(Q_j) \times T_{CACHING}(Q_j)$). The constants α and β adjust the two previous factors.

In [GIMB94] we explain with more detail the cost model, the parameters, the notion of *implicitly cached* queries[1] and give an algorithm to calculate the set of optimal queries to be cached.

The process that calculates the queries worth to be cached has to take into account many parameters and it cannot be executed each time a query is made. Furthermore, it is possible that this process decides not to cache the last made queries because their probabilities of asking are not great or because they are very volatile. However, the work with the knowledge base is made in sessions where one loads it, queries something and ends the session. It is quite possible that the most recently query made is done again in the same session (although in all sessions is not usually asked). To solve this problem, we use two different replacement policies for the cache: the *static* and the *dynamic*. The *static* strategy is to cache the set of optimal queries to be cached and the *dynamic* strategy is to cache the last queries and deallocate, when space is needed, the Least Recently Used queries. The *static* strategy is used between sessions (at the end of the session or in off-peak hours) and the *dynamic* strategy is used during a session.

2.5 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, courses and which teachers teach what courses and which students attend what courses.

db1	db2
student(id,name,address)	teacher(id,name,address,title,degree)
studies(s_id,course#)	course(c#,name,depart,creds)
	teaches(t_id,c#)

Figure 2: Simplified schemata

The Translator and the Integrator have obtained the primitive classes *person* with attributes *name* and *address*, the class *teacher* with attributes *title, degree, teaches* and *teaches_to* and the class *student* with attributes *studies*. And also the defined classes *teaching_assistant* (teachers and students at the same time), *super_student* (students enrolled in atleast ten courses) and *lucky_teacher* (teachers with no students). These classes and attributes can be seen in figure 3.

The mapping information associated to some of these classes and attributes appears in figure 4.

Notice that there is *replication of data* because the instances of the class *teaching_assistant* can be obtained from three different ways: a) accessing only to *db1*, b) accessing only to *db2* or c) accessing both *db1* and *db2* and doing the intersection.

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

[1] Implicitly cached queries are queries that can be answered using other explicitly cached queries. They do not occupy space in the cache memory.

CLASSES
person :< *anything*
student :< *person*
teacher :< *person*
course :< *anything*
teaching_assistant := *teacher and student*
super_student := *student and atleast(10,studies)*
lucky_teacher := *teacher and atmost(0,teaches_to)*

ATTRIBUTES (only some of them appear here)
name :< *domain(person) and range(string)*
title :< *domain(teacher) and range(string)*
teaches :< *domain(teacher) and range(course)*
teaches_to :< *domain(teacher) and range(student)*
studies :< *domain(student) and range(course)*

Figure 3: Integrated schema

CLASS	$<attr, rel>$
person	$<id, db1.student \cup_{(\mathrm{id})} db2.teacher>$
student	$<id, db1.student>$
teacher	$<id, db2.teacher>$
teaching_assistant	$<id, db1.student \cap_{(\mathrm{id})} db2.teacher>$
teaching_assistant	$<id, \sigma_{\mathrm{cat="grad\_stud"}} (db2.teacher)>$
teaching_assistant	$<id, \sigma_{\mathrm{course\# \geq 600}} (db1.student)>$
super_student	$<s\_id, \sigma_{\mathrm{new\_attr \geq 10}} ((\mathrm{idp}\ \mathcal{F}_{\mathrm{count(course\#)}} (db1.studies)))>$
lucky_teacher	$<s\_id, db2.teacher -_{(\mathrm{id=t\_id})} db2.teaches>$

ATTRIBUTE	$<attr\_inst, attr\_attr, rel>$
studies	$<s\_id, course\#, db1.studies>$

Figure 4: Mapping Information

3 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 Processor when it is needed.

The different tasks made by the Global Query Processor are (see figure 5):

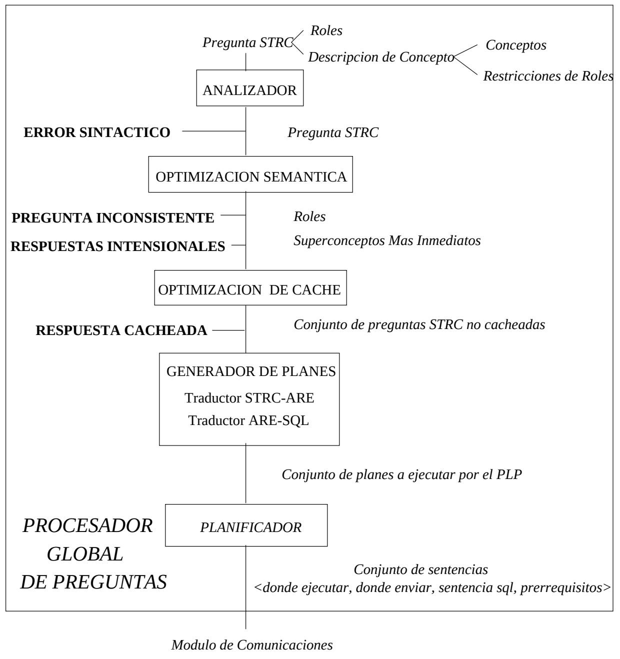

Figure 5: Global Query Processor.

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.

3.1 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. These MIS can be used to transform the user formulated query into a semantically equivalent one, to detect inconsistent queries or to give intensional answers.

3.1.1 Transformation or Detection

Dealing with MIS, it is possible to detect if the query is *inconsistent* (when the special class *nothing*[2] 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 the query *getall lucky_teacher and atleast(5,teaches)* the set of MIS is {*nothing*} meaning that the query is inconsistent because any instance of *lucky_teacher* does not have a value for the attribute *teaches*, so there cannot be one with atleast five values for that attribute. 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 [IBG94].

For the query *getall teacher and student and atleast(15,studies) and atmost(0,teaches)* then the set of MIS is {*teaching_assistant, lucky_teacher, super_student, atleast(15,studies)*}. Therefore *teaching_assistant, lucky_teacher* or *super_student* are classes that can be used to answer the query instead of the original classes *teacher, student* and the restriction *atmost(0,teaches)*

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.

For example, if *super_student* is not cached it is not worth to use it in the query plan because the mapping information for *super_student and atleast(15,studies)* is more complex than the mapping information for *student and atleast(15,studies)*. But although *teaching_assistant* is not cached it is still worth to use it because it has a mapping information better than the mapping information corresponding to *teacher and student* (*teaching_assistant* can be found in either *db1* or in *db2* but *teacher and student* have to access both databases).

3.1.2 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(15,studies) and atmost(0,teaches)*

[2]Nothing is a special class such that no instance can belong to it

Intensional answer (MSF): *getall teaching_assistant and lucky_teacher and super_student and atleast(15,studies)*

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

   For example:

   Query: *getall teaching_assistant and super_student*

   Intensional answer (EQF): *getall teacher and student and atleast(10,estudies)*

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

   Query: *getall lucky_teacher and atleast(5,teaches_to)*

   Intensional answer (MSF): *getall nothing*

   Intensional answer (EQF): *getall teacher and* **atmost(0,teaches_to)** *and* **atleast(5,teaches_to)**

3.2 Cache optimization

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.

3.2.1 Identification of the cached parts of the query

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 *super_student* are cached and that the attribute *name* is cached for both classes.

If the query *[rf(name)] for getall student and teacher and atleast(10,studies)* is formulated, it must be verified that it is cached because although neither *student* nor *teacher* nor *studies* are cached, the *MIS* of the class description, *teaching_assistant* and *super_student*, are cached and the projected attribute *name* is also cached for them.

3.2.2 Obtaining of a set of DL queries to be cached

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 parts of the query may already be cached.

For example, suppose that only the class *teacher* is cached and that the attribute *name* is also cached for *teacher*. When the next query is formulated

[rf(name)] for getall student and teacher and atleast(10,studies) and atmost(1,teaches)

then it is verified that the query is not cached because not all the MIS are cached (in fact none of them).

MIS = {*teaching_assistant, super_student, atmost(1,teaches)*}

In the figure 6 the classes, attributes and attribute restrictions needed to answer the query can be seen as a tree. In the first level of the tree, the nodes are the MIS for the query and the projected attributes. Every node that corresponds to a defined class is expanded with its MISs and so on. The underlying classes and attributes are cached.

```
Query
          class: teaching_assistant
                              class: student
                              class: teacher
          class: super_student
                              class: student
                              cardinality restriction: atleast(10,studies)
          cardinality restriction: atmost(1,teaches)
          attribute to project: name
```

Figure 6: Tree corresponding to the example query

Looking up the non-cached nodes, it can be decided which are the DL queries needed to retrieve from the underlying databases. The answers for these DL queries are then cached so that they can be answered from the cache memory after that. But there are several possibilities of DL queries to cache in order to answer the original query:

1. to cache the query, equivalent to the initial one, formed by the conjunction of the MIS;

   [rf(name)] for getall teaching_assistant and super_student and atmost(1,teaches)

   teaching_assistant is a MIS that has alternative mapping informations; it can be retrieved from different databases: only from *db1*, only from *db2* or from both;

super_student is a MIS that does not have an alternative mapping information different to the mapping information corresponding to the expression given by its MIS *student and atleast(10,studies)*, so the previous query is the same as

[rf(name)] for getall teaching_assistant and student and atleast(10,studies) and atmost(1,teaches)

but *student* subsumes *teaching_assistant*, so it can be ignored

[rf(name)] for getall teaching_assistant and atleast(10,studies) and atmost(1,teaches)

2. to cache the conjunction of only the non-cached parts of the query;

   getall student and atleast(10,studies) and atmost(1,teaches)

   The MIS of *teaching_assistant* are *teacher* (cached) and *student*, so *teaching_assistant* is substituted by *student* in the query.

   The projection of the attribute *name* has been deleted because *name* is cached.

   Notice that the previous query is more general than the one needed to answer the user query because it is retrieving more information than the strictly necessary: *student* subsumes *student and teacher*.

3. to cache the values for the attributes of the restrictions;

   [rf(studies),rf(teaches)] for getall student

   To look for the instances that verify *atleast(10,studies)* or *atmost(1,teaches)* it is necessary to access the databases. But, if after that, the user wants to know the particular values for the attributes *studies* or *teaches* another searching in the databases has to be made.

   The next two queries can also be retrieved because they can be executed in parallel in both databases:

   [rf(studies)] for getall student

   [rf(teaches)] for getall teacher

   But as *teaching_assistant* has a support in both databases, the queries to be retrieve can be:

   [rf(studies)] for getall teaching_assistant

   [rf(teaches)] for getall teaching_assistant

In general it can be stated that there are two phases to get the set of DL queries to cache: a) to transform the query formed by the MIS and the projections of attributes into another one and b) to split it into queries to be executed in parallel if it is possible. In the phase a) the following steps are made:

a1) MIS and projected attributes that are already cached are removed from the query.

a2) *Defined* MIS with alternative mapping informations are kept in the query.

a3) *Defined* MIS with no alternative mapping informations are substituted by their corresponding non-redundant MIS. To these new MISs the steps a1), a2) and a3) are applied.

a4) *Defined* MIS with alternative mapping informations may be substituted by their corresponding non-redundant MIS (and applying a1), a2) and a3)), above all if some of them are already cached, and there are possibilities of asking for some of them and space in the cache memory.

a4) Attribute restrictions may be substituted by the projection of the corresponding attributes if there are possibilities of asking for these attribute values and space in the cache memory.

In the phase b) the query previously obtained can be splitted in several queries:

b1) The query can be splitted in queries such that all the mapping informations of the classes and attributes of each one of the queries are in the same database. In this case, the splitted queries can be executed in parallel.

b2) If there are possibilities of asking for part of the query, this part can be separated from the query.

Some of the previous transformations, in particular a4), b1) and b2), imply to bring more data than the requested in the query. It may be thought that if the data was not cached it is because it was not worth (as found by the algorithm to find the optimal set of objects to be cached), but at this moment the user has asked for another information and it can be supposed that there are possibilities of asking for these data. Anyway, it is possible to *cooperate* with the user and ask him if he is interested in other related data to the query. But *this can be done only if there is enough space in the cache memory.*

For example if the user queries *getall teacher and atleast(3,teaches)* then the system can ask the user:

`Do you want to know the values of attribute` *teaches*?

If the answer is *yes* then the the query *[rf(teaches)] for getall teacher* is splitted.

3.3 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 integrated schema. Furthermore, the mapping information for any class description can be expressed in terms of the mapping information of their class names and attributes ([Bla94]).

  For example:

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

Mappings for *teaching_assistant*	Mapping for *studies*
$<id, db1.student \bigcap_{(\mathrm{id})} db2.teacher>$	$<s\_id, course\#, db1.studies>$
$<id, \sigma_{\mathrm{course\#} \geq 600}\ (db1.student)>$	
$<id, \sigma_{\mathrm{cat="grad\_stud"}}\ (db2.teacher)>$	

  The mapping for the query would be:

  $<id, \sigma_{\mathrm{new\_attr} \geq 3}\ ((_{\mathrm{id}}\mathcal{F}_{\mathrm{count(course\#)}}\ (\sigma_{\mathrm{course\#} \geq 600}\ (db1.student) \bowtie_{(\mathrm{id=s\_id})} db1.studies)))>$

- translation of a multidatabase ERA expression into a set of SQL sentences. This translation is very easy if a multidatabase SQL processor is available. 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). The steps b) and c) would not be necessary because the would do this work. 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. ([CP84, OV91]).

  In the example, the next SQL sentence executed in *db1* node obtains the answer for the query.

```
select id
from db1.studies,db1.student
where course#>=600 and id=s_id
group by id
having count(*)>=3
```

4 Experiments

In this section we present some graphics that show mainly the behavior of the cache memory. We have started by calculating the set of optimal queries (see section 2.4) for the queries that ask for the classes (*getall person, getall teacher, getall student, ...)* and for the queries that ask for the attributes (*rf(name) for getall person, rf(address) for getall person, rf(title) for getall teacher, ...*). The values for the parameters $T_{DBS}(Q_i)$, $T_{CACHE}(Q_i)$, $T_{CACHING}(Q_i)$ and $|Q_i|$ are obtained after translating the DL queries into SQL queries and executed in the underlying databases.

The queries in the sessions change the probabilities $P_Q(Q_i)$ of them to be asked. The new queries are added to the set of queries. After each session, the set of optimal queries is recalculated. This is why the benefit of the cache memory augments with the sessions and the cost remains more or less constant under the cache size (see figure 7: graphics on the right).

When a query is not cached, the strategy used to query in the underlying databases have been always the same: to cache the conjunction of the non-cached parts of the query. *Inconsistent* queries have not been asked because they do not affect the cache contents.

The hit ratio (number of cached queries / number of made queries) is quite good. It takes only 5-6 sections to be more than 90% (see figure 7: graphic on the left).

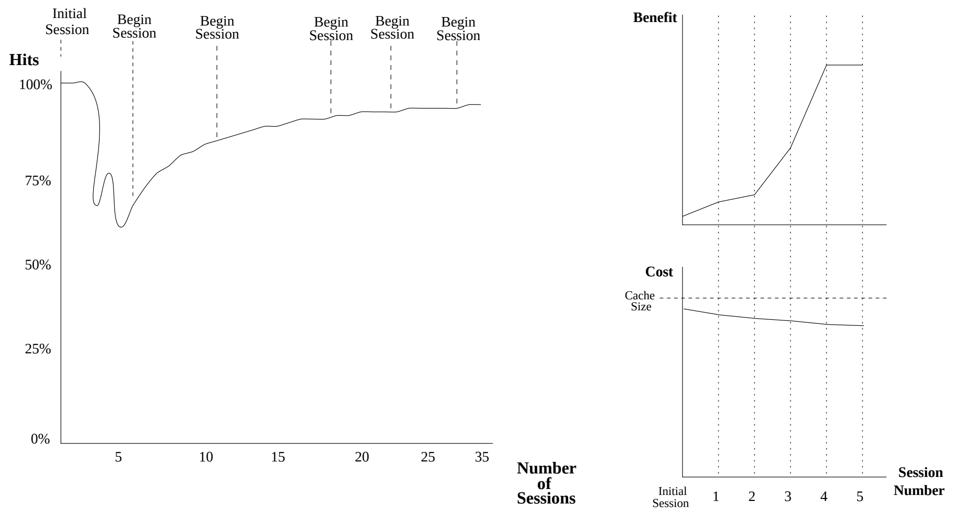

Figure 7: Graphics the show the hit ratio, benefit and cost of the cache

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*. Some graphics have been also shown.

References

[ACHK93] Y. Arens, C.Y. Chee, C. Hsu, and C.A. Knoblock. Retrieving and integrating data from multiple information sources. *International Journal of Intelligent and Cooperative Information Systems*, 2(2):127–158, 1993.

[BB93] A. Borgida and R. J. Brachman. Loading data into description reasoners. In *Proceedings of the ACM SIGMOD Conference*, 1993.

[BIPG92] J. M. Blanco, A. Illarramendi, J. M. Pérez, and A. Goñi. Making a federated database system active. In *Database and Expert Systems Applications*. Springer-Verlag, 1992.

[Bla94] J.M. Blanco. *Integración de bases de datos relacionales por medio de un sistema terminológico: una propuesta utilizando BACK*. PhD thesis, Basque Country University, April 1994.

[BNPS89] E. Bertino, M. Negri, G. Pelagatti, and L. Sbatella. Integration of heterogeneous database applications through an object-oriented interface. *Information Systems*, 14(5), 1989.

[Bor92] A. Borgida. From type systems to knowledge representations: Natural semantics specifications for description logics. *Intelligent and Co-operative Information Systems*, 1(1), 1992.

[CHS91] C. Collet, M. N. Huhns, and W. Shen. Resource integration using a large knowledge base in CARNOT. *IEEE Computer*, December 1991.

[CP84] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems*. Mac Graw Hill, 1984.

[Day85] U. Dayal. *Query Processing in a Multidatabase System*, pages 81–108. Springer-Verlag, 1985.

[Dev93] P.T. Devanbu. Translating description logics to information server queries. In *Proceedings of the ISMM International Conference on Information and Knowledge Management CIKM*, 1993.

[GIMB94] A. Goñi, A. Illarramendi, E. Mena, and J.M. Blanco. A method to define an optimal cache for a multidatabase system. Submitted for publication, 1994.

[IBG94] A. Illarramendi, J.M. Blanco, and A. Goñi. Making the knowledge base systems more efficient: A method to detect inconsistent queries. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):634–639, August 1994.

[IBM$^+$95] A. Illarramendi, J. M. Blanco, E. Mena, A. Goñi, and J. M. Pérez. Maintaining schemata consistency for interoperable database systems. In *Proceedings of the Fourth International Conference on Database Systems for Advanced Applications (DASFAA'95)*. World Scientific Publishing, 1995. To appear.

[LNE89] J. A. Larson, S. B. Navathe, and R. Elmasri. A theory of attribute equivalence in databases with application to schema integration. *IEEE TOSE*, SE-15(4), April 1989.

[LOG92] H. Lu, B. Ooi, and C. Goh. On global multidatabase query optimization. *SIGMOD RECORD*, 21(4), December 1992.

[NGG89] S. Navathe, S. K. Gala, and S. Geum. Federated information bases: A loose-coupled integration of databases systems and application subsystems. In *Proc. of the 4th. Database Symposium*, 1989.

[OV91] M. T. Ozsu and P. Valduriez. *Distributed Databases: Principles and Systems*. Prentice Hall, 1991.

[QFG92] M.A. Qutaishat, N.J. Fiddian, and W.A. Gray. Association merging in a schema meta-integration system for a heterogeneous object-oriented database environment. In *Lecture Notes in Computer Science Proc. 10th British National Conference on Databases*, 1992.

[SK92] W. Sull and R. L. Kashyap. A self-organizing knowledge representation scheme for extensible heterogeneous information environment. *IEEE Transactions on Knowledge and Data Engineering*, 4(2), April 1992.

[SPD92] S. Spaccapietra, C. Parent, and Y. Dupont. Model independent assertions for integration of heterogeneous schemas. *VLDB*, 1:81–126, 1992.

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:

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

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.

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

   Demonstration:

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

For example, in the integrated schema of section 2.5, the set of MIS for the query *[rf(name)] for getall teacher and student and atleast(10,studies) and atmost(1,teaches)* is {*teaching_assistant, super_student, atmost(1,teaches)*} because 1) *teaching_assistant* and *super_student* belong to the schema, and *atmost(1,teaches)* belongs to the query; 2) *teaching_assistant* does not subsume *super_student* nor *atmost(1,teaches)* and so on; and 3) the set of subsumers of *teaching_assistant and super_student and atmost(1,teaches)* is {*person, student, teacher, teaching_assistant, super_student, atmost(1,teaches)*}. The subsumers that do not belong to MIS, namely *person, student* and *teacher*, subsume *teaching_assistant.*