Localization of Distributed Data in a CORBA-based environment

MILKO MARINOV, SVETLANA STEFANOVA
Department of Computer Systems & Technologies
University of Rousse,
8 Studentska St., 7017 Rousse
BULGARIA
MMarinov@ecs.ru.acad.bg, SStefanova@ecs.ru.acad.bg

Abstract: - Query processing over distributed and fragmented databases is more challenging than doing so in a centralized environment. In a distributed environment, the DBMS needs to know where each node is located. The main role of data localization layer is to localize the query's data using data distribution information. We propose an approach to incorporate the artificial intelligence techniques into a distributed database management system (DBMS), namely to extend the core of a distributed CORBA-based environment with deductive functionalities of the query and view services during the process of data localization. The basic principles and the architecture of the software tool are considered. The implementation and class hierarchy of the object-oriented theorem prover which is built in the core of distributed CORBA-based system are also discussed.

Key-Words: - Distributed systems, Data localization, CORBA-based architecture, Theorem prover.

1 Introduction

The increasing volume of stored data poses new challenges to efficient query processing. Queries posed over centralized databases may take very long times to be answered, since large amounts of data need to be accessed. In most of the cases, indexes are not enough to increase query performance. A solution to this problem may be to distribute and fragment data across the network. In fact, lots of systems for processing queries over distributed data have been proposed [1,4,9,12,16,25,27]. Some others focus on adaptive query processing over heterogeneous distributed systems [7,8,10,31]. In relational databases [23] fragmentation techniques have been successfully used to increase query performance in distributed databases. By fragmenting and distributing the data, queries can be sent to specific fragments, avoiding a complete scan over large portions of irrelevant data. Query processing is the process by which a declarative query is translated into low level data manipulation operations. Query processing deals with designing algorithms that analyze queries and convert them into a series of data manipulation operations. Besides the methodological issues, an important aspect of query processing is data localization. The main role of data localization is to localize the query's data using data distribution information. The localization layer translates an algebraic query on global relations into an algebraic query expressed on physical fragments. Localization uses information stored in the fragment schema. The problem is how to decide on a strategy for executing each query over the network.

New technologies were developed to share data scattered on the net. The Object Management Architecture (OMA) is an environment defined by the Object Management Group (OMG). OMA defines a common object model, a common model of interaction by means of object invocations and a set of common object services and facilities. OMA modules consist of the application objects; the Common Object Request Broker (CORBA), which directs requests and responses between objects; a set of common object services, which are the basis functions required for object management; and a set of common facilities, which are generic object-oriented tools for various applications [24,26]. CORBA is the key communication mechanism of OMA, in which objects communicate with each other via an Object Request Broker (ORB) that provides brokering services between clients and servers. CORBA is an open, standard solution for distributed object systems. CORBA objects communicate directly across a network using standard protocols, regardless of the programming languages used to create objects or the operating systems and platforms on which the objects run. An alternative to CORBA infrastructure is the Distributed Component Object Model/Object Linking and Embedding (DCOM/OLE) environment from Microsoft [5,11]. DCOM is similar in functionality to CORBA ORB, while OLE

is the complete environment for componentization. DCOM has the similar goals as CORBA. It can be used to develop very advanced distributed systems. Unlike CORBA that is language-neutral, DCOM is mainly used for WINDOWS applications with Microsoft's ActiveX. DCOM object model is quite different than the model supported by CORBA [14,22]. Borland's Multi-tier Distributed Applications Services (MIDAS) [6] delivers distributed applications. MIDAS thin-client applications are easy to develop, easy to deploy, require zero configuration and automatically update server based business logic without re-deploying thin-clients.

CORBA is a middleware that enables interoperability and supports distributed object computing. As an object-oriented distributed computing platform CORBA can be helpful for database interoperability [11,24,32]. The fundamental contribution is in terms of managing heterogeneity and to a lesser extent managing autonomy. Heterogeneity in a distributed system can occur at the hardware and operating system level, communication level, DBMS level and semantic level. CORBA deals mainly with platform and communication heterogeneities. It is also addresses DBMS heterogeneity by means of IDL interface definition. However, the real problem of managing multiple DBMSs in the sense of a multidatabase system requires the development of a global layer that includes the global-level DBMS functionality.

Our work proposes an approach to incorporate the artificial intelligence techniques into a distributed database management system (DBMS), namely to extend the core of a distributed CORBA-based environment with deductive functionalities of the query and view services during the process of data localization. The present paper considers an architecture and implementation of the distributed tool. The structure of this paper is as follows. After this introduction, in section 2 the generic layering scheme for distributed query processing is discussed, in which the first two layers are responsible for query decomposition and data localization. Section 3 provides some details about data localization layer. The architecture and implementation of the tool are described in the next paragraph. The class hierarchy of the object-oriented theorem prover which is built in the core of distributed CORBA-based system is considered in Section 5. The conclusion summarizes the authors' contributions and their future research intentions.

2 Querying distributed databases

Query processing over distributed and fragmented databases is more challenging than doing so in a centralized environment. In a distributed environment, the DBMS needs to know where each node is located, as well as parameters such as communication costs and current load of each node, to be used by the query optimizer. Fragmentation further adds complexity related to reducing the query so it can be executed only in nodes that have relevant data to that query answer. In [2,19,23] we can find a very good reference about distributed databases, and also a methodology for distributed query processing in relational databases. The general ideas of this methodology can be applied to other data models [12,18,25]. More advanced topics on distributed query processing, such as optimization techniques, execution techniques, dynamic replication in the distributed environment, caching, architectures, etc., can be found in [16,19,28,30].

In general, to process a distributed query we need to transform a high-level query over the global (centralized) view of the distributed data into one or more sub-queries of lower level (to be executed over the nodes in the distributed environment). We now present a summary of fragmentation and query processing techniques in the relational model, object-oriented model and semi-structured model.

Ozsü and Valduriez [4,23] present a methodology to process distributed queries over the relational model. The methodology consists of several layers: query decomposition; data localization; global optimization; and local optimization, as shown in Figure 1.

The first layer decomposes the query in an algebraic query over global relations. In this process, syntactic and semantic analyses are performed over the submitted query. The query is simplified and rewritten in an algebraic form. Information about data distribution is not used in this layer.

The data localization layer has as a goal to locate query data by using information about data distribution (fragmentation and allocation of fragments among the nodes). This layer determines the fragments that are involved in the query, and replaces the references to the global view by references to fragments in the algebraic query. This replacement can be performed automatically when the fragmentation design follows correction rules [23,29] such as the reconstruction rule (the global database can be reconstructed from its fragments).

The global optimization layer tries to find a near to optimal strategy (or plan) to execute the global

query. The optimization is performed, in general, by minimizing a cost function. The cost function is usually a combination of CPU, communication and I/O costs. To databases distributed in slow networks, the communication cost must be the most important factor in the cost function.

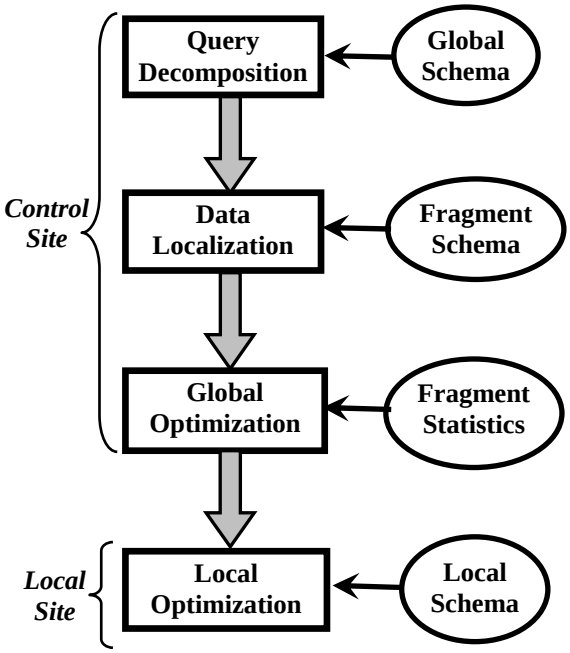

Fig.1 Generic scheme for distributed query processing

After global optimization of the global query, sub-queries are generated and sent to the remote sites. Each sub-query executed in a given site is further optimized using the local site schema in the local optimization layer. The results of the remote sub-queries are processed by the DBMS according to the adopted strategy of final result composition.

3 Localization of distributed data

The initial algebraic query generated by the query decomposition step is input to the second step: data localization. The initial algebraic query is specified on global relations irrespective of their fragmentation or distribution. The main role of data localization is to localize the query's data using data distribution information. In this step, the fragments that are involved in the query are determined and the query is transformed into one that operates on fragments rather than global relations. As indicated earlier, fragmentation is defined through fragmentation rules that can be expressed as relational operations (horizontal fragmentation by selection, vertical fragmentation by projection). A distributed relation can be reconstructed by applying the inverse of the fragmentation rules. This is called a localization program. The localization program for a horizontally (vertically) fragmented query is the union (join) of the fragments. Thus, during the data localization step each global relation is first replaced by its localization program, and then the resulting fragment query is simplified and restructured to produce another "good" query. Simplification and restructuring may be done according to the same rules used in the decomposition step. As in the decomposition step, the final fragment query is generally far from optimal; the process has only eliminated "bad" algebraic queries. We do not consider the fact that data fragments may be replicated, although this can improve the performance. A naïve way to localize a distributed query is to generate a query where each global relation is substituted by its localization program. This can be viewed as replacing the leaves of the operator tree of the distributed query with subtrees corresponding to the localization programs. The query obtained by this way is called localized query. In the reminder of this section for primary horizontal fragmentation we present reduction techniques that generate simpler queries. The horizontal fragmentation function distributes a relation based on selection predicates. The following example is used in subsequent discussions.

Consider the employees relation EMP(ID, MGR_ID, NAME, SALARY). The ID column is a primary key for the EMP table and MGR_ID is the ID of the manager of the corresponding employee. The EMP relation is partitioned into three fragments:

$$EMP_1 = \sigma_{SALARY>100,000}(EMP)$$
$$EMP_2 = \sigma_{50,000<SALARY\leq 100,000}(EMP)$$
$$EMP_3 = \sigma_{SALARY\leq 50,000}(EMP)$$

The localization program for an horizontally fragment relation is the union of the fragments. In our example we have

$$EMP = EMP_1 \cup EMP_2 \cup EMP_3$$

Thus the generic form of any query specified on EMP is obtained by replacing it by

$(EMP_1 \cup EMP_2 \cup EMP_3)$.

The reduction of queries on horizontally fragmented relations consists primarily of determining, after restructuring the subtrees, those that will produce empty relations and removing them. Selections on fragments that have a qualification contradicting the qualification of the

fragmentation rule generate empty relations. Given a relation R that has been horizontally fragmented as $R_1, R_2, \ldots, R_n$, where $R_j = \sigma_{Pj}(R)$, the rule can be state formally as follows:

$$\sigma_{Pi}(R) = \varnothing \text{ if } \forall x \text{ in } R: \neg (p_i(x) \wedge p_j(x))$$

Where p_i and p_j are selection predicates, x denotes a tuple, and $p(x)$ denotes "predicare p holds for x". We now illustrate reduction by horizontal fragmentation using the following example query:

```
SELECT *
FROM EMP
WHERE SALARY=70000
```

The generic query is presented on figure 2a. By commuting the selection with the union operation, it is easy to detect that the selection predicate contradicts the predicates of EMP_1 and EMP_3, thereby producing empty relations. The reduced query is simply applied to EMP_2 as shown on Figure 2b.

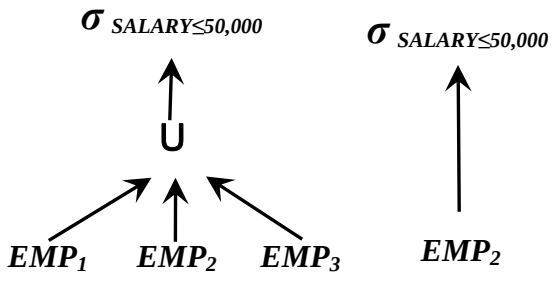

Fig.2 Reduction for horizontal fragmentation

We propose to use theorem proving techniques (this issue is the subject of discussion in Section 5) to determine the contradicting predicates.

4 Architecture of the distributed CORBA-based environment

Figure 3 shows the most important modules that make up a distributed system utilizing CORBA architecture. The system is based on a special three level Client/Server model, which is composed of end-users (clients), core system (client+server) and DB server. The server side is composed of a set of data site in peer-to-peer manner. The core system serves as a main controller and receives operations from every client. Since the components of the DB server on each data site use ODBC API to manipulate the local database, so no restrictions of local DBMS selection exists. Most of relational database products provide ODBC drivers. Using the ODBC layer to access local DB provides standardized access, which achieves more scalability. The user interacts with the system through a user interface that simplifies communication and hides much of the system complexity.

The CORBA architecture is designed to support the distribution of objects implemented in a variety of programming languages. A fundamental part of the CORBA architecture is the Object Request Broker (ORB). An ORB is a software component that mediates the transfer of messages from a program to an object located on a remote network host. The ORB hides the underlying complexity of network communications from the programmer. The primary responsibility of the ORB is to resolve requests for object references, enabling application components to establish connectivity with each other. The major benefit offered by the ORB is its platform-independent treatment of data. The Interface Definition Language (IDL) is used to define interfaces between application components. All components of the architecture and all object type defined in the architecture are described and constructed as modules with interfaces, which are specified in the IDL. CORBA defines a wire protocol for making requests to an object and for the object to respond to the application making the request. The Internet Inter-ORB Protocol (IIOP) ensures interoperability between client application and server based objects. The IIOP runtime communication protocol provides a standard based data representation, which allows the objects to be located anywhere while removing the need for network programming.

The *SQL parser* is the master module of the core system. The parser is generated with famous lex and yacc [33,34,35,36]. *Lex* [33] is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to *Lex*. The *Lex* written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings

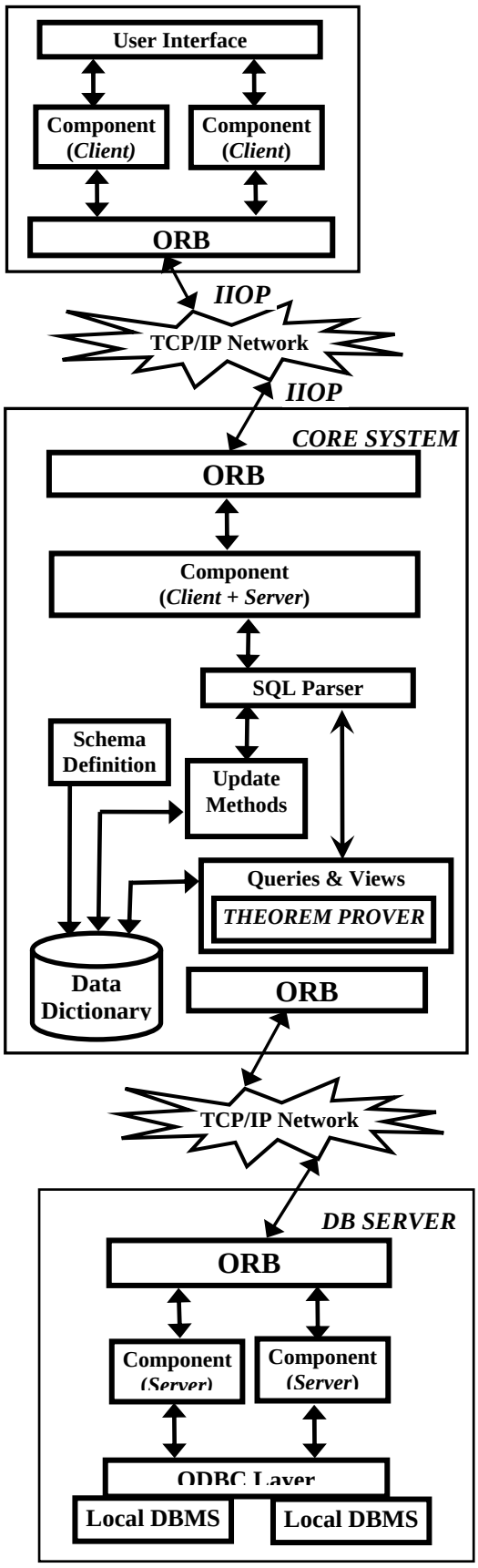

Fig.3 Architecture of the CORBA-based environment

program sections provided by the user are executed. The *Lex* source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by *Lex*, the corresponding fragment is executed.

Yacc [34] provides a general tool for imposing structure on the input to a computer program. The *Yacc* user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. *Yacc* then generates a function to control the input process. This function, called a parser, calls the user-supplied low level input routine (*the lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions. *SQLStatement* is the root rule. When it is called, the parsering is to be completed, all decomposition operation had been finished. So we need to call the correct core system module by the type of SQL statement.

`SQLStatement`:

```
SELECT \ _statement
                    { call Queries & Views module }

| CREATE\ _table\ _statement
                    { call Schema definition module }

| INSERT \_statement
        { call Insert function of Update Methods module }

| DELETE \_statement
        { call Delete function of Update Methods module }
```

A theorem prover could be built in the core of distributed CORBA-based system to provide a deductive functionality of the query and view services during the process of data localization (this issue is the subject of discussion in Section 5).

The *Data Dictionary* will be used at three places:

1. in *Query Decomposition*, we will save the information of a global schema, or we will check the correctness of the user input SQL string, so we need the global schema of each table.

2. in *Data Localization*, we will substitute the table with its fragments. Certainly we must save the information of fragmentation in Data Dictionary.
3. in *Query Optimization*, we need statistics of each fragment. To sum up, we need the global schema, the fragmentation information and the statistics information of fragments. So we can design our Data Dictionary based on above uses.

In a CORBA application, any component that provides an implementation for an object is considered a server, at least where this object is concerned. Being a CORBA server means that the component (the server) executes methods for a particular object on behalf of other components (the clients). A client is a component that consumes services provided by a server or servers. Frequently, an application component can provide services to other application components while accessing services from other components. In this case, the component is acting as a client of one component and as server to the other components (Fig. 4). In fact, two components can simultaneously act as clients and servers to each other. A server is defined as an interface in CORBA IDL. Data passing between the client and the server is defined as IDL structures. Clients communicate with the object through an object reference. When an operation is performed on the object reference, network communication occurs, operation parameters are sent to the server and the actual distributed object executes the operation. It then returns any appropriate data to the client. On CORBA clients, the stubs are the automatically client-side generated code from an IDL interface, it is a local mirror object of its corresponding remote server-object. The stub acts as a proxy for an object that may be implemented by the same process, another process, or on another (server) machine. The skeletons are the automatically server-side generated code from an IDL interface providing the code needed to register and activate an object.

Using CORBA as the infrastructure affects the upper layers of a multi-database system, since CORBA-based environment provides basic database functionality to manage distributed objects. If the most important database-related services (Transaction, Backup and Recovery, Concurrency and Query) are available in the ORB implementation used, it is possible to develop the global layers of a multi-database system on CORBA, mainly by implementing the standard interfaces of these services for the involved objects. A fundamental design issue is the granularity of the CORBA objects. In registering a DBMS to CORBA, a row in a relational DBMS, an object in an object DBMS can be an individual CORBA object. If a whole DBMS is registered as a CORBA object, the functionality needed to process the entities is left to that DBMS. Most commercial DBMSs support the basic transaction and query primitives, either through their Call Level Interface (CLI) library or their XA Interface library. This property makes it possible to define a generic database object interface through CORBA IDL to represent all the underlying DBMSs. CORBA allows multiple implementation of an interface. Hence it is possible to encapsulate each of the local DBMSs by providing a different implementation of the generic database object. CORBA provides three alternatives concerning the association mode between a client request and server method (one interface to one implementation, one interface to one of many implementations and one one interface to multiple implementations). The choice of the alternative is dependent both on the data location and the nature of the database access requests. If the requested data is contained in one database, then it is usually sufficient to use the second alternative and choose the DBMS that manages that data, since DBMSs registered to CORBA provide basic transaction management and query primitives for all the operations the interface definition specifies. If the request involves data from multiple databases, then the third alternative needs to be chosen.

5 Classes hierarchy of the object-oriented theorem prover

Automated theorem proving has long been a concern of computer science and artificial intelligence [15,17]. State-of-the-art theorem provers for first-order logic are highly sophisticated and efficient programs [17,20]. Moreover, they are very flexible tools and can be tuned to a number of applications. First order provers are highly optimised for general-purpose reasoning and are especially optimised for reasoning with equality. They are based on a highly advanced theory of saturation algorithms with redundancy [13,20]. Recently, there have been papers showing how first order provers can be used to reason in theories with a rich definitional structure [3,13,17,21]. However, the use of an explicit set of inference rules and

axioms is not solely applicable to the world of mathematical proofs. Any system that depends on the consistency of its knowledge base and on the

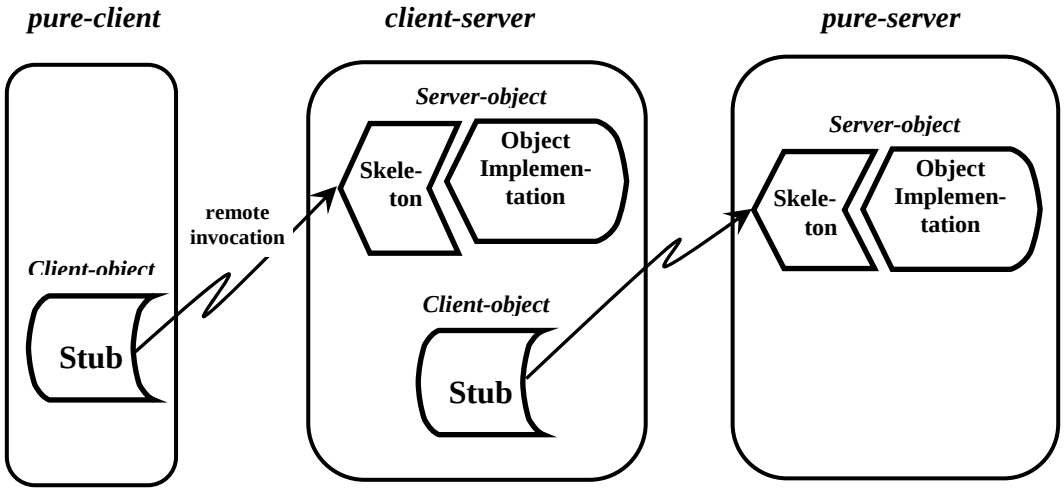

Fig.4 CORBA clients and servers

ability of that base to make deductions also requires a reliable inference mechanism. This theorem prover works on propositional logic statements that have already been translated into clauses of literals "or"ed together. For example, the clause

$$\mathbf{P \vee \neg Q \vee R}$$

is represented by [p, ~q, r]. The resolution principle describes a way of finding contradictions in a database of clauses with minimum use of substitution. Resolution refutation proves a theorem by negating the statement to be proved and adding this negated goal to the set of axioms that are known to be true. It then uses the resolution rule of inference to show that this leads to a contradiction. Once the theorem prover shows that the negated goal is inconsistent with the given set of axioms, it follows that the original goal must be consistent. This proves the theorem. Resolution refutation proof involves the following steps:

- Put the premises or axioms into clause form;
- Add the negation of what is to be proved, in clause form, to the set of axioms;
- Resolve these clauses together, producing new clauses that logically follow from them;
- Produce a contradiction by generating the empty clause;
- The substitutions used to produce the empty clause are those under which the opposite of the negated goal is true.

The resolution rule of inference can be applied only to a formula that is a conjunction of clauses. A clause is a disjunction of literals. A literal is an atom or the negation of an atom. An atom and its negation are referred to as complementary literals. Before the resolution rule can be applied to any formulae all of them have to be converted into equivalent conjunctions of clauses. A conjunction of clauses being true signifies that each of the clauses is true. A formula written as a set of clauses is said to be in clause form. The clauses in the set are said to be derived from the formula. The prover uses a variant of the linear strategy, called ordered linear resolution because it orders the literals of each clause. For example, if "[p,q]" is resolved with "[~q, r]", then it produces "[p, r]". All of these clauses are ordered. The linear strategy is a direct use of the negated goal and the original axioms: take the negated goal and resolve it with one of the axioms to get a new clause. This result is then resolved with one of the axioms to get another new clause, which is again resolved with one of the axioms. This method is based on another method described by Chang and Lee [13,20]. They used the concept of framed literals to keep track of literals

that have been used as a resolvent in a previous step. These literals are saved rather than just deleted as in normal resolution, as knowing what literals have already been used can speed up the resolution process.

Figure 5 summarizes the classes hierarchy of the ordered linear resolution theorem prover which is built in the core system.

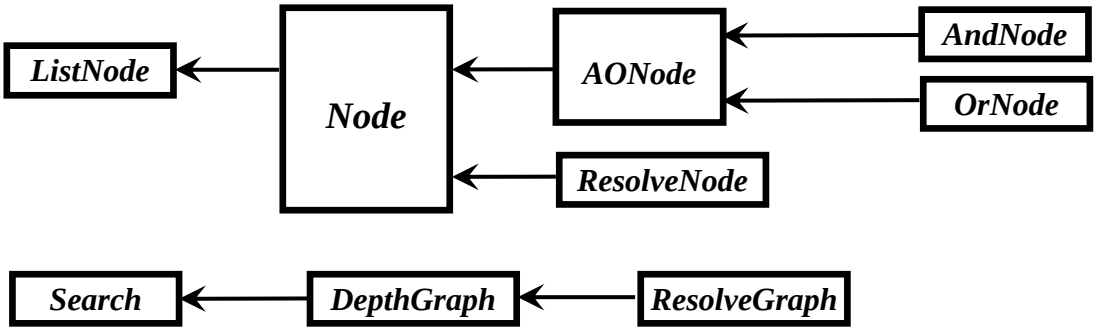

Fig.5 Classes hierarchy of the theorem prover

The ***ResolveNode*** and ***ResolveGraph*** classes set up a structure for performing the ordered linear resolution. They are based on the search library. The ***Search*** class contains the all necessary information for describing a state. The ***Search*** base class is used to keep track of states in the search algorithm. This class implements the linked lists of ***Node ** objects. The ***Node*** base class is derived from the ***ListNode*** class and defines basic states that will be generated during the search process, i.e. it defines the objects that the search space consists of. The ***AONode*** class defines nodes that will be generated in an AND-OR search process. It is an abstract class and derived from the ***Node*** class. The ***AndNode*** and ***OrNode*** classes are derived from the ***AONode*** class. They are used in the process of an AND-OR search. The ***ResolveNode*** class represents the nodes in the search graph. It contains the resolvent and the side clause used with the resolvent to create the next resolvent. If the node is the root node, the resolvent will be the top clause and the side clause will be empty. If the node is a goal clause the resolvent will be empty clause. The ***ResolveGraph*** class implements the search algorithm. The *depth-first* algorithm is chosen. The *depth-first* search immediately goes as deep into the search space as possible in hopes of finding a solution quickly. The ***DepthGraph*** class implements the search algorithm. It is derived from the ***Search*** class. The ***ResolveGraph*** class contains the table consisting of the axioms to be used in the resolution proof. The ***Clause*** class represents clauses as used in resolution. The ***Literal*** class represents literals that are part of a propositional statement or clause. A ***Literal*** object consists of a string, a flag indicating whether the literal is negated or not and a flag indicating whether the literal is framed or not.

6 Conclusion

CORBA is a well-developed architecture for distributed systems. Most of the features required for building a distributed system have been included in it. In this paper, we reported on our on-going project related to distributed databases. In particular, we presented our proposed architecture of a distributed CORBA-based environment and discussed the middleware implementation. The CORBA implementation has several distinct advantages in the area of legacy integration, dynamic interface invocation, load balancing and location transparency. CORBA provides support for dynamic location and integration of information sources while maintaining their autonomy. We proposed to use theorem proving techniques to determine the contradicting predicates during localization layer. An object-oriented theorem prover implementation has been proposed which is used in the database management system to provide a deductive functionality of the query and view services. This theorem prover works on propositional logic statements. The experimental prototype of the toolkit was created. The whole

system is developed in the C++ programming language.

The authors intend to extend the functional capabilities of the toolkit prototype, improving the algorithms incorporated into the theorem prover engine and building the tool into a knowledge-based distributed information system to support the meta-knowledge.

References:

[1] V. Aguilera, S. Cluet, T. Milo, P. Veltri, D. Vodislav, Views in a Large Scale XML Repository, *The VLDB Journal*, Vol. 11(3), 2002, pp. 238-255.

[2] A. Andrade, G. Ruberg, F. Baiao, V. Braganholo, M. Mattoso, Efficiently processing XML queries over fragmented repositories with PartiX, *In: DATAX*, Munich, Germany, 2006, pp.150-163.

[3] S. Andrei, A. Cheng, G. Grigoras, L. Osborne, Incremental Theorem Proving, Information Technologies & Control, Vol.3, 2007, pp. 2-9.

[4] C. Binnig, D. Kossmann, E. Lo, M. T. Özsu, QAGen: Generating Query-Aware Test Databases, *in Proc. ACM SIGMOD International Conference on Management of Data*, 2007, pp. 341-352.

[5] S. Birnam, Distributed *JAVA Platform Database Development*, The Sun Microsystems Press, 2001.

[6] *Borland C++ Builder 5: Developer's Guide*, Inprise Corporation, 2000.

[7] S. Bottcher, M. Jarke, J.W. Schmidt, Adaptive Predicate Managers in Database Systems, *in Proc. of 12th Int. Conf. on VLDB*, 1986, pp. 21-29.

[8] S. Chaudhuri, K. Shim, Optimization of Queries with User-Defined Predicates, *ACM Transactions on Database Systems*, Vol. 24, No 2, 1999, pp. 177–228.

[9] D. Che, K. Aberer, M. T. Özsu, Query Optimization in XML Structured Document Databases, *VLDB Journal*, Vol. 15, No. 3, 2006, pp. 263-289.

[10] A. Deshpande, Z. Ives, V. Raman, Adaptive Query Processing, *Foundations and Trends in Databases*, Vol. 1, No 1, 2007, pp. 1–140. [Online]: http://www.cis.upenn.edu/~zives/research/aqp-survey.pdf

[11] R. Geraghty, S. Joyce, T. Moriarty, G. Noone, *COM - CORBA Interoperability*, Prentice Hall, 1999.

[12] N. Gupta, J. Haritsa, M. Ramanath, Distributed Query Processing on the Web, *In: ICDE, IEEE Computer Society*, 2000, pp. 1-20.

[13] I. Hatzilygeroudis, H. Reichgelt, ACT-P: A Configurable Theorem-Prover, *Data & Knowledge Engineering*, Vol. 12, 1994, pp. 277-296.

[14] M. Henning, S. Vinoski, Advanced CORBA(R) Programming with C++, Addison-Wesley Professional, 2001.

[15] A. Imine, P. Molli, G. Oster, P. Urso, VOTE: Group Editors Analyzing Tool, *in Proc. of 4th International Workshop FTP*, 2003, pp. 179-186.

[16] Z. G. Ives, A.Y. Halevy, D. S. Weld, An XML query engine for network-bound data, *The VLDB Journal*, Vol. 11, No. 4, 2002, pp. 380-402.

[17] M. Kaufmann, J. S. Moore, Some Key Research Problems in Automated Theorem Proving for Hardware and Software Verification, *RAC, Rev. R. Acad. Client Serie A. Mat.*, Vol. 98, No. 1, 2004, pp. 181–195.

[18] G. Kokkinidis, E. Sidirourgos, V. Christophides, Query Processing in RDF/S-based P2P Database Systems, *Semantic Web and Peer-to-Peer*, S Staab, H Stuckenschmidt (eds.), Springer-Verlag, 2006. [Online]: http://139.91.183.30:9090/RDF/publications/sqpeer-chapter.pdf

[19] D. Kossmann, The State of the Art in Distributed Query Processing, *ACM Computer Surveys*, Vol. 32, 2000, pp. 422–469.

[20] B. MacCartney, S. McIlraith, E. Amir, T.E. Uribe, Practical Partition-Based Theorem Proving for Large Knowledge Bases, *in Proc. of 18th Int. Joint Conference on Artificial Intelligence (IJCAI '03)*, 2003.

[21] M. Marinov, An implementation of a theorem prover used in deductive object-oriented databases, *in Proc. of 17th International conference SAER'2003*, 2003, pp. 206-210.

[22] R. Marvie, P. Merle, J. Geib, S. Leblanc, TORBA: Trading Contracts for CORBA, *in Proc. of 6th USENIX Conference on Object-Oriented Technologies and Systems*, 2001.

[23] M. T. Özsu, P. Valduriez, *Principles of Distributed Database Systems*, 2nd edition, Prentice-Hall, Englewood Cliffs, NJ., 1999.

[24] M. T. Özsu, B. Yao, Building Component Database Systems Using CORBA, *In Component Databases*, K. Dittrich and A. Geppert (eds.), Morgan Kaufmann, 2001, pp. 207-236.

[25] D. Suciu, Distributed Query Evaluation on Semistructured Data, ACM TODS, Vol. 27, No. 1, 2002, pp. 1-62.

[26] J. Szymaszek, A. Uszok, K. Zielinski, Building a Scalable and Efficient Component Oriented System using CORBA - Active Badge System Case Study, *Distributed Systems Engineering*, Vol. 5, 1998, pp. 203-213.

[27] F. Tian, D. J. DeWitt, Tuple Routing Strategies for Distributed Eddies, *in: Proc. of 29th VLDB Conference*, Berlin, Germany, 2003.

[28] J. Urban, MPTP 0.1 - System Description, *in Proc. of 4th International Workshop FTP*, 2003, pp. 171-178.

[29] K. Voruganti, M. T. Özsu, R. Unrau, An Adaptive Data-Shipping Architecture for Client Caching Data Management Systems, *Distributed and Parallel Databases*, Vol. 15, No. 2, 2004, pp. 137-177.

[30] N. Wang, K. Parameswaran, D. Schmidt, O. Othman, The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware, *in Proc. of 6th USENIX Conference on Object-Oriented Technologies and Systems*, 2001.

[31] C. Yu, W. Meng, *Principles of Query Processing for Advanced Database Applications*, Morgan Kaufmann, San Francisco, California, 1998.

[32] W. Zhao, L. E. Moser, P. M. Melliar-Smith, Unification of Transactions and Replication in Three-Tier Architectures Based on CORBA, *IEEE Transactions on Dependable and Secure Computing*, Vol. 2, No. 1, 2005, pp. 20-33.

[33] http://www.cs.man.ac.uk/~pjj/cs211/lex/lex.html

[34] http://www.cs.man.ac.uk/~pjj/cs211/yacc/yacc.html

[35] http://dinosaur.compilertools.net/

[36] http://yard-parser.sourceforge.net/cgi-bin/index.cgi