University of Crete
School of Sciences and Engineering
Computer Science Department

QUETE :
QUERY PROCESSING IN DISTRIBUTED DATABASE SYSTEMS

by

HARIDIMOS G.KONDYLAKIS

Master of Science Thesis

Heraklion, February 2006

University of Crete
School of Sciences and Engineering
Computer Science Department

QUETE:
QUERY PROCESSING IN DISTRIBUTED DATABASE SYSTEMS

by
HARIDIMOS G.KONDYLAKIS

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Author:

_____
Haris Kondylakis, Computer Science Department

Supervisory
Committee:

_____
Dimitris Plexousakis, Associate Professor, Supervisor

_____
Grigoris Antoniou, Professor, Member

_____
Anastasia Analyti, Researcher, Member

_____
George Potamias, Researcher, Member

Approved by:

_____
Dimitris Plexousakis, Associate Professor
Chairman of the Graduate Studies Committee

Heraklion, February 2006

QUETE:
QUERY PROCCESSING IN DISTRIBUTED DATABASE SYSTEMS

HARIDIMOS G.KONDYLAKIS

MASTER THESIS

COMPUTER SCIENCE DEPARTMENT,
UNIVERSITY OF CRETE

ABSTRACT

The exponential growth of the web and the extended use of database management systems has brought to the fore the seamless interconnection of diverse and large numbers of information sources. The main problem in such an environment is the heterogeneity between these different sources.

Our essential proposal to resolve the issue of heterogeneity, is finding mappings across schemata and a global reference ontology, the terms of which are used for annotation and querying. By accepting ontology as a point of common reference, naming conflicts are eliminated and semantic conflicts are reduced.

Our contribution is a system that provides an automatic and scalable approach to integrate and then query transparently multiple data sources. It maps automatically semantic queries to SQL and presents the results to the user. Database metadata, are independently captured into XML documents, which also store semantic names for schema elements to identify identical concepts across systems. The query system is capable of handling complex join constructs, and choosing the appropriate attributes, relations and join conditions to preserve user query semantics.

Moreover, since joins across databases are most difficult to handle, two join algorithms were implemented in order to study the efficiency of such a system. The query engine extended to support and exploit horizontal and vertical distribution of database's tables. Those extensions boost the whole system performance when the knowledge of such a distribution exists. Experiments showed that the system has an acceptable performance even in large databases.

Supervisor: Dimitris Plexousakis
Associate Professor

QUETE:
ΕΠΕΞΕΡΓΑΣΙΑ ΕΠΕΡΩΤΗΣΕΩΝ ΣΕ ΚΑΤΑΝΕΜΗΜΕΝΕΣ ΒΑΣΕΙΣ ΔΕΔΟΜΕΝΩΝ

ΧΑΡΙΔΗΜΟΣ Γ.ΚΟΝΔΥΛΑΚΗΣ

ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ,
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

ΠΕΡΙΛΗΨΗ

Η ραγδαία ανάπτυξη του διαδικτύου και η εκτεταμένη χρήση των συστημάτων διαχείρισης βάσεων δεδομένων, έφερε στο προσκήνιο την ανάγκη για την διασύνδεση ποικίλων πηγών πληροφορίας. Κύριο πρόβλημα σε ένα τέτοιο περιβάλλον είναι η ετερογένεια των διαφορετικών αυτών πηγών.

Για την επίλυση του προβλήματος της ετερογένειας η βασική μας πρόταση είναι η ανεύρεση συσχετισμών ανάμεσα στα σχήματα και σε μια οντολογία αναφοράς, οι όροι της οποίας χρησιμοποιούνται για τον σχολιασμό των πηγών και για το σχηματισμό επερωτήσεων που απευθύνονται σ' αυτές. Με την αποδοχή της οντολογίας ως κοινό σημείο αναφοράς οι ονομαστικές συγκρούσεις εξαλείφονται και οι σημασιολογικές διαφορές μειώνονται αισθητά.

Η συνεισφορά μας στον τομέα είναι ένα σύστημα που παρέχει μια αυτόματη προσέγγιση στην ενοποίηση πολλαπλών πηγών πληροφορίας. Η ενοποίηση αυτή είναι διάφανη στον τελικό χρήστη, ο οποίος μπορεί να εκτελεί επερωτήσεις σε πηγές δεδομένων που εξελίσσονται και εμπλουτίζονται συνεχώς. Οι διάφορες σημασιολογικές επερωτήσεις συσχετίζονται αυτόματα με SQL επερωτήσεις οι οποίες απευθύνονται στις ξεχωριστές πηγές. Τα μεταδεδομένα κάθε σχήματος καταγράφονται σε XML έγγραφα, στα οποία αποθηκεύονται και τα σημασιολογικά ονόματα για κάθε στοιχείο των υποκείμενων πηγών. Προσδιορίζονται έτσι τα

ταυτόσημα στοιχεία ανάμεσα στις πηγές. Το σύστημα έχει τη δυνατότητα να χειρίζεται πολύπλοκες συνενώσεις, να επιλέγει τα κατάλληλα γνωρίσματα, τις σωστές σχέσεις και τις απαραίτητες συνθήκες έτσι ώστε να διατηρείται η σημασιολογία των επερωτήσεων του χρήστη.

Επιπλέον, μια και οι συνενώσεις ανάμεσα σε διαφορετικές βάσεις δεδομένων είναι ιδιαίτερα δύσκολες στο χειρισμό τους, υλοποιήθηκαν δυο αλγόριθμοι με σκοπό να μελετηθεί η αποδοτικότητα ενός τέτοιου συστήματος. Η μηχανή επερωτήσεων επεκτάθηκε για να υποστηρίζει και να αξιοποιεί οριζόντια και κατακόρυφη κατανομή σχεσιακών πινάκων. Όταν υπάρχει εκ των προτέρων η γνώση για τέτοιες κατανομές, η απόδοση του συστήματος αυξάνεται κατακόρυφα. Οι μετρήσεις που πραγματοποιήθηκαν έδειξαν ότι το σύστημα έχει αποδεκτή συμπεριφορά ακόμα και σε μεγάλες βάσεις δεδομένων.

Επόπτης: Δημήτρης Πλεξουσάκης
Αναπληρωτής Καθηγητής

Ευχαριστίες

Η εργασία αυτή υλοποιήθηκε και χρηματοδοτήθηκε εν'μέρει στα πλαίσια του έργου "Prognochip" από το Ινστιτούτο Έρευνας και Πληροφορικής του Ιδρύματος Τεχνολογίας και Έρευνας. Έτσι αρχικά θα ήθελα να ευχαριστήσω ολόκληρη την ομάδα Πληροφοριακών Συστημάτων καθώς και το Τμήμα Επιστήμης Υπολογιστών του Πανεπιστημίου Κρήτης για όσα μου προσέφεραν όλα αυτά τα χρόνια και για τις γνώσεις που απέκτησα κατά τις σπουδές μου.

Επιπλέον, θα ήθελα να ευχαριστήσω όλους τους ανθρώπους που με βοήθησαν στην υλοποίηση αυτής της δουλειάς. Ιδιαίτερες ευχαριστίες αξίζουν στον επόπτη μου κ. Δημήτρη Πλεξουσάκη, για όσα μου προσέφερε αυτά τα τρία χρόνια της συνεργασίας μας και για τις ευκαιρίες που μου έδωσε. Χωρίς την ουσιαστική του καθοδήγηση και τις επισημάνσεις του η ολοκλήρωση αυτής της εργασίας θα ήταν αδύνατη.

Θα ήθελα ακόμα να ευχαριστήσω την κ. Αναστασία Αναλυτή γιατί ήταν πάντα διαθέσιμη για συζήτηση και πρόθυμη να με βοηθήσει σε οτιδήποτε ζήτησα την βοήθειά της. Την ευχαριστώ ιδιαίτερα για τις υποδείξεις της.

Ακόμα θα ήθελα να ευχαριστήσω τον κ. Γρηγόρη Αντωνίου καθώς και τον κ. Γεώργιο Ποταμιά για την προθυμία τους να συμμετάσχουν στην επιτροπή για την αξιολόγηση της εργασίας αυτής καθώς και για τις επισημάνσεις τους πάνω στην εργασία μου.

Οφείλω επίσης να ευχαριστήσω την Λύδα Χαράμη ,την Μαίρη και την Χαρά Στεφάνου για τις διορθώσεις τους σε διάφορα τμήματα αυτής εδώ της εργασίας.

Ένα μεγάλο ευχαριστώ ανήκει σε όλους τους συμφοιτητές και συναδέλφους με τους οποίους συνεργάστηκα καθ'όλη την διάρκεια των σπουδών μου. Αισθάνομαι τυχερός που μερικές από τις συνεργασίες κατέληξαν σε πραγματικές φιλίες. Ευχαριστώ λοιπον όλους όσους στάθηκαν πλάι μου όλα αυτά τα χρόνια για τις εμπειρίες που μοιραστήκαμε και θα θυμόμαστε για όλη μας τη ζωη. Ιδιαίτερα θα ήθελα να ευχαριστήσω τον συνεργάτη και φίλο μου Δημήτρη Μανακανάτα για την πολύτιμη συμπαράσταση και την υποστήριξή του, καθώς και για την άψογη συνεργασία μας όλα αυτά τα χρόνια.

Τελευταίο αλλά μεγαλύτερο ευχαριστώ ανήκει όμως στην οικογένειά μου και πιο συγεκριμένα στους γονείς μου Γιώργο και Μαρια και στην αδερφή μου Χαρά που ήταν πάντα δίπλα μου και με στήριξαν σε όλες τις δυσκολίες. Για το λόγο αυτή η εργασία αυτή είναι αφιερωμένη σ'αυτούς και ελπίζω να αποτελέσει μια μικρή ανταμοιβή για τις θυσίες και τις προσπάθειές τους όλον αυτό τον καιρό.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Mediation: a practice under which, in a conflict, the services of a third party are utilized to reduce the differences or to seek a solution. Mediation differs from "good offices" in that the mediator usually takes more initiative in proposing terms of settlement. It differs from arbitration in that the opposing parties are not bound by prior agreement to accept the suggestions made."

-Encyclopedia Britannica

Contents

Data Integration is one of the key problems for the development of modern information systems. The exponential growth of the web and the extended use of database management systems has brought to the fore the seamless interconnection of diverse and large numbers of information sources. An important factor on that problem is the capability to effectively store and process information and to provide access uniformly and efficiently.

In order to provide uniform access to heterogeneous autonomous data sources, complex query mechanisms have to be designed and implemented. The design and implementation of a query mechanism is not trivial because of the heterogeneity of the various components. In information systems, heterogeneity appears for instance in

different structured schemas, different scopes and meanings of schema elements, and different access interfaces. Coping with heterogeneity is always cumbersome. The necessary effort grows with the degree of autonomy of systems being integrated.

Several systems have been implemented in order to integrate heterogeneous databases and to query them. This thesis examines the current distributed query processing proposals, and proposes a framework for answering queries, in environments that integrate heterogeneous databases.

1.1 Motivation

The motivation for this thesis was the integration of two database systems in the project PROGNOCHIP [Potamias G. et al 2005]. The aim of the project was to develop and establish DNA microarray experiments in Greece and the identification and validation of classification and prognosis molecular markers for breast cancer.

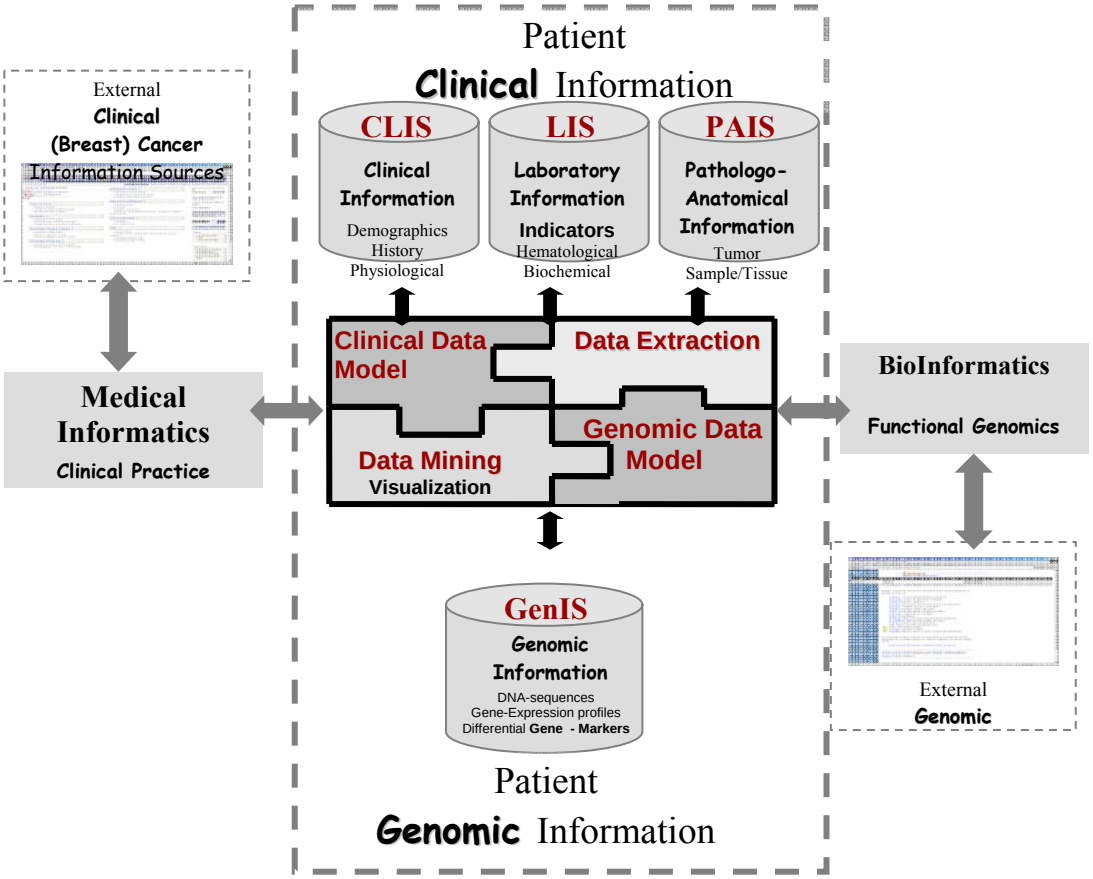


Figure 1. System Overview

Our task was to integrate two information systems as shown in figure 1: The Genomic Information System, that provides storage of microarray experiments, and the Clinical Information System, that provides storage of clinical information about patients. The task was to provide a transparent layer that could enhance knowledge extraction and data exchange between these two systems, which could accept queries from tools and users, and would transparently break queries based on metadata, send them to subsystems and integrate the results returned from them.

The current approach to data source integration is using mediator and wrapper systems, which answer queries across a wide-range of data sources. These systems construct integrated global views, using designer-based approaches, which are mapped using a query language or logical rules into views or queries on the individual data sources. Once an integrated global view and corresponding mappings to source views are logically encoded, wrapper systems are systematically able to query and provide interoperability between diverse data sources.

Unfortunately, mediator and wrapper systems require dedicated database designers and many man-hours of query design and engineering to build a global view for any given multidatabase environment. As a result, database integration is, in many cases, prohibitively expensive and the results are not usually transferable to other multidatabase environments. Further, when data sources are added or removed from the global view, the integration must be performed again.

In our implementation, we try to resolve those boundaries by extending the mediator-wrapper architecture. Moreover, our framework tries to meet several requirements. Some of these are implemented in several systems designed for query answering in distributed database environments, but none of them meets them all:

- The requirement to provide *comfortable access* to all *available* information in each field.
- The capability to perform *queries* without the *knowledge* of the schemas of each database.
- The data could *physically* reside on computers distributed all over the world.
- Data sources would be *heterogeneous* in terms of the access mechanisms they offer, the schemas they use to describe their data, the meaning they

give to schema elements, and the format in which data is eventually provided.

- Data sources could be intentionally and extensionally overlapping. Intention is represented in schemas, whereas extension is represented in instances.
- Data in different data sources could be *inconsistent*.
- Data sources would *evolve* frequently and independently.

The approach to data integration we develop in this thesis is by no means restricted to bioinformatics. On the contrary, it is completely domain independent. However, the motivation for its development was largely taken from problems occurred in Prognochip.

1.2 Contributions

The primary contributions of this thesis are:

- A solution that provides full location, language and schema transparency for users.
- Dynamic integration of large numbers of data sources in evolving environments.
- Standardized Ontology for use across integration domains.
- Capture process performed only once per data source using integration software.
- Automatic global view updating to reflect local database changes.
- Data integration at query time that does not depend on data replication.
- Horizontal, Vertical, and Hybrid fragmentation is highly considered at query execution time.
- Optimization based on fragmentation.
- Dynamic Policy for query answering.
- Quick results in large databases with a high number of joins between them.
- Alternative join implementation for relations that span across databases.

1.3 Organization

This thesis is structured as follows. Chapter 2 is an overview of query processing approaches and techniques used to query multidatabase systems. Then, in chapter 3 the most common integration approaches are shown, and the most important systems used to integrate biological data are presented.

In Chapter 4 we give an overview of the architecture of our system, and we present its basic components. After describing abstractly system's components, we describe the query language used to build queries in QueTe in Chapter 5, and we define its capabilities.

The implementation and the design choices we made are placed in Chapter 6, where also resides the system evaluation. Finally, Chapter 7 concludes the research contributions of the thesis, discusses ways to extend the capabilities of query processing and draws directions for further research work.

Chapter 2

Query Processing

"There can be no understanding between the brain and the hands, unless the heart acts as mediator."

-from the movie "Metropolis"

Contents

Research community has been interested in distributed database systems since the 1970s. Although many ideas had been appeared, distributed database systems were never commercially successful. The main reason for that was the instability of communication technology to ship megabytes of data as required and that large businesses managed to survive without sophisticated distributed database technology by using tapes, diskettes or just paper to exchange data.

The situation today has changed dramatically. Distributed data processing is both feasible and needed. Almost all database vendors offer products to support distributed data processing (e.g., Oracle, Sybase, IBM, and Microsoft) and large database application systems have a distributed architecture. Distributed data processing is feasible because of recent technological advances and is needed because of changing business requirements, which have made distributed data processing cost-effective and in certain situations the only viable option.

Specifically, businesses are beginning to rely on distributed rather than centralized databases because of the cost and the scalability they provide, the capability to integrate different software models, legacy systems that were used and still coexist with modern systems. Furthermore an even growing number of applications have come to rely on distribution technology such as workflow management; tele-conferencing etc. and many companies are forced to reorganize their business in order to remain competitive and more effective.

For the rest of this chapter it is assumed that users and application programs issue queries using a declarative query language such as SQL [Melton and Simon1993] and without knowing where and in which format the data is stored in the distributed system. The goal is to execute such queries as efficiently as possible in order to minimize the time that users must wait for answers or the time application programs are delayed. To this end, we will discuss a series of techniques that are particularly effective to execute queries in today's distributed systems. For example, we will describe the design of a query optimizer that compiles a query for execution

and determines the best possible way among many alternative ways to execute a query. We will also show how techniques such as caching and replication can be used to improve the performance of queries in a distributed environment. Furthermore, we will cover specific query processing techniques for client-server, middleware (multitier), and heterogeneous database and information systems, which represent architectures that are frequently found in practice.

2.1 Research Scope

Over last decades a very large body of work exists in the area of databases. All this work can be roughly classified into work on architecture and techniques for transaction processing, work on query processing, and work on data models, languages, and user interfaces for advanced applications. In this chapter we will focus primarily on query processing. A discussion of transaction processing and of alternative data models is beyond the scope of this work.

This thesis does not intend to give a full coverage of all query processing techniques used today; in fact, a number of query processing techniques for the World Wide Web are not discussed. For instance we will not present the architecture of search engines such as AltaVista. Furthermore there have been several proposals to manage Web sites and query a network of Web Pages [Florescu et al. 1998], to manage and query XML data [McHugh and Widom 1999],[Abiteboul et al. 1999], [Florescu et al.1999]. Instead of going into the details of all these techniques the focus of this chapter is on fundamental mechanisms to process queries that involve data from several sites. We will, therefore, concentrate on structured data and on query languages for structured data, so we will assume that the reader is familiar with basic database system concepts, SQL and the relational data model. Nevertheless, the techniques described in this paper are also relevant to process other kind of data in a distributed environment.

2.2 Distributed Query Processing: Basic Approach and Techniques

In this section we will describe the "text-book" architecture for query processing and present a series of specific query processing techniques for distributed databases and information systems. The purpose of this section is to give an overview of basic mechanisms that can be used in any kind of distributed database system.

2.2.1 Architecture of a Query Processor

The "text-book" architecture was first used in IBM's Starburst project [Haas et al. 1989] .This architecture can be used for any kind of database system including centralized, distributed or parallel systems. In this architecture, queries issued at the system are being translated and optimized in several phases into an execution plan. This plan is being executed in order to obtain the results of the query. Several plans of repeated queries (so called "canned" queries) can be stored in the database and executed by the query execution engine each time this query is issued [Chamberlin et al. 1981].

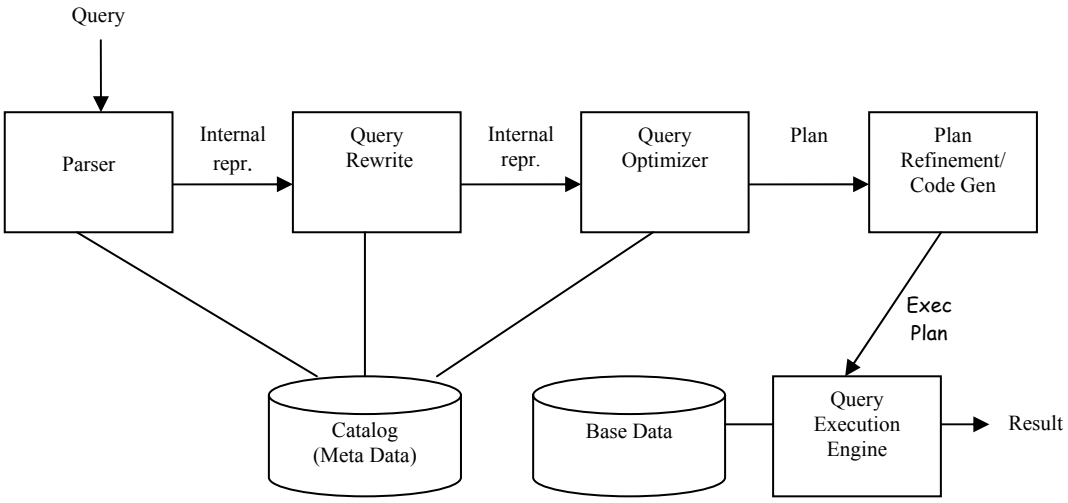


Figure 2. Phases of Query Processing

The components of the "text-book" architecture are shown in the previous figure. At first, the query is issued in the parser component where it is parsed and

translated into an internal representation (e.g., a query graph [Jenq et al 1990], [Pirahesh et al. 1992]), that can be easily processed by the latter phases. Next, the parser query rewriter transforms a query in order to carry out the optimizations that are optimal regardless the state of the system. Typical transformations are the elimination of redundant predicates, simplification of expressions, and unnesting of subqueries and views. In a distributed system, query rewrite also selects the partitions of a table that must be considered to answer the query [Ceri and Pelagatti,1984],[Ozsu and Valduriez, 1999].

The next step is Query Optimizer. This component carries out optimizations that depend on the physical state of the system. The optimizer decides which indices to use to execute a query, which methods (e.g., hashing of sorting) to use to execute the operations of a query and in which order to execute the operations of a query. Moreover it decides how much main memory to allocate for the execution of each operation. In a distributed system, the optimizer must also decide at which site each operation is to be executed. To make these decisions, the optimizer enumerates several alternative plans and chooses the best plan (usually a plan which is not the worst) using a cost estimation model.

Usually in databases, plans are represented as trees, where the nodes are annotated, indicating where the operator is to be carried out. The edges represent consumer – producer relationships of operations. In the Plan Refinement stage, the plan produced by the optimizer is being transformed into an executable plan.

Finally, each operator is implemented by the query execution engine. All state-of-the-art query execution engines are based on an iterator model [Graefe 1993], where operators are implemented as iterators and all iterators have the same interface. As a result two iterators can be plugged together and moreover the results of one operator can be plugged as an input in another operator (pipelining).

The main components cooperate with the Catalog. All the information needed for parsing rewriting and optimizing a query is stored in the+ Catalog. It maintains the schema of the database (i.e. definitions of tables, views, user-defined types and functions, integrity constraints etc.), the partitioning schema (information about what global tables have been partitioned and how they can be reconstructured) and physical information such as the location of replicas, information about indices, and statistics that are used to estimate the cost of a plan. In most relational database systems,

catalog information is stored like all other data in tables. In a distributed database however, the question of where to store the catalog arises. The simplest approach is to store the catalog at one central site, but in wide-area networks, it makes sense to replicate the catalog at several sites in order to reduce communication costs. It is also possible to cache catalog information [Williams et al 1981]. Both replication and caching of catalog information are very effective because catalogs are usually quite small and their information is rarely updated in most environments. However in certain environments, catalogs can become very large and be frequently updated. In such environments it makes sense to partition the catalog and store catalog data where it is most needed.

Of course the architecture described above is not the only possible way to process queries. There is no such thing as a perfect query processor. For example, an alternative architecture has been developed in [Graefe 1995], [Graefe and McKenna 1993], [Graefe et DeWitt 1987] and is used in several commercial database products such as Microsoft's SQLServer. In that architecture, query rewrite and optimization are executed in one phase.

2.2.2 Query Optimization

In this section, we will give a short description of the main techniques used to implement the query optimizer of a distributed database system. First, we will describe the most popular algorithm called "enumeration algorithm" for query optimization.

2.2.2.1 Plan Enumeration with Dynamic Programming

A large number of alternative enumeration algorithms has been proposed in the literature [Steinbrunn et al 1997],[Kossman and Stocker 2000].One of them, which is used in almost all commercial databases, called dynamic programming, is described. The main advantage of this algorithm is that it produces the best possible plans if the cost model is sufficiently accurate. Unfortunately, its main disadvantage is that it has exponential space and time complexity and it is not viable in complex queries. Moreover in distributed environments, the complexity of dynamic

programming is prohibitive for many queries. Several extensions exist with the most popular one being the "iterative dynamic programming", which produces optimal plans, as the ones produced using basic dynamic programming for simple queries, and "as good as possible plans" for more complex ones [Kossman and Stocker 2000].

The basic dynamic algorithm is shown in the following figure and it works in a bottom-up way by building more complex sub-plans from simple sub-plans. In the first step the algorithm builds an access plan for every table involved in the query. Then it enumerates all two-way join plans using the access plans as building blocks. Next the algorithm builds three-way join plans using access plans and two-way join plans as building blocks, e.t.c. The algorithm continues in this way until it has enumerated all n-way join plans which are complete plans for the query, if the query involves n tables.

```
Input: SPJ query q on relations R1,. . . , Rn
Output: A query plan for q
1:      for i = 1 to n do {
2:              optPlan({Ri}) = accessPlans(Ri )
3:              prunePlans(optPlan({Ri}))
4:      }
5:      for i = 2 to n do f {
6:              for all S ⊆ {R1, . . . , Rn} such that |S| = i do {
7:                      optPlan(S) = Ø
8:                      for all O ⊂ S do {
9:      optPlan(S) = optPlan(S) ∪ joinPlans(optPlan(O), optPlan(S - O))
10:     prunePlans(optPlan(S))
11:                     }
12:             }
13:     }
14:     return optPlan({R1, : : : , Rn})
```

Figure 3. Dynamic programming algorithm for query optimization

The beauty of the dynamic programming is that inferior plans are pruned as early as possible. A plan is being discarded if an alternative plan exists that does the

same or more work at a lower cost. Pruning significantly reduces the complexity of query optimization since the earlier inferior plans are pruned and more complex plans are not constructed from such inferior plans.

As things get distributed the decision of which plan must be pruned gets more and more difficult. Two plans may do the same work, but they might produce their results at different sites so shipping time must be considered. In general a plan P_1 may be pruned, if there exists a plan P_2 that does the same or more work and the following criterion holds:

$$\forall\ i \in \text{interesting\_sites}(P_1)\text{: } \mathbf{Cost}\ (\ \text{ship}\ (\ P_1, i\ ) \geq \mathbf{Cost}\ (\ \text{ship}\ (\ P_2, i\ )\ )$$

Here, *interesting_site* denotes the set of sites that are potentially involved in processing the query. This means, that the plan with higher shipping cost shall be eliminated. The concept is formally defined in [Kossman and Stocker 2000] where it is shown that this expression can be evaluated efficiently during query optimization under certain conditions.

2.2.2.2 Cost Estimation for Plans

The classic way to estimate the cost of a plan is to estimate the cost of every individual operator and then sum up these costs [Mackert and Lohman 1986]. In this model, the cost of a plan is defined as the total resource consumption of the plan. In a centralized system the cost of an operator is composed of CPU costs plus disk I/O costs. In a distributed system, communication costs must also be considered. A general formula for determining the total cost can be specified as follows.

$$\text{Total\_cost} = C_{CPU} * \#\text{insts} + C_{I/O} * \#\text{I/Os} + C_{MSG} * \#\text{msgs} + C_{TR} * \#\text{bytes}$$

The two first cost components measure the local processing time, where C_{CPU} is the cost of a CPU instruction and $C_{I/O}$ is the cost of a disk I/O. The communication cost is depicted by the two last components. C_{MSG} is the fixed cost of initiating and receiving a message, while C_{TR} is the cost of transmitting a data unit from one site to another. The data unit is defined here in terms of bytes but could be in different units

(e.g., packets). A typical assumption is that C_{TR} is constant which simplifies query optimization. Thus the communication cost of transferring #bytes of data from one site to another is assumed to be a linear function of #bytes.

$$CC(\#bytes) = C_{MSG} + C_{TR} * \#bytes$$

In general one optimizer will favor plans that carry out operations on fast and unloaded machines and avoid expensive communication links, whenever possible.

2.2.2.3 Response time of Plans

Except from total cost (time), the cost of a distributed execution strategy can be expressed with respect to the response time. When the response time is the objective function of the optimizer, parallel local processing and parallel communications must be considered. A general formula for response time is:

$$Response\_time = C_{CPU} * seq\_\#insts + C_{I/O} * seq\_\#I/Os + C_{MSG} * seq\_\#msgs + C_{TR} * seq\_\#bytes$$

where *seq_* denotes the maximum number of operations which must be done sequentially for the execution of the query. Thus any processing and communication done in parallel is ignored. Minimizing the response time is achieved by increasing the degree of parallel execution. This does not, however imply that the total cost is also minimized. On the contrary, it can increase the total cost, for example, by having more parallel local processing and transmissions. Minimizing the total cost implies that the utilization of the resources improves, thus increasing system throughput. In practice a compromise of those two is desired.

2.2.3 Query Execution

Here we will give a short overview of the alternative ways to execute queries in distributed database systems, how data can be shipped and how joins between

tables stored at different sites can be computed. We will not describe "standard" execution techniques that are commonly used in centralized database systems [Graefe 1993], [Mishra and Eich 1992] and can consequently be used in distributed environments too. We will discuss some of the many options to implement some operators in distributed systems and we will examine how a query optimizer must be extended in order to decide if and how to make use of these techniques for a specific query.

2.2.3.1 Row Blocking

In a distributed environment, communication is typically implemented by send and receive operators. The more messages you send the more resources you are consuming. A good idea is to send fewer messages by sending a lot of tuples in a blockwise fashion instead of sending every tuple individually. This approach is obvious much cheaper than the naïve approach of sending one tuple at a time. Furthermore, the size of the blocks is a parameter that can be regulated according to the characteristics of the network.

One particular advantage of row blocking is that it compensates for burstiness in the arrival of data up to a certain point. If tuples are shipped one by one through the network, any short delay would immediately stop the execution of the query at the receiving site because of shortage of tuples to consume. Due to row blocking, the receive operator has a reservoir of tuples and can feed its parent operator even if the next block of tuples is delayed. As a result, it is often better to choose a block size that it is larger than the message size used by the network.

2.2.3.2 Optimization on Multicasts

It is obvious that communication costs may vary significantly depending on the locations of the sending and receiving sites. Moreover sometimes, a site needs to send the same data to several sites to execute a query. If the network itself does not provide cheap ways to implement multicasts then it is desired to choose the "shortest" paths between sites. Furthermore the load of the sites and their processing capability is a matter that must be considered in order to build the best execution plan.

2.2.3.3 Multithreaded Query Execution

In order to take the best advantage of intraquery parallelism, it is sometimes advantageous to establish several threads at a site [Graefe 1990]. As an example, consider the plan $A_1 \cup A_2 \cup A_3$ where A_1 is stored in Site 1, A_2 is stored in Site 2 and A_3 in Site 3 and the result must be presented in Site 0. If the union and receive operators of Site 0 are executed within a single thread, then Site 0 only requests one block at a time and the opportunity to read and send the three partitions from the three sites is wasted. Only if the union and receive operators at Site 0 run in different threads, they can run and produce tuples in parallel.

However establishing a separate thread for each operator is not the best thing to do every case. This is because the threads need to be synchronized since they use the same shared-memory which adds additional cost to the whole process. Moreover, it is not always advantageous to parallelize all operations and of course not all operations can be executed in parallel. The query optimizer must decide at run time which parts of the query should be run in parallel, and which operators should run in the same thread.

2.2.3.4 Joins with Horizontally Partitioned Data

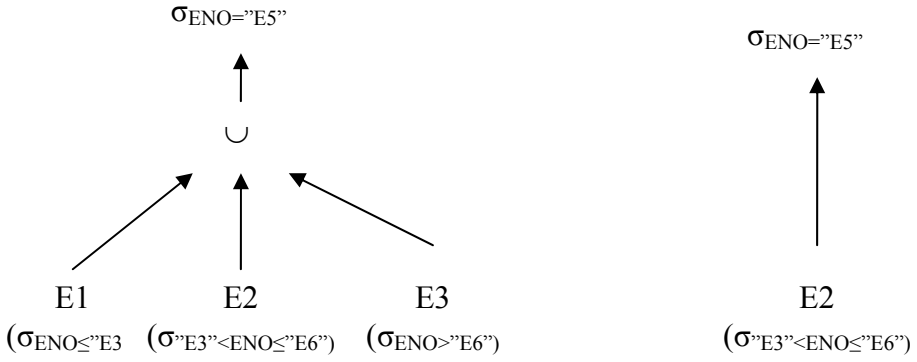
The horizontal fragmentation function distributes a relation based on selection predicates. The reduction of queries on horizontally fragmented relations consists primarily of determining, after restructuring the subtrees, those that will produce empty relations, and moving them. Horizontally fragmentation can be exploited to simplify both selections and join operations.

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 $R_1, R_2, \ldots, R_w$, where $R_j = \sigma_{pj}(R)$, the rule can be stated formally as follows:

$$\sigma_{pj}(R_j) = \sigma_{pj}(\sigma_{pj}(R)) = \emptyset \text{ if } \forall x \text{ in } R: \neg (p_i(x) \wedge p_j(x))$$

Here, p_i and p_j are selection predicates, *x* denotes a tuple, and $p(x)$ denotes "predicate *p* holds for *x*." The rule states that if our select condition does not interest

with the distribution predicate, empty result is produced. For example, in the following figure, the selection predicate ENO="E5" conflicts with the predicates of fragments E1 and E3 and the reduced query is produced after examining the fragmentation.



(a) Generic Query (b) Reduced Query

Figure 4. Reduction with Horizontal Fragmentation

Moreover joins on horizontally fragmented relations can be simplified when the joined relations are fragmented according to the join attribute. The simplification consists of distributing joins over unions and eliminating useless joins. The distribution of join over union can be stated as

$$(R_1 \cup R_2) \rhd\lhd R_3 = (R_1 \rhd\lhd R_3) \cup (R_2 \rhd\lhd R_3)$$

With these transformations, unions can be moved up in the query tree so that all possible joins of fragments are exhibited. Useless joins of fragments can be determined when the qualifications of the joined fragments are contradicting. Assuming that fragments R_i and R_j are defined, respectively, according to predicates p_i and p_j on the same attribute, the simplification rule can be stated as follows:

$$R_i \rhd\lhd R_j = \emptyset \text{ if } \forall x \text{ in } R_i, \forall y \text{ in } R_j : \neg (p_i(x) \wedge p_j(y))$$

The determination of useless joins can greatly reduce the cost of query processing.

2.2.3.4 Semijoins

The theory of Semijoins was defined in [Bernstein, 1981] and was proposed as another technique to process joins between tables stored at different sites. If a table A is stored at Site 1 and table B is stored at Site 2, then the conventional way to execute a join between those tables is to ship A from Site 1 to Site 2 and execute the join at Site 2 (or the other way around). The basic idea of a Semijoin is to send only the columns of A that are needed to evaluate the join predicates from Site 1 to Site 2, find the tuples of B that qualify the join criteria at Site 2, send those tuples to Site 1 and then match A with those B tuples at Site 1. Formally this procedure can be described as:

$$A \triangleright\triangleleft B = A \triangleright\triangleleft (B \triangleright< \pi (A))$$

where $\triangleright<$ is the Semijoin operator

The use of Semijoin is beneficial if the cost to produce and send it to the other site is less than the cost of sending the whole operand relation and of doing the actual join. Several extensions such as like bloom filters [Babb 1979] exist, [Valduriez end Gardarin 1984] but experimental work [Lu and Carey 1985], [Mackert and Lohman 1986] has shown that Semijoin programs are not very attractive for join processing in standard distributed systems because the additional computational overhead is usually higher than the savings in communication costs. However in very specific tasks Semijoin is used with good results.

2.2.3.5 Double Pipelined Hash Joins

Recently, double-pipelined hash-join algorithms were proposed [Ives et al. 1999] ,[Urhan and Franklin 1999]. The basic idea is that in order to execute $A \triangleright\triangleleft B$, two main memory hash tables are constructed, one for tuples of A and one for tuples of B. Initially, the two tables are empty and the tuples from A and B are processed one tuple at a time. To process a tuple of A, the B hash-table is probed in order to find the

B tuples that match this A tuple, A and the matching tuples are then immediately output. After that, the A tuple is inserted into the A hash table for matching B tuples that have not been processed yet. The algorithm terminates when all the tuples of A and B have been processed and is guaranteed to find all the results of the join. Special actions need to be taken if the hash tables grow in such a way that main memory is exhausted, like hybrid hashing and the use of partitioning schemata.

The use of such join algorithms make it possible to deliver the first results of a query as early as possible. In addition such join algorithms make it possible to fully exploit pipelined parallelism and thus reduce the overall response time of the query in a distributed system. Those methods can be used with great advantages in distributed systems where the delivery of tuples through the network is bursty because certain phases of a join processing can be carried out at a site while the site waits for the next, possibly delayed batch of tuples.

2.2.3.6 Top N and Bottom N Queries

In specific cases, Top N or Bottom N queries are posed in database systems. Examples of such queries are "find the ten highest paid employees that work in a research department" or "find the ten researchers that have published the most papers". The goal here is to avoid wasted work when executing these queries by isolating the top N (or bottom N) tuples as quickly as possible and then performing other operations only on those tuples.

In standard relational databases, stop operators have been proposed to isolate the top N and the bottom N tuples [Carey and Kossmann 1998]. The techniques proposed have been developed primarily for centralized databases, but they can be directly applied to distributed systems as well. To give an example, consider a table A that is horizontally partitioned in three sites and we want the top ten tuples of table A. The stop operator in the individual sites makes sure that every site will ship at most ten tuples to the output site, and the stop operator at the output site makes sure that no more than ten query results are produced.

Several algorithms have been proposed in multimedia databases [Chaudhuri and Gravano 1996], [Fagin 1996], or for meta-searching [Gravano and Garcia-Molina

1997], [Gravano et al 1997] but those implementations are beyond the scope of this thesis.

2.3 Client-Server Database Systems

Here we will turn to a specific class of distributed systems with client-server architecture. We will characterise different kinds of those systems and then we are going to give an overview of the crucial questions for query-processing in these systems and we will discuss query optimization and query execution issues. Some of these techniques presented here can be applied in other system architectures too, but they are presented in this section because are mostly used by client-server database systems.

2.3.1 Architectures

In general client-server protocols refer to a class of protocols that allows one site, the client, to send a request to another site, the server, which sends an answer as a response to this request [Tanenbaum 1992]. Using this mechanism, it is possible to implement a variety of different database architectures.

The most general architecture is the peer-to-peer architecture where each node can act both as a client initiating queries and as a server answering them and storing parts of the database.

In a strict client-server environment every node has a fixed role either as a client or as a server. Typically clients do not interact and often servers neither. The clients send queries which are being answered by the servers.

Another type of architecture is the multitier architecture where the sites are organized in a hierarchical way and every site plays the role of a server for the sites of the upper level and the role of a client for the lower level sites. Thus, a site in one of the middle tiers can *only* communicate with its clients at the level above and its servers at the level below.

Several examples of such systems exist, like SHORE [Carrey et al 1994], SAP R/3[Buck – Emden and Galimow 1996] . Most of the commercial database systems

today have strict client-server architecture. Compared to a peer – to – peer architecture, one advantage of a strict separation between client and server machines is that only server machines need to be administrated and security issues can be addressed by controlling the server machines and the client – server communication links. Another advantage is that client and server machines can be equipped according to their specific purposes. Client machines are often PCs with good support for graphical user interfaces whereas server machines are usually more powerful with multiple processors, large disks (RAID), and very good I/O performance. Except from strict client – server architecture multitier architecture can be highly advantageous when we want to integrate functionality provided by different vendors. Scalability can be another reason to use middleware architecture because at every tier, additional sites can be added in order to deal with a heavier load.

In the rest of this section we will describe query processing techniques that are applicable for all three architectures but we will concentrate on the strict client – server architecture and assume that every site has the fixed role of acting either as a client or as a server.

2.3.2 Exploiting Client Resources

The essence of client – server computing is that the database is persistently stored by server machines and that queries are initiated at client machines. The question is whether to execute a query at the client machine which initiated it, or at the server machines that store the relevant data. In other words the question is whether to move the query to the data or to move the data to the query. Another related question is whether and how to make use of caching and store temporarily copies of data at client machines.

The first approach is called *query shipping*. The principle of query shipping is to execute queries at servers. The SQL is shipped from clients to the server machine and the server evaluates the query and sends back to the client the results. In systems with several servers, query shipping works only if there is a middle – tier site that carries out joins between tables stored at different servers or if there are gateways between the servers so that joins across sites, can be carried out at one of the servers.

Query shipping is used in many relational and object – relational database systems today such as IBM DB2, Oracle, and Microsoft SQL Server.

The exact opposite of query shipping is *data shipping*. Here, queries are executed at the client machine at which they were initiated and data is rigorously cached at client machines in main memory or on disk [Franklin et al. 1993]. That is, copies of the data used in a query are kept at a client so that these copies can be used to execute subsequent queries at the client. Caching is typically carried out in the granularity of pages [DeWitt et al. 1990] and it is possible to cache individual pages of base tables and indices [Zaharioudakis and Carey 1997]. Data shipping is used in many object – oriented database systems such as ObjectStore and O_2.

Neither query shipping nor data shipping is the best policy for query processing in all situations. The advantages of both approaches can be combined in a *hybrid shipping* architecture [Franklin et al. 1996]. Hybrid shipping provides the flexibility to execute query operators on client and server machines, and it allows the caching of data by clients. In the following figure this approach is shown.

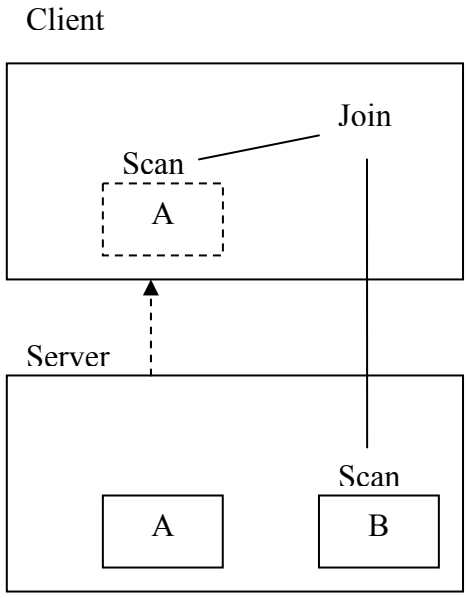


Figure 5. Hybrid Shipping

Here, scan (A) and join operators are carried out at the client, whereas the scan (B) operator is carried out at the server. The scan (A) operator uses the client's cache as much as possible and ships to the client only those parts of A that are not in the cache. In contrast, the scan (B) operator neither uses nor changes the state of the

client's cache. Today hybrid shipping is used in some database products such as UniSQL [D'Andrea and Janus 1996], application systems such as SAP R/3, research prototypes such as ORION-2 [Jenq et al. 1990], KRISYS [Dessloch et al. 1998] and to some extent, in heterogeneous systems such as Garlic [Carey et al. 1995], TSIMMIS [Papakonstantinou et al. 1995] and DISCO [Tomasic et al 1998].

The performance tradeoffs of query, data and hybrid shipping have been studied in extent in [Franklin et al. 1996]. Query shipping performs well if the server machines are powerful and the client machines are really slow. On the negative side, query shipping does not scale well if there are many clients because the servers are the potential bottlenecks in the system. Data shipping scales well because it uses the client machines, but data shipping can be the cause of very high communication costs if caching is not effective and a great deal of unfiltered base data must be shipped to the clients. Obviously, hybrid shipping has the potential to, at least, match the best performance of data shipping and query shipping by exploiting caching and client resources such as data shipping if that is beneficial, or otherwise by behaving like query shipping. In some cases, hybrid shipping will show better performance than both data and query shipping by exploiting client and server machines and intraquery parallelism to execute a query. The price for this improved flexibility is that query optimization is significantly more complex in a hybrid shipping system than in a query or data shipping system because the optimizer must consider more options.

Experiments have shown that in many cases it is better to read data from the server's disks in a hybrid shipping system even if the data are cached at the client. This happens when we have to read and join for example two tables that are already cached at client. If we read the tables from the cache and we try to join them in the same time then concurrently I/Os on the same disk will delay the whole work whereas reading the tables from the server and executing locally the join is the preferable plan. Moreover, sometimes the best strategy to execute query in a hybrid shipping system is to ship cached data or intermediate query results from the client to the server. Such a strategy, for example, is useful in situations in which the data are cached in the client's main memory, the network is fast, and join operations can be carried out most efficiently at the server. Furthermore, transactions that involve small update operations should be carried out at clients, whereas transactions that update large amounts of data should be carried out directly at servers. The advantage is that small

transactions can be rolled back at clients without affecting the server and that updates can be propagated to the server in one batch with fairly little overhead. [Bogle and Liskov 1994], [O'Toole and Shira 1994].

2.3.3 Query Optimization

Having described the fundamental different approaches for query processing, we will now show how query optimizers for query, data and hybrid shipping systems can be built and describe several alternative query optimization strategies.

2.3.3.1 Site Selection

From the perspective of a query optimizer, data shipping, query shipping and hybrid shipping can be modelled by the options they allow for site selection. So every operator of a plan has a site annotation, which indicates where the operator is to be executed. For example, display operators that pass the results of select queries to application programs need to be carried out at the client which issued the query. For all other operators such as updates, joins, scans, sorts, group by, etc the approaches are different according to which model we are using. Data shipping carries out all operations at the client, whereas query shipping carries out all the operations at servers. Hybrid shipping allows the optimizer to annotate operations in any way allowed by data or query shipping.

All site annotations are logical. A client site annotation indicates that the operator is to be carried out by the client that issued the query. Such an annotation does not indicate that the operator is carried out by a specific machine. Likewise, a consumer annotation indicates that the operator is carried out at the same site as the operator that processes the operator's results. A server annotation for a scan indicates that the scan is carried out at one of the servers that store a copy of scanned data. A server annotation for an update indicates that the update is carried out at all the servers that store a copy of the affected data (read – one – write – all ROWA is assumed). These logical site annotations are translated into physical addresses when a plan is prepared for execution. As a result the same plan can be used to execute a query at different clients so that a query need not be recompiled for every client individually. If

there is replication, translating a server annotation for a scan involves selecting one specific server machine which can be done heuristically or based on a cost model.

2.3.3.2 Where and When to Optimize

The two main questions in query optimization are *where* and *when* a query should be optimized. The *where* question was extensively studied in [Hagmann and Ferrari 1986], in an environment with many clients and one server. They proposed carrying out certain steps of query processing at the client at which a query originates and other steps at the server. For example, parsing and query rewrite could be carried out at the client whereas query optimization and plan refinement could be carried out at the server. This approach makes sense because operations that can easily be executed at clients do not disturb the server whereas steps that require a good knowledge of the current state of the system should be carried out by the server. In systems with many servers, no single server has complete knowledge of the whole system so a server is chosen to carry out optimization. This server needs to either guess the state of the network and other servers based on statistics on the past, or try to discover the load of other servers by asking them for their current load. While asking is obviously better than guessing, asking involves at least two extra messages for every server that is potentially involved in a query.

The answer to the second question determines the accuracy of the information about the state of the system that the optimizer receives. This question arises for canned queries that are part of application programs and evaluated during their execution. As already stated, the traditional approach is to compile and optimize these queries at the time the application program is compiled, store plans for these queries in the database, and retrieve and execute these plans whenever the application program is executed. When something drastic happens, it makes the execution of the plan impossible (for example when an index is dropped) the plan stored in the database is not valid any more, and a new plan must be generated before the application program is executed [Chamberlin et al. 1981]. Obviously, this approach cannot adapt to changes such as shifts in the load of sites, and the precompiled plans show poor performance in many situations.

More dynamic approaches were proposed in [Graefe and Ward 1989] , [Cole and Graefe 1994], [Ioannidis et al 1992]. The idea is to generate several alternative plans and subplans at compile time, store these alternative plans and subplans in the database, and choose the plan or subplans that best matches the current state of the system just before executing the query. Even more dynamic approaches optimize queries on the fly. The idea is to start executing a compiled or dynamically chosen plan and observe whether intermediate query results are produced and delivered at the expected rate. If the expectations are not met, the execution of the plan is stopped, intermediate results are materialized and the optimizer is called to find a new plan for those parts of the query that still need to be carried out. In [Uhran et al. 1998] is shown how useful can be a reoptimization like that to improve the response time, in situations in which the arrival of data from certain servers is delayed or bursty because those servers are heavily loaded or the communication links are congested. For this purpose the approach reorders and reschedules operations at the client so that the client carries out other operations while waiting for the delayed data. In [Kabra and DeWitt 1998] is shown how such a reoptimization approach helps in situations in which the initial plan performs poorly because it was based on wrong estimates of the size of tables and intermediate query results.

In [Ozcan et al. 1997], another dynamic on the fly query optimization approach is proposed. In that approach queries are optimized and executed in two phases. First, the query is decomposed and it is divided into a set of subqueries that can each be executed by a single server. The final query result is composed by joining the results of the subqueries by the client or a middle-tier machine. Query decomposition for this purpose is described in [Evrendilek et al 1997]. The subqueries are processed by the servers in parallel. The order in which the results of the subqueries are joined at the client depends on the speed in which the servers produce subquery results and the selectivity and cost of joins which need to be carried out to combine the subquery results. Heuristic approaches can be used to decide whether to join the subquery results produced in two fast servers immediately or to delay a join and wait for the delivery of other subquery results from a slower server, first. The goal is to parallelize work at the client with work at slow servers as much as possible, and also to avoid the execution of very expensive joins that may result from poor join ordering.

2.3.3.3 Two Step Optimization

Two step query optimization is an approach that has become popular for both distributed and parallel database systems [Du et al. 1995], [Gangulu et al.1996], [Hasan and Motwani 1995], [Stonebraker et al. 1996], [Thomas et al. 1995]. Two step optimization is an alternative to the dynamic approaches presented in the previous section because it carries out certain decisions just before a query is executed. Two step optimization also reduces the overall complexity of distributed query optimization. Several variants of two – step optimization exist.

For distributed systems, the basic variant of two – step optimization works as follows. At compile time, a plan is being generated that specifies the join order, join methods and access paths. Every time just before the query is executed, the plan is transformed and site selections are carried out. All the steps can be carried out by dynamic programming or any other enumeration algorithm. Two – step optimization has a reasonable complexity because both steps require reasonable effort. The first step has essentially the same, mostly acceptable, complexity as query optimization in a centralized database system. The second step also has acceptable complexity because it only carries out site selection.

Moreover, two – step optimization is useful to balance the load on a distributed system because executing operators on heavily loaded sites can be avoided by carrying out site selection at execution time [Carey end Lu 1986]. Two – step optimization is also useful to exploit caching in a hybrid shipping system because query operators can dynamically be placed at a client if the underlying data is cached by the client [Franklin et al. 1996]. On the negative side, two – step query optimization can result in plans with unnecessarily high communication cost because in many cases the first step ignores the location of data and the impact of join ordering on communication cost in a distributed system.

2.3.3 Query Execution Techniques

Most of the techniques presented in section 2.2.3 are useful in a client – server environment as well as any other distributed database system. Row blocking for

example, is essential to ship data from servers to clients and vice versa and it has been implemented in almost all commercial systems.

One particular issue that arises in hybrid shipping systems is how to deal with transactions that first update data in a client's cache and then execute a query at a server that involves the updated data. For example, consider a transaction that first updates the salary of one employee and then asks for the average salary of all employees. The update is likely to be executed at the client at which the transaction was started in order to batch updates as described in a previous section. On the other hand the optimizer will probably decide to execute the second query at the server that stores all the data needed in employee's table in order to avoid the cost of shipping the whole table to the client. The point is that the computation of the average salary must consider the new salary of the updated employee, which is known to the client but not to the server. Two possible solutions have been proposed here.

The fist solution is to propagate all relevant updates such as employee's new salary to the server just before starting to execute the query at that server [Kim et al. 1990] and the second one is to carry out the query a the server and then pad the results returned by the server at the client using the updated values [Srinivansan and Carey 1992]. In either case, carrying out the query at the server involves additional costs that should be taken into account by a dynamic or two – step optimizer in order to decide whether it is cheaper to carry out the query at the server or at the client. Such issues do not arise in query shipping and data shipping systems. Query shipping systems do not support client-side caching and batched updates, and data shipping systems carry out all operators at the client using the latest cached versions of data.

2.4 Heterogeneous Database Systems

This section gives an overview of how queries can be processed in heterogeneous database systems. The purpose of such systems is to enable the development of applications that need to access different kinds of component databases (e.g. multimedia databases, relational, object oriented, xml databases). One characteristic of heterogeneous database systems is that the individual component databases can have different capabilities to store data, carry out database operations,

and communicate with other component databases of the system. One of the challenges therefore, is to find query plans that exploit the specific capabilities of every component database in the best possible way and to avoid query plans that attempt to carry out invalid operations at a component database. Another challenge is to deal with semantic heterogeneity, which arises for example, if several components use the same term but they mean different things. Furthermore, every component database has its own specific interface (API), decides autonomously when and how to execute a query, and might not be designed to interact with other databases.

There has been done a great deal of work on various aspects of heterogeneous databases. There have been issued excellent tutorials in the past [ACM Computing Surveys 1990], and a lot of commercial systems. In this section therefore we will concentrate on basic technology and recent developments in this area.

2.4.1 Wrapper Architecture

In order to construct heterogeneous database systems, several tools have been developed in recent years. Examples are DISCO [Tomasic et al.1998], Garlic [Carey et al.1995], Hermes [Adali et al. 1996], TSIMMIS [Papakonstantinou et al.1995], Pegasus [Shan et al. 1994], Junglee's VDB [Gupta et al. 1997]. Furthermore a number of tools have been designed for the specific purpose of integrating data from different relational and object oriented databases (IBM's data joiner etc). Essentially all of these tools have a three – tier software architecture as shown in the figure on the next page.

Clients connect to a mediator [Wiederhold 1993]. The mediator parses a query, carries out query rewrite and query optimization, and executes some of the operations of a query. The mediator also maintains a catalog to store the global schema of the whole heterogeneous database system (i.e. the schema used in queries by application programs and users), the external schema of the component databases (i.e. which parts of the global schema are stored by each component database), and statistics for query optimization. Thus, the mediator has very much the same structure as the "textbook" query processor described in the beginning of this chapter. The difference is that an extended query optimization approach needs to be used and that certain query execution techniques are particularly attractive in the mediator that

might not be attractive in other distributed database systems. Also, in most cases, a mediator is designed to integrate any kind of component database. That is, a mediator does not contain any code that is specific to any one component database and as a result a mediator cannot directly interact with component databases.

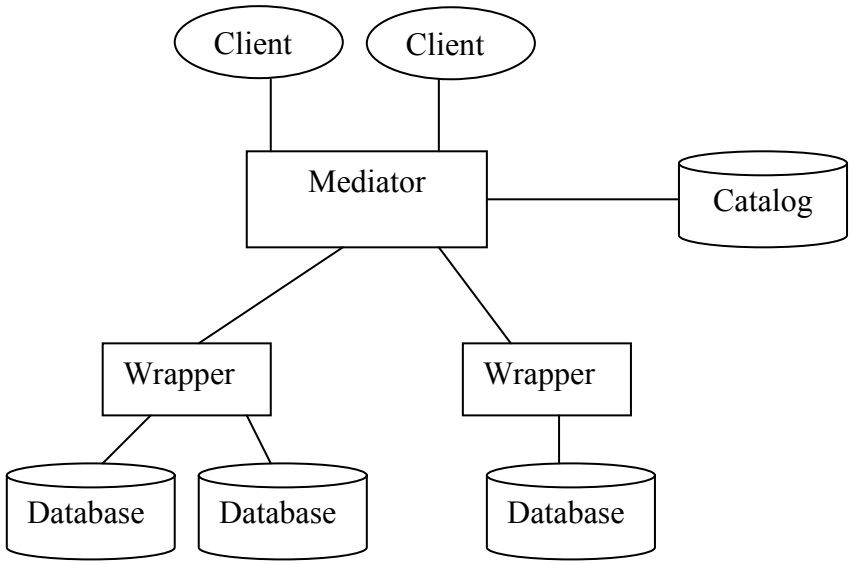


Figure 6. Wrapper Architecture

To encapsulate the details of component databases, a wrapper is associated to every component database. The wrapper translates every request of the mediator so that the request is understood by the component database API, and translates the results returned by the component database so that the results are understood by the mediator and are compliant with the external schema of the heterogeneous database. In some cases, wrappers also implement special techniques such as row blocking or caching to improve performance. In addition, wrappers may participate in the optimization process.

Obviously wrappers are fairly complex pieces of software, and it is not unusual for it to take several months to develop one. The TSIMMIS and Garlic projects have specifically addressed the question of how to make wrapped design as cheap as possible. Similar wrappers work for many different kinds of component databases and it is quite easy in most cases to adjust an existing wrapper in order to obtain a wrapper for a new component database. Moreover, it is possible for several component databases to be handled by the same wrapper as shown in the previous

figure. Furthermore, the architecture is extensible which means that at any time, wrappers and component databases can be upgraded or new component databases can be integrated without changing the mediator or adjusting existing wrappers. They can be installed at any machines in the system and they even can be distributed in several machines. It is quite likely that in the near future wrappers will be commercially available for many common classes of databases.

2.4.2 Query Optimization

One of the challenges of query optimization in heterogeneous database systems is that the capabilities of component databases are different. The optimizer of a heterogeneous system must therefore be generic and be able to understand what capabilities, component databases have. Several alternative approaches for query optimization in heterogeneous database systems have been proposed in the literature. One approach is to describe the capabilities of the component databases as views, store the definitions of these views in the catalog, and see during optimization how a query can be subsumed by these views [Levy 1999]. While this approach is flexible, it is very difficult to implement successfully. Other work has proposed the use of capability records [Levy et al. 1996] or context – free grammars to describe the capabilities of queries and the use of various new cost – based and heuristic algorithms to generate plans for a query [Papakonstantinou et al. 1996], [Tomasic et al. 1998]. In this section we will focus on the approach where the capabilities of the component databases are described by enumerating rules, which are interpreted by the optimizer, and this approach uses either dynamic programming in order to find a good plan or iterative dynamic programming in order to find a good plan with reasonable effort [Haas et al. 1997]. This approach was implemented in IBM's system Garlic.

2.4.2.1 Plan Enumeration with Dynamic Programming

The idea of plan enumeration is quite simple. Every wrapper provides a set of planning functions, which are called by the optimizer's *accessPlan* and *joinPlan* functions in order to construct subplans, which can be handled by the wrapper and its component databases. In other words, query optimization is carried out using the same

dynamic programming based algorithms as described before with the only difference being that accessPlan and joinPlan functions call planning functions defined by wrappers developers in order to enumerate subplans rather than constructing such subplans themselves.

plan_access(T, C, P) = **R_Scan** (T, C, P, ds(T))
ds(T) returns the ID of the relational component database that stores T

Figure 7. Access plan enumeration rule

Conceptually, planning functions can be seen as enumeration rules. The figure above shows the plan_access rule of a wrapper for relational component databases. This rule generates an R_Scan operator to read table T from the component database that stores T (i.e. ds(T)), apply predicates P to the tuples of T, and project out columns C of T. This rule is called by the optimizer's accessPlan function for every table used in a query that is stored by a component database which is associated to the relational wrapper. Consider for instance the following query:

```
SELECT e.name, e.salary, d.budget
FROM Emp e, Dept d
WHERE e.salary > 100.000 and e.works_in = d.dno;
```

If both Emp and Dept are stored in the relational database D then the plan_access rule of the figure is instantiated twice as follows

plan_access(Emp, {salary, works_in, name},{salary > 100,000})=
R_Scan (Emp, {salary, works_in, name},{salary>100,000}, D)
plan_access (Dept, {dno, budget},{})=
R_Scan (Dept, {dno, budget},{}, D)

The R_Scan operator generated with every application of the plan_access rule is specific to and used internally by the relational wrapper; neither other wrappers nor the mediator need to know about the existence or semantics of such an R_Scan operator. Likewise, the relational component databases do not need to know about

R_Scan operators. To execute plans that involve R_Scan operators the wrapper translates R_Scan(T,C,P,D) into "select C from T where P" and submits this query to the relational component D.

Just like wrappers, the mediator provides a set of rules that enumerates portions of plans that are to be executed by the mediator. For example, the mediator provides a rule that says that any kind of join can be carried out by the mediator, regardless of where the tables involved in the join are stored. So an Emp $\triangleright\triangleleft$ Dept operation could be carried out by the mediator or by the relational component database. The optimizer enumerates both alternatives by calling the mediator and wrapper join enumeration rules, and the overall cheaper plan is selected.

The full details of the algorithm can be found in [Haas et al. 1997]. Having presented the basic idea, we will briefly summarize the major advantages of this approach.

This approach relies on well established distributed database technology which gives vendors an easy migration path to adapt for their products. The use of dynamic programming or iterative dynamic programming will generate good plans with reasonable effort just as in any other distributed database system. Moreover this approach is very flexible since the capabilities of the component databases can be modeled very accurately by writing simple enumeration rules that might fit in several databases. Those enumeration rules and planning functions for wrappers can be very simple and easily implemented because the enumeration rules describe the kind of operations that can be carried out rather than exactly how these operations are implemented. Finally it is possible to define very simple enumeration rules at the beginning and to add more sophisticated enumeration rules, or even change the rules once the wrapper is operational.

2.4.2.2 Cost Estimation for Plans

Having described how alternative query evaluation plans can be enumerated in a heterogeneous database system, we now turn to the question of how to estimate the cost or response time of these plans.

Both the classic and response time approach presented in previous sections can be used for this purpose, and the cost or response time of the individual operators that

are to be carried out by the mediator can be estimated just as in any other distributed database system. This is because the mediator uses standard, well-understood algorithms to execute joins, group – bys and so on. The challenge is to estimate the cost or response time of wrapper plans that are to be carried out by the component databases because the details of how a component database executes such a plan might not be known. Estimating the cost of wrapper plans in heterogeneous database systems is still an open research issue. There are three alternative approaches, which differs in the accuracy of the estimates and in the amount of required effort by wrapper developers.

The first one is called *Calibration* approach. The idea is to define a generic cost model for all wrappers and adjust certain parameters of this cost model for every individual wrapper and component database by executing a set of test queries. This way, the specific hardware and software characteristics of a wrapper and a component database can be taken into account. For example, a very simple generic model would be to estimate the cost of a wrapper plan as C*N where N is the estimated number of tuples returned and C is the wrapper specific parameter which would be small for very fast components and large for slow component databases or component databases that are only reachable by a slow communication link. Several generic cost models have been proposed to implement the calibration approach [Du et al. 1992], [Zhu and Larson 1994], [Gardarin et al. 1996], [Roth et al. 1999] and they are significantly more complex than the simple example given above. The big advantage of the calibration approach is that wrapper developers need not worry much about costing issues when they design a new wrapper. The generic cost model is predefined as part of the mediator, and the calibration of the generic cost model for a new component can be carried out automatically or semi – automatically. The big disadvantage of the calibration approach is that not all components databases can be tweaked into a generic cost model.

An alternative to the calibration approach is to define a separate cost model for every wrapper. In this approach, the developer of the wrapper not only provides enumeration rules as described in previous section, but also a set of cost formulas. One cost formula is associated with every enumeration rule in order to estimate the cost of the plan generated by that rule. Obviously the advantage of this approach is

that the cost of all wrapper plans can be modeled as accurately as possible or desired. However a heavy burden is put on the developers.

Finally the third approach to estimate the cost of wrapper plan is based on monitoring the system and keeping statistics about the cost to execute wrapper plans [Adali et al. 1996]. Like the calibration approach this one releases wrapper developers from the heavy burden of worrying about costing issues, but it can be very inaccurate. One particular advantage of this approach is that it automatically and dynamically adapts to changes in the system that impact the cost of operations.

2.4.3 Query Execution

In this section we are going to describe two techniques that are commonly used in executing queries in heterogeneous database systems. Of course all the techniques described in previous sections are applicable here but wrappers and component databases have usually limited capabilities which restrict the possible ways to execute a query. For instance, two component databases may not be capable of participating in a Semijoin program with duplicate elimination, or it may not be possible to place query operators at component databases (operators must be translated into queries that are understood by the components databases).

The first technique simulates a nested – loop join in a heterogeneous system. This technique exploits the fact that many component databases take input parameters as part of their query interfaces. To illustrate how bindings can be exploited for query processing consider a heterogeneous system with two relational component databases D_1 and D_2, that store tables A and B respectively. One way to execute A $\triangleright\triangleleft$ B with join predicate A.x=B.y would be firstly to ask the mediator D_1 to execute the following query in order to scan table A.

Select * from A

The wrapper of D_1 then will return the tuples of table A to the mediator, one by one or in blocks using row blocking. For every tuple of A the mediator asks the wrapped of D_2 to evaluate the following query in order to find the matching B's:

```
select * from B where B.y=?
```

Here "?" denotes the binding parameter and is instantiated with the A.x value of the current tuple of A. This approach shows good performance if A is fairly small or a predicate restricts the number of tuples of A that need to be probed. This approach is also useful because it might be the only possible way to execute the join. Certain component databases accept blocks of tuples as parameters which can be exploited to process joins by passing a block of tuples to the outer table or even the whole outer table to the component database. Since this blocking reduces the number of messages it is usually significantly faster than the tuple at a time approach and should be used where possible.

Except from bindings, cursor caching is another technique. There are many workloads for which the mediator submits the same query, with different parameters, many times to a component database. The idea of cursor caching is to optimize a query only once in order to reduce the overhead of submitting the same query to the same component database again and again. For component database systems that understand JDBC, cursor caching can be implemented by using JDBC's *prepareStatement* command to optimize the query, the *set* command to pass the binding parameters every time the query is executed and the *executeQuery* command to execute the query. Cursor caching is extensively used in systems such as SAP R/3, Oracle e.t.c. Of course, cursor caching has the same tradeoffs as static query optimizations since a cached plan may not be always the best plan to execute.

2.5 Dynamic Data Placement

The previous three sections answered the following question: Given a query and the location of copies of data and other parameters, how can this query be executed in the cheapest of fastest possible way. In this section we will look at this question from a different perspective and show where copies of data should be placed in a distributed system so that the whole query workload can be executed in the cheapest or fastest possible way.

Traditionally, data placement has been carried out statically. With static data placement, a system administrator decides where to place copies of data, speculating what type of queries might be carried out at what locations in the system. Obviously, static data placement has several weaknesses since most of the time the query workload is not predictable. Moreover even if the workload could be predicted it would be expected to change and in many cases so quickly that the administrator would be unable to keep up with the changes. Moreover the complexity of a sufficiently accurate model for static placement is too big (*N-P* complete [Apers 1988]). This section is therefore, focused on dynamic data placement approaches which keep statistics about the query workload and automatically move data and establish copies of data at different sites in order to adjust the data placement to the current workload. Those approaches do not aim to be perfect, but they try to improve the data placement with every move. Concurrency control and consistency are not addressed here nor techniques that place copies of entries of the catalog at different sites [Eickler et al. 1997].

2.5.1 Replication vs. Caching

In principle there are two different mechanisms to establish copies of data at different sites of a distributed system: caching and replication. Whereas they share the same goal in order to reduce communication costs and balance the system load, there are a number of subtle differences between them.

First, replication takes effect at server machines in a client-server environment. Replication establish copies of data at servers based on statistics that are kept with the purpose of better meeting the requirements of a potentially large group of clients. Caching on the other hand, takes effect at clients or at middle – tier machines and caching is based on statistics kept on these machines. Only one client or a small group of clients, therefore, benefit from a cached copy of a data item, but it establishes copies of data where the data is needed. Also, caching exploits client machine resources which might remain unused without caching.

Moreover replication is typically coarse – grained. Only a whole table, a whole index, or a whole partition of a table or index can be replicated. Replicating data in a coarse granularity is acceptable because a large group of clients benefit from

replication and it is quite likely that most parts of a table or index will be used by this group of clients. Caching on the other hand, is typically fine – grained: Individual pages of a table or index can be cached by a client machine, and some systems even allow the caching of individual rows of a table. Caching in a fine granularity is important because caching supports the queries of a single client or a small group of them, and clients are usually interested in a small fraction of the whole data.

Usually replication decisions are more long-term than caching decisions. That is because replication is intended to support a large group of clients whose overall access behavior does not change as rapidly as the access behavior of a single client. Replication typically involves placing data on servers' disks, whereas a client's working set of data typically fits in the client machine's main memory. Server replicas are registered in the system's distributed catalog so they can be used by all clients, while caching does not affect the catalog. Propagation – based protocols are used to keep replicas of data consistent and accessible at servers all the time. For caching on the other hand, it was shown that the best way to maintain consistency is to use a protocol that is based on invalidation, and removes out of date copies from client's cache so that copies of data are only available in a client's cache as long as they have not been updated [Franklin et al. 1997]. Furthermore replicas are kept at servers until they are explicitly deleted whereas copies of data are kept in a client's cache until they are replaced by copies of other and more interesting data using a replacement policy such as LRU.

The last difference between replication and caching concerns the mechanism used to establish copies of data. Replicas are established by a separate process that copies a table, index, or partition and moves it to the target server. Caching on the other hand is a by-product of query execution. When a table scan or index scan is executed at a client, the client fault in all the pages of the table or index that the client has not cached and, after the scan is complete, the client keeps all the used pages of the table or index in its cache, if the cache is large enough. As a consequence, caching decisions need to be made by the query processor while replication decisions can be made by a separate component that is established at every server and works independently.

To conclude, there is no more useful technique between caching and replication. They are complementary and they should be both implemented.

Replication helps to move data near to a large group of clients so that these clients can access the data cheaply the first time they need the data. Caching makes it possible to access data cheaply when data are used repeatedly by the same client even when we have server failure.

Several dynamic replication algorithms have been proposed in the literature [Bestavros and Cunha 1996], [Sidell et al. 1996], [Wolfson et al. 1997] and can be roughly classified in two groups. In algorithms that try to reduce communication costs in a WAN by moving copies of data to servers that are located near clients, and in algorithms that try to replicate "hot" data in order to balance the load of servers in a LAN or in an environment which communication is cheap. Just like replication, a lot of algorithms have been proposed for dynamic caching too. The most common algorithm is called "cache investment" and fully analyzed in [Kosssman et al. 2000].

2.5.2 View Caching, View Materialization and Data Warehouses

So far we assumed that only base data can be cached and replicated (i.e. base tables or indices or parts of them). We will now illustrate systems that cache or replicate (i.e. materialize) derived data or views. Such systems could for example, cache the average salary of all *Emps* that work in a research department instead of or in addition to the complete *salary* information of all *Emps*.

View caching and materialization has been addressed in a number of research projects [Desphpande et al. 1998], [Dessloch et al. 1998] and view materialization has also be implemented in Oracle [Bello et al. 1998]. Data warehouses are the most prominent example of commercial systems that materialize and cache views [Widom 1995]. Data warehouses are typically established for decision support in companies or as product catalogs and classified ads for electronic commerce on the web. They are usually installed in a three – tier environment and they are located in the middle tier, which is connected to one or more data sources , and it keeps materialized views over the base data stored at those data sources. Its role is to answer queries from clients without interacting with data sources. From our narrow perspective, in a data warehouse, the data sources and the clients are part of a distributed system in which views are materialized or cached in the warehouse.

Compared to the replication and caching of base data, the benefits of materializing and caching views are significantly larger. Caching the result of a join for example, might completely eliminate the cost of join or group-by processing for subsequent queries in addition to savings in communication costs and potential load balancing effects. View caching and view materialization are significantly more complex to implement. That is because keeping cached or materialized views consistent in the presence of updates is complex and often expensive [Quass and Widom 1997], and it is unclear how invalidation based protocols, which have proven to be very useful to implement cache consistency, can be applied to view caching. Cache investment can be used but there is an explosion in the number of "what-if" analyses that need to be carried out for every query so that a naïve application of cache investment is impractical. Moreover query optimization is more complicated and more expensive in the presence of cached and materialized views [Levy 1999] because the optimizer must determine whether a cached or materialized view is applicable and which of the applicable views to use. To this end the optimizer must be extended in order to enumerate *read (view)* plans for applicable views just like other *access* and *join* plans and carry out cost based optimization using dynamic programming or iterative dynamic programming.

Chapter 3

Biological Data Integration Systems

"If the informatics is not handled well, the HGI [human genome initiative] could spend billions of dollars and researchers might still find it easier to obtain data by repeating experiments than by querying the database. If this happens, someone blew it."

- Frenkel, K. A.

Contents

While the previous set of techniques is sufficient for most of today's applications the advent of biology has sparked a large number of new applications and led to systems with an ever growing number of challenges.

3.1 Characteristics and challenges

The challenges that must be overcome when integrating heterogeneous bioinformatics sources are numerous.

The first challenge that must be resolved is the *variety of data*. The data exported by the available sources cover several biological and genomic research fields. Typical data that can be stored includes gene expressions, and sequences, disease characteristics, molecular structures, microarray data, protein interactions etc. Depending on how large or domain specific the sources are, they can store different types of data. Moreover, bioinformatics data can be characterized by many relationships between objects and concepts, which are difficult to identify formally, usually because they span across several research topics. Not only the quantity of data available in a source can be quite large, but also the size of each datum or record can itself be extremely large (DNA sequences, protein structures etc). This differs from non-scientific integration scenarios where there is usually no specific need to address the issue of very large entries.

Moreover, in bioinformatics, that similar data can be contained in several sources but represented in a variety of ways depending on the source. This representational heterogeneity encompasses structural, naming, semantic and content differences [Sujansky 2001]. In other words not only are they very large, but they also each have their own schema complexity. Furthermore, each source may refer to the same semantic concept or field with its own term or identifier, which can lead to a semantic discrepancy between the many sources. The opposite can also occur, as some sources may use the same term to refer to different semantic objects. Moreover the content differences involve sources that contain different data for the same semantic object, or that simply have some missing data, thus creating some possible inconsistencies between sources. This representational heterogeneity leads to issues such as entity identification across sources and data quality issues, as well as data consistency and redundancy.

Most of these sources operate autonomously, which means that they are free to modify their design and schema, remove some data without prior notification, or occasionally block access to the source for maintenance purposes. Moreover, they may not always be aware of or concerned by other sources referencing them or integration systems accessing them. This instability and unpredictability is further

affected by the simple fact that nearly all sources are web – based and are therefore dependent on network traffic and overall availability. An important consequence of the sources being autonomous is that the data is dynamic. New discoveries or experiments will continually modify the source content to reflect new hypotheses or findings. In fact the only way for an integration system to be certain that it will return the latest data is to actually access the sources at query time.

Finally, individual sources provide their own user-access interface, all of which a user must learn in order to retrieve information that is likely spread across several sources. Additionally the sources often allow for only certain types of queries to be asked, thereby protecting and preventing direct access to their data. These intentional access restrictions force end-users and external systems to adapt and limit their queries to a certain form. In [Sujansky 2001] it is noted that some potentially useful information in many cases cannot be retrieved because of query restrictions and those potentially pertinent queries cannot be asked even though the data necessary to answer them is available at the sources.

3.2 Integration Approaches

The existing systems for integrating bioinformatics sources vary along several dimensions. The integration approaches used in the existing systems can be classified first in terms of the data model they use – text, structured data or linked records. For systems that view sources as exporting mainly text, integration involves supporting keyword/text search across the sources. When the sources are viewed as exporting more structured data, there are two board types of integration approaches based on whether the data from the sources is "warehoused" or accessed on demand from the sources. For systems that view sources as exporting a linked set of browsable records, integration involves supporting effective navigation across sources. Since the majority of systems use (semi-) structured or linked record models, we will discuss the integration approaches for these in more detail.

3.2.1 Warehouse Integration

As we already discussed warehouse integration consists in materializing the data from multiple sources into a local warehouse and executing all queries on the data contained in the warehouse rather than the actual sources. Warehousing emphasizes data translation, as opposed to query translation in mediator-based integration. In fact, warehousing requires that all data loaded from the sources be converted through data mapping to a standard unique format before it is physically stored locally.

Relying less on network to access the data, obviously eliminates various problems such as network bottlenecks, low response times and occasional unavailability of sources. They allow query optimization to be performed locally [Davidson et al. 1995] and provide, the user the functionality to filter, validate, modify and annotate the data obtained from the sources [Davidson et al. 2001], [Hammer and Schneider 2003] and this has been noted as a very attractive property for bioinformatics.

This approach however has an important and costly drawback in terms of result reliability and overall system maintenance caused by the possibility of returning outdated results. As we have said, biological data usually evolve rapidly and warehouse integration must regularly check throughout the underlying sources for new or updated data and then reflect those modifications on the local copy of data.

3.2.2 Mediator Based Integration

Mediator based integration concentrates on query translation. A mediator in the information context is a system that is responsible for reformulating at runtime a query given by a user on a single mediated schema into a query on the local schema of the underlying data sources. Unlike in the warehouse approach, none of the data in a mediator-based integration system is converted to a unique format according to data translation mapping. Instead, a different mapping is required to capture the relationship between the source descriptions and the mediator, thus allowing queries on the mediator to be translated to queries on the data sources. Specifying this correspondence is a crucial step in creating a mediator, as it will influence both how

difficult the query reformulation is and how easily new sources can be added to or removed from the integration system.

The two main approaches for establishing the mapping between each source schema and the global schema are global-as-view (GAV) and local-as-view (LAV) [Florescu et al. 1998]. In the GAV approach the mediator relations are directly written in terms of the source relations. In other words, each mediator relation is nothing but a query over the data sources. The GAV approach greatly facilitates query reformulation as it simply becomes a view unfolding process; however handling the addition or removal of a source in a GAV mediator is much more difficult as it requires a modification of the mediator schema to take into account changes. In a LAV based mediator every source relation is defined over the relations and the schema of the mediator. It is therefore, up to the individual sources to provide a description of their schema in terms of the global schema, making very simple to add or remove sources but also complicating the query reformulation and processing role of the mediator. Clearly both of these approaches have some positive and negative consequences, but LAV is considered to be much more appropriate for large scale ad-hoc integration because of the low impact changes to the information sources that have on the system maintenance, while GAV is preferred when the set of sources being integrated is known and stable.

Furthermore, most systems assume that sources they are integrating, export different parts of the same "complementary" schema. In real world applications, however, we should consider the possibility that sources may be overlapping in which case aggregation of information is required as opposed to pure integration of information. Integrating complementary sources is often called *horizontal* integration whereas integrating the overlapping sources is called *vertical* integration.

Several of the bioinformatics integration systems were developed before the advent of the mediated systems, and instead follow the federated database model. A federated database integration system consists of underlying sources which are autonomous components but which also cooperate to allow controlled access to their data. In [Sheth et al 1990] it is explained that federated integration can be seen as the middle ground between no integration, where a user must query each source individually, and total integration, where a user can only query the sources through the integration system, in federated integration this schemas of the component sources

are put together to form an integrated schema on which queries will be asked. Seen from this vantage point, mediated systems could be seen as very loosely coupled versions of federated systems.

3.2.3 Navigational Integration

The idea of navigational or link-based integration emerged from the fact that an increasing number of sources on the web require the users to manually browse through several web pages and data sources in order to obtain the desired information [Davidson et al. 1995]. In fact the major premise and motive justifying this type of integration is that some sources provide the users with pages that would not or hardly be accessible without point-and-click navigation. The specific paths essentially constitute workflows in which the output of a source tool is redirected to the input of the next source until the requested information is reached [Buttler et al. 2002]. In effect queries are transformed into path expressions that could reach each answer the query with different levels of satisfaction [Mork P. et al 2001]

Pure navigational integration eliminates relational modeling of the data, and instead applies a model where sources are defined as sets of pages with their interconnections and specific entry-points, as well as additional information such as content, path constraints, and optional or mandatory input parameters. In [Friedman et al. 1999] is claimed that this model effectively allows the representation of cases where the page containing the desired information, is only reachable through a particular navigation path across other pages.

3.3 Existing Bioinformatic Integration Systems

This section covers a description of some well – known systems that are currently available in the domain of bioinformatics.

3.3.1 SRS

The Sequence Retrieval System is closer to a keyword – based retrieval system than an integration system. Its approach to Bioinformatic integration is to parse flat files or databanks that contains structured text with field names. It then creates and stores an index for each field and uses the local indexes at query time to retrieve relevant entries [Lopez 2001]. Although extensive indexed entries are kept locally to be used by the query processor at query time, SRS is not actually a warehouse system as the actual data is neither modified nor stored locally. The main feature of SRS is that it keeps track of the cross-references between sources. In order to parse the flat files, the system has its own parser which is called ICARUS and it is designed to recognize the presence of links and index all source records using a keyword-based indexing approach. Therefore, while parsing, the system can identify links that exist between entries in different sources. These links are then used to suggest more results to a user after a query has been processed.

The user query interface is straightforward in SRS. A user first selects which of the many available sources should be queried, depending on the type of data expected, and then asks a keyword or gene sequence query on those sources. After the query is processed, the relevant document in terms of the query keywords is displayed. Additionally, SRS will search in its local index of parsed links for entries that are related in some way to the query. All such links are then made available to the user and grouped by source or by the type of data they point to. In other words, the results of the query in this system are essentially composed of a set of tuples or entries directly retrieved from the initially selected sources and a set of paths across other sources which lead to information that is related to the query.

3.3.2 K2/BioKleisli

BioKleisli is primarily a loosely – coupled federated database system. The mediator on top of the underlying sources relies mainly on a high-level query language that is more expressive than SQL and that provides the ability to query across several sources (it is called Collection Programming Language or CPL). CPL [Davidson et al. 2001] requires source specific wrappers to map sub-queries to specific heterogeneous sources, which are accessed through predefined atomic query-

functions. The data model used is an object-oriented type system that is more expressive than the relational model since it includes bags, lists, variants, nested records etc.

BioKleisli does not use any global molecular biology schema or ontology that the user could use to formulate queries. This approach therefore requires that the users posses an expertise in CPL and a perfect knowledge of the underlying data schemata. This project was mainly aimed at performing horizontal integration and in fact a query attribute is usually bound to an attribute in a single predetermined source. There is essentially no integration of sources with content overlap and as a consequence no optimization based on source characteristics or source content is performed. In fact the procedural nature of CPL makes the query optimization task really difficult. In the newer version K2 of the system, CPL is abandoned and OQL is used, but the overall flow of the system is not modified.

3.3.3 TAMBIS

Transparent Access to Multiple Bioinformatics Information Sources or TAMBIS [Baker et al 1998], [Paton et al 1999] is a mediator-based and ontology driven information system. Queries are formulated through a graphical interface where a user needs to browse through concepts defined in a global schema and select the ones that are of interest for the particular query. Then the system expresses the graphical query in GRAIL, declarative source independent description logic and after that the query is translated into a Query Internal Form (QIF), which is in turn translated into a source dependent query execution plan in CPL. Because TAMBIS needs external wrappers, it uses wrappers from BioKleisli system to access the underlying sources.

The planning and optimization subsystem in TAMBIS only performs reordering of query components. It does not store source statistics or analyze source capabilities. Reordering is based on the cost of individual query components, where the cost combines the predicted time necessary to evaluate a component as well as the expected number of results it will return. This optimization therefore does not include any evaluation of sources in terms of content overlap or source availability. In fact, a given concept and its CPL function are always linked to a predetermined source,

which means that even if several sources contain information about a concept, one of them will always be addressed for that particular concept. Moreover, it must be noted that the ontology defined by TAMBIS is not primarily used for schema mapping between the underlying bioinformatic sources. Instead it is a dictionary and a classification of biological concepts that represents subsumption relationships between concepts. The mapping of ontology concepts to source dependent CPL functions is done by another subsystem called the Source Model. Hence the TAMBIS domain ontology mainly serves the purpose of easing the user's task of formulating queries.

3.3.4 DiscoveryLink

DiscoveryLink [Hass et al. 2000], [Hass et al. 2001] was IBM's proposal on the area of bioinformatics. It is a wrapper-oriented system and it serves as an intermediary for applications that need to access data from several biological sources. It is an integration layer built on the Garlic project technology and it serves as a middleware between the applications and a set of wrappers. The source specific wrappers must register their data source in order to be integrated.

Users connect to DiscoveryLink and issue queries in SQL based on some global schema. Garlic technology is mainly a federated database query processor that communicates with source-specific wrappers to determine optimal plan for a given query and executes the query over possibly several sources. The data model used is the object-relational model and the wrappers provide source-specific information about query capabilities that help the optimizer to determine which parts of a query can be submitted to each source.

Using the information provided by wrappers, the query is broken into portions that can be handled by different sources. Then each wrapper produces a plan that the underlying source is capable of executing, and evaluates the execution cost of that plan. The overall cost of all plans is calculated by the optimizer where several factors are taken into account such as the local execution cost, network cost, selectivity, and the cost of any remaining operations that cannot be performed by the data sources. After the wrappers have produced their plans and the optimizer have decided on the best plan to adopt, the execution engine will send out individual plans to be executed

by wrappers. Once the wrappers have performed their plan, the processed data flows from the data sources into DiscoveryLink engine, which in turn performs any operations that could not be handled by the sources, and returns the data to the client application.

Unlike TAMBIS, DiscoveryLink is not a user-end product. A user interface is required to operate on top of DiscoveryLink to elicit queries that are processed and sent to the underlying sources.

3.3.5 BACIIS

Biological and Chemical Information Integration System [Ben Miled et al. 2003] is an on-demand information integration system for life science web databases. It was developed using the mediator-based approach combined with extensive use of a knowledge base. The knowledge base contains a domain ontology which serves as global schema for the system and which captures object classes, attributes, and multiple complex relationships between them. The knowledge base also keeps the data source schema which maps the schema of individual sources to the domain ontology. One of the goals of this project is also to derive extraction rules automatically and store them in the source wrappers. The whole architecture consists of five servers that cooperate to answer multi-database queries over a set of geographically distributed life science databases. These servers can be executed on the same machine or in different machines, which maximize resources utilization and reduces the effort needed to add new services. The user formulates queries interactively within forms and the sources that need to be queried are automatically selected by the system while the data model used here is structured, object – relational.

3.3.6 Other Systems and the ideal system

Except from those systems several others exist. GUS [Davidson et al. 1995], is a system that follows the approach of data warehousing and allows users to add annotations that may want to associate to some retrieved data. KIND [Gupta et al.

2000], [Ludascher et al. 2001] attempts to combine the use of formal ontologies and conceptual models with source-specific wrappers and ENTREZ is a web-based link-driven federation in which sources are interconnected so that any entry returned from one of the integrated sources will also have related links to the other sources.

There is no such system that we could describe as the best one. The question here is what the biologists and other researchers want from a system. The primary use of such systems is to enable scientists to acquire some knowledge from large amounts of data, to then formulate hypotheses from the knowledge acquired and finally perhaps to validate these hypotheses. The amount of work necessary without an integration system is prohibitive, which is why the main goal of these systems should be to automate a maximum number of tasks. It is clear that it is up to the system to ensure that users will find what they are looking for in a minimum amount of time and interactions. In many cases users may do not want a fully transparent query layer because they might want to choose which sources is to be accessed and by what plan (i.e. in TAMBIS). This tends to show that the system must be able to provide enough flexibility to the user as well as display the provenance of the data.

Moreover, it is desirable that source representation and source capabilities be automatically extracted. As of today, most source descriptions are obtained through a manual analysis of the source schema or interface by both a domain expert and an integration expert, which are usually two distinct people. Automating the process will reduce the cost and time necessary to develop full-scale integration systems that can keep up with the pace at which biological data is generated. Furthermore, it is important for an integration system to gather source statistics in order to refine the query plans and improve the overall functionality and performance of the system even as the sources evolve. Except from that we must take into account the interesting fact that most biologists or researchers value data even though it may be only partially complete and potentially incorrect. Any data can indeed be relevant to a scientific researcher.

Much like TAMBIS and K2, most of the currently widely used integration systems only address the horizontal dimension of data integration. In integrating only sources that have complementary data, an integration system does not take into account the potential overlapping aspect of sources or the probable incompleteness of some of them. Restricting the integration process to simply combining data from

sources that contain different types of information for the same semantic entity, limits the capability of a system, especially in terms of reliability and completeness. A purely horizontal integration system cannot address issues of effectiveness and efficiency. In fact, aggregation of information of sources is also necessary.

Chapter 4

Quete: A System For Data Integration

"An expert is a man who has made all the mistakes which can be made, in a narrow field."

- Niels Borh

Contents

4.1 Introduction

In previous chapter, we illustrated that there is no system considered to be complete in the area of bioinformatics. The brief discussion justified the need for systems that provide an integration of bioinformatic sources as there exists a real demand from biological researchers who are now overwhelmed by the amount of work necessary to manually go through the integration process. After a short description of the major systems used by biologists we pointed out the lack of aggregation systems, which could integrate sources containing semantically similar data, also known as *vertical* integration. Existing methods either require the user to

know the semantics of all data sources or they impose a static global view that is not tolerant of schema evolution. These assumptions are not valid in many environments.

Moreover we illustrated that when designing a heterogeneous database, the goal is to encapsulate the heterogeneity of the component databases and to use existing homogenous distributed database techniques as much as possible.

Having all that in mind and knowing that the ideal integration system should truly take into consideration the wishes of those who will use the system we built QueTe. QueTe was based on Unity [Mason T. and Lawrence R. 2005] which was extended and enhanced in order to produce a system capable of integrating bioinformatic sources. This work proposes an automatic schema integration algorithm which removes all naming conflicts by utilizing a standard ontology to describe schema element semantics.

4.2 The Initial Idea

In this thesis we propose a method for semi-automatic schema integration by using a standard ontology to describe schema element semantics. The use of ontology resolves naming problems, which allows our algorithm to automatically resolve the more complex structural and semantic conflicts. The major contribution of this work is a systemized method for capturing data semantics using a reference ontology and a model which uses this information to perform schema integration in relational databases.

The starting idea was to build a system that would integrate several databases. Those databases would be autonomous and independent and would evolve at will. Several kinds of databases have been studied but we eventually focused on relational ones, as they are most used today. All those databases, that could store data from several areas, would have a schema that describes how data are organized and stored. In many cases, different sources may want to share only a portion of their data so it was crucial to have the ability to decide which fragments of data were going to be shared.

After deciding which parts of the local schema each database would share, those schemata would be integrated to build a Global Schema. The Global schema

would be used internally by the system and mappings between the Global schema and the local schemata would exist to determine data allocation.

Moreover, users needed a common starting point in order to understand the information stored in the integration system and how to query it. That common starting point is an ontology that is defined at the top level of the system and users can use ontological terms to query underlying data sources. Of course a set of mappings is needed between the ontology and the Global Schema in order to answer queries transparently.

That basic idea is shown at the figure below. Having that idea as a staring point we extended our implementation further, and we are going to examine it, in the rest of this chapter.

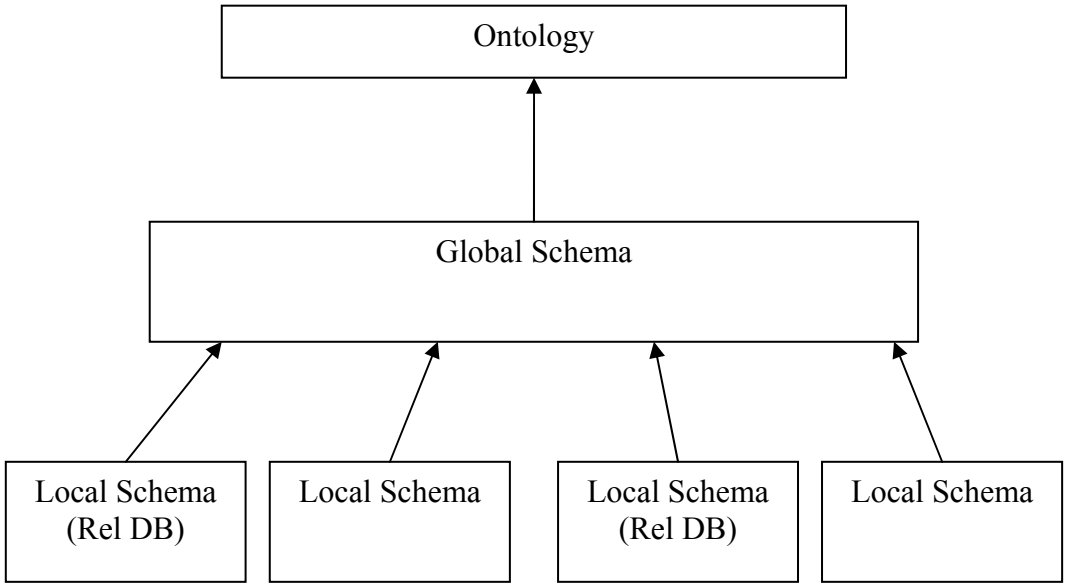


Figure 8. Integration Schema

4.3 The Integration Architecture

Before going further we should describe the architecture of our system. The integration architecture consists of two separate and distinct phases: The *capture* process and the *integration* process.

The fist phase is used to capture the data to be integrated. This process is performed independently in each data source and the only "binding" between individual capture processes, at different data sources, is the use of the common ontology to provide standardized terms for referencing data. Here, the database schema to be integrated is being extracted and the metadata are stored in a specific XML file called X-Spec which we are going to describe later in this chapter. Those metadata extracted are being annotated using ontology terms, and that semantic information is stored in X-Spec too.

The *integration* process actually performs the integration of the various data sources. It is assumed that there is a central site where the integration is performed by combining the X-Specs of the data sources. Clients wishing to access the individual data sources submit their queries to this central site which handles the necessary mappings and transaction management.

The key benefit of these two phases is that the capture process is isolated from the integration process. This allows multiple capture processes to be performed concurrently and without knowledge of each other. Thus, the capture process at one data source is not affected by the capture process at any other data source. This allows the first phase to be performed only once regardless how many data sources may actually be integrated. Moreover each data source is able to change the semantics, the schema and the portion of the data to be shared by just altering the X-Spec file that they provide. These are significant advantages as they allow application vendors and database designers to capture the semantics of their systems at design-time or at any other time they want, and the clients of their products are able to integrate them with other systems with minimum effort.

The central site takes the X-Specs of the individual data sources and executes the integration algorithm to produce an integrated view (i.e Context View) that will be used internally. Users then can issue queries on the central site using an SQL like query language that is going to be described in the following chapter. When a query is sent to the central site, the necessary mapping from ontology to system names is performed and the query is divided into several subqueries against the data sources. The central site is assumed to implement the functionality of a DBMS manager which includes transaction management and query processing. Once results are returned

from the individual data sources they are integrated based on the unified view and then returned to the user.

It is important to note that by the use of a central site and relational underlying databases, no translational or wrapper software is required at individual data sources. Once the X-Spec has been provided for the data source and integrated by the central site, the software at the central site communicates directly with the data sources using ODBC or proprietary protocols. All translation, integration and global transaction management is handled by software at the central site.

This approach allows full autonomy of the underlying data sources as the central site appears as another client issuing queries to them. Moreover this approach allows the development of standard ontologies that could be used across industries, organizations and the scientific community. Those ontologies do not need to be complete or widely accepted. Application specific ontologies can as well be used without any semantic loss.

4.4 Integration Components

After briefly describing the integration architecture it is necessary to explain the three basic components: The standardized ontology, the metadata specification for capturing data semantics, and an integration algorithm for combining metadata specifications into an integrated view.

The ontology provides a set of terms for constructing semantic names describing schema elements. By defining semantic names using a standardized ontology we resolve naming conflicts since two schema elements with the same semantic name are assumed to represent identical concepts regardless of their structural organization. Metadata specifications, called X-Spec, store schema information in XML documents. An X-Spec contains also mappings from semantic names to system names used in the data sources. The integration algorithm matches the semantic names to produce an integrated view of concepts.

4.4.1 The Reference Ontology

People, in order to exchange knowledge, use a common language to describe knowledge. Knowledge transfer in conversation arises from the definitions of the words used and the structure in which they are represented. Since a computer has no built-in mechanism to associate semantics to words and symbols, a common point of reference is required to allow the computer to determine semantically equivalent expressions.

Determining semantically equivalent words and phrases is a complex problem. The English language is very large with many equivalent words for specifying equivalent concepts. Thus, the size of the database is a problem, and it is complicated for the computer to determine in which cases two words represent semantically equivalent data.

Ontologies are a common point of reference and they have been used in various roles for database integration [Batini et al 1986], [Sheth et al 1990]. Most organizations such as the National Cancer Institute or the National Institutes of Health have been developing standard ontologies for their domains that could be useful in the process of integrating several data sources. The idea is to match each source to the domain ontology, and each schema-to-ontology map is validated by the administrator. The advantage of this approach is that the administrator only needs to understand the semantics of their schema when validating matches. Schema-to-ontology mappings can be used to build mappings to any schema that is also matched to the ontology by composing the schema-to-ontology matching.

The ontology in our system is organized as a graph of concepts. All concepts are placed into a graph and are related using two types of relationships. ‘IS-A’ relationships and ‘HAS-A’ relationships. ‘IS-A’ relationships are the standard subclass and superclass type of relationships and are used to model generalization or specialization data concepts. Component relationships relate terms using ‘Part-of’ or ‘HAS-A’ relationship. For example, an address may have city, state, postal code, etc. Similarly, a person’s name may have first and last name components. To represent ontologies like these, we could use RDFS. Although this is a rather simple modeling mechanism it is adequate for modeling the real-world. In case that our ontology uses more complex relations, they can be rewritten by using only ‘IS-A’ and ‘HAS-A’ relationships. We believe that although it is a trivial task, it may be time consuming

for complex ontologies. We must note however that the ontology is not the integrated view. It is just a standard set of terms to consult in order to describe semantics for creating the integrated view and the ontology provides standardized names for concepts with unambiguous definitions.

Initially the ontology may contain a limited set of concepts commonly stored in databases. We can assume that it is possible for the ontology to expand over time as new types of data appear and the underlying databases evolve. Thus, we allow an organization to add nodes to the ontology to both the concept hierarchy and component relationships to capture and standardize names used in their organization which are not in the standardized ontology. These additional links are stored and transmitted along with the metadata information during integration. We expect that the evolution of the ontology would be directed by some standardization organization to insure that new concepts are integrated properly over time.

It is important to realize that the exact terms and the organization of the ontology are irrelevant. Although this may seem surprising, consider that language is simply a standard for expressing semantics. There is no fundamental reason why the word "table" should describe a table. Similarly, the exact organization of the concept hierarchy and the terms used to represent concepts is irrelevant as long as they are agreed upon. However the goal is to produce something readable by humans, so the terms should be recognized English words for their concepts, and the base hierarchy should be evolved in a way that models current standardization efforts and real-word organizations. Any standardized ontology can be used as long as it is formatted correctly and has the necessary terms to capture the semantics of every data element to be integrated in the corresponding data sources.

The definition of a semantic name for a given schema element is not a straight-forward mapping to a single ontology term. A *semantic name* captures the system-independent semantics of a schema element including contextual information by combining one or more ontology terms. A semantic name has the form

semantic_name = "[" OT [[; OT] | [, OT]] "]" * [ON]

OT = < ontology_term > , ON = < ontology_term >

That is, a semantic name consists of an ordered set of context terms (OT) separated by either a comma or a semi-colon, and an optional concept name term (ON). The comma between terms A and B (A, B) represents that term B is a

subtype of term A. A semi-colon between terms A and B (A; B) means that term A HAS-A term B, or term B represents a concept that is part of term A. The context terms provide a context framework for the concepts that describe them. Every semantic name has at least one context term. The concept name is a single, atomic term describing the lowest level semantics. Fields have concept names to represent their base meaning void of any context information.

Abstractly, a semantic name is a hierarchy of concepts related by IS-A and HAS-A relationships. Typically in relational databases all terms in a semantic name are related by HAS-A associations. For example consider the table Books (ISBN, Title, Author, Publisher, Price). Their semantic names in a really simple ontology are shown in the following table.

Type	Semantic Name	System Name
Table	[Book]	Book
Field	[Book] ISBN	ISBN
Field	[Book] Title	Title
Field	[Book] Price	Price
Field	[Book; Author] Name	Author
Field	[Book; Publisher] Name	Publisher

Table 1. Books Database schema

4.4.2 X-Spec – Metadata Specification

The definition of a standardized ontology by itself is not enough to achieve integration because the ontology is not defining a standard schema for communication. It simply defines terms used to represent concepts. These concepts can be represented in vastly different ways, in various data sources, and we are not assuming a standardized representation and organization for a given concept. Thus, a system for describing the schema of a data source using ontology terms and additional metadata must be defined. Our integration approach uses a structure called X-Spec to store semantic metadata on a data source. The X-Spec is essentially a database schema encoded in XML format and is organized in relational form with tables and fields as basic elements.

An X-Spec consists of the relational database schema being described along with additional information about keys, relationships, and field semantics. More

importantly, each table and field in the X-Spec has an associated semantic name built from terms in the standardized ontology as previously discussed.

The use of XML for describing an X-Spec is not required, but it is used because XML is an emerging standard to exchange semantics between systems. However, the definition and usefulness of an X-Spec is not tied to XML. Information stored in XML in an X-Spec can just be transmitted as a formatted text files or a structured binary file. XML is used for convenience and interoperability with emerging standards on semantic exchange.

In order to ease the capture process of sources metadata, a tool (i.e. Extractor) has been developed that can read each database schema, and produce the X-Spec corresponding to the whole information needed. Key, foreign keys and constraints are captured automatically and the administrator has only to relate system names with ontology terms.

4.4.3 Integration Algorithm

The integration algorithm is a straightforward matching algorithm of terms. The same term used in two different X-Specs is assumed to represent an identical concept regardless of its representation. The algorithm receives as input one or more X-Specs describing data schemata and then it uses the semantic names present in them to match related concepts and to build a global view (as we can see in the next chapter this global view is named Context View and has many interesting properties).

For example consider that the database schema shown in table 1 is annotated and semantic names are given in tables and fields. When our algorithm starts, the first semantic name that is being processed is *[Book]*. This semantic name consists of only one term which does not match any other term in this depth. So, it is added to the tree under the root. The next semantic name to be processed is *[Book] ISBN* with two terms. The first term already exists in Global View and is matched. According to our algorithm, we go one level below the current term in *V* (i.e *Book*) and then we proceed to the next term *ISBN* which is not matched at this level and it is added below the *Book* term. The algorithm goes on the same way until *[Book; Author] Name* is met. The *Book* term is matched so we go one level down and we search under the *Book* term to find the *Author* term exists. Since *Author* does not exist the remaining terms

of the semantic name are placed under the *Book* term. Moreover, since the term *Name* is after the term *Author* in the semantic name it is placed under the *Author* term. The algorithm will continue the same way if another table, Authors for example, exists. After processing every semantic name, the final global view constructed is shown in figure 10.

```
Input: One or more X-specs
Output: Global View V
1:      For each X-Spec X {
2:              For each semantic_name SN in a X {
3:                      Go to top level in V
4:                      For each term T of SN {
5:                              If T does not match any term at this level Then
6:                                      Add this and all remaining terms of SN to V
7:                                      in the proper levels
8:                                      Break
9:                              Else
10:                                     Current term= matching term in V
11:                                     Go one level below current term in V
12:
13:                     }
14:             }
15:     }
16:     return V
```

Figure 9. Integration Algorithm

The architecture identifies similar concepts by name regardless of their physical or logical representations in the individual data sources. The integration result is a hierarchy of contexts and concepts which implies no particular physical representation. The physical representation of the concepts is irrelevant to the user. The user accesses data sources through semantic names which map to physical schema elements. Thus, by not imposing structural constraints or concept representation, knowledge from systems is combined regardless of data representation characteristics, and the user is provided with only the relevant information.

The integration is valid because it combines correctly database schema into an integrated view given the assumption of no naming conflicts. The architecture avoids naming conflicts by developing and using a standard list of terms referenced in our ontology and combining them appropriately into context and concept information to

express schema element matches. Since the semantic names constructed are assumed to represent the same concept if their name matches, integration of concepts across schema is possible simply by matching semantic names. Concepts are integrated across data sources solely by name regardless of their implementation or physical structure. Of course, we keep in memory the corresponding fields for each semantic name.

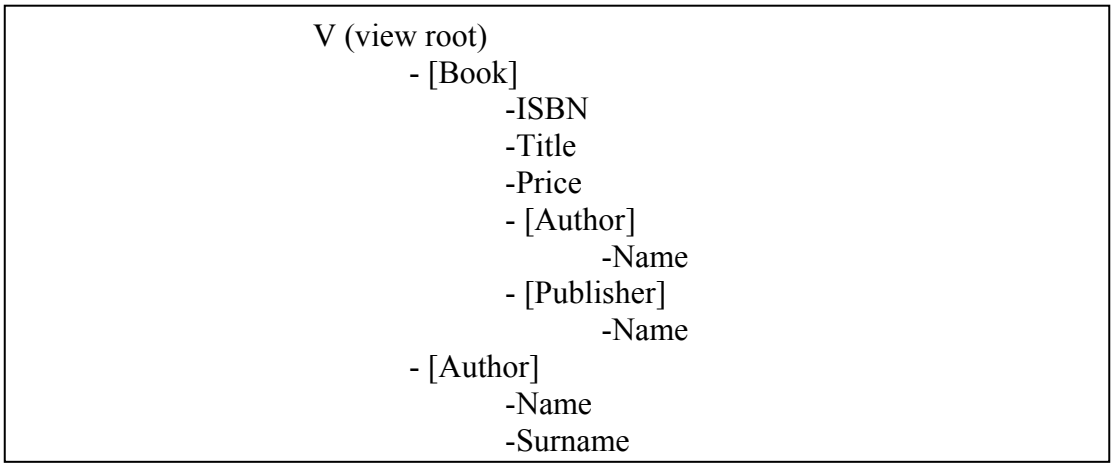


Figure 10. Building Integration Schema (Context View)

4.4.4 Querying in QueTe

After building the integrating view in memory, the user is given the capability to issue queries. The query language is an attribute-only version of SQL, where the SELECT clause contains the concepts to be projected in the final results and the optional WHERE specifies selection criteria for the query. An example query that gets the price of the book “A Semantic Web Primer” could be

SELECT [Book] Price WHERE [Book] Title = “A Semantic Web Primer”

Notice that the FROM clause is absent since the integration system will automatically identify the tables to be used. Of course, the user must express the queried terms by describing them using their semantic name that is being built according to the ontology. Then the semantic names are matched against the global view and the query is answered. The required joins between the tables are automatically inserted by the query processor. The order in which X-Specs are

integrated is irrelevant, and the same X-Specs can be integrated several times with no change.

Using the reference ontology, users can formulate queries as the previous one by choosing which terms they want to view through their semantic names according to the ontology. These semantic names map to physical fields and tables in the underlying data sources. The user is not responsible for determining joins between physical tables in a given data source or across a data source nor where each table is placed. The system handles the necessary joins based on the relationships between the schema elements. The query implementation is similar to MIX [Baru C. et al. 1999] except that the query is formulated on an integrated view based on the ontology instead of the mediated views.

In many cases there is a straightforward mapping from semantic names to physical fields. Typically, a semantic name will have only one mapping to a physical field in each data source. Given a list of semantic names in the query used either for projection or for selection criteria, the query processor maps the semantic names to system names using the information stored in the X-Spec. To handle joins between tables, X-Specs stores information on join conditions between tables in order to be used by the query processor. Thus, all required mapping information is present to construct a select-project-join query which then is translated into several subqueries that are sent to the individual sources. When subqueries are answered the results are being integrated and then presented to the final user. Joins are selected by the system from X-Spec information and if no join condition exists between tables, a cross-product is used as real databases do. “Global keys” are important in query generation, as they guarantee unique values across databases similar to social security number which identifies distinct human beings. Such keys allow the system to perform joins across databases. So when a query is divided into several subqueries that involve some global keys, the results returned, are joined or unioned using appropriate global keys and then the outcome is presented to the final user.

As we have already said, in many cases most biologists or researchers value data even though it may be only partially complete and potentially incorrect. Any data can indeed be relevant to a scientific researcher. That’s why we designed the system to show even tuples when the data source does not have all the fields required in the result. In such cases, the fields missing are left blank in the returned result.

Chapter 5

Multidatabase Querying in Quete

"I only ask for Information"

-Charles Dickens

Contents

5.1 Introduction

Despite dramatic changes in database size, complexity and interoperability, SQL has remained fundamentally unchanged. The wide variety of applications, users and implementation systems accessing databases rely on the Structured Query Language (SQL) [Date C. J., 1994] to retrieve the required information. Although the complexity of SQL generation has been partially hidden by graphical design tools and more powerful programming languages, the fundamental challenges of SQL remain.

The fundamental problem of SQL is also one of its greatest advantages. SQL allows a database to be queried by a clearly defined structure which is a vast

improvement over hierarchical methods and direct access technologies that require explicit navigation between records. Unfortunately an SQL user is responsible for understanding the structure of the database schema, the names associated with semantic names and the relations between them. Query formulation involves mapping query semantics into the semantics of the database and then realizing those semantics by combining the appropriate structures.

SQL is a powerful language when used by people who understand its semantics and the database queried. However, nowadays the need to interact with multiple database systems with little and limited database understanding is emerging. Moreover, organizations are attempting to achieve database interoperability by combining database systems into a more unified organization. Those systems force users to understand the structure and semantics of all databases which introduces exponential complexity as the number of databases increases.

To address those shortcomings, our architecture automatically integrates diverse relational schemata into a unified view of concepts, called context view and those concepts come from the reference ontology. The context view is a special type of Universal Relation describing the data source and has features that resolve some of its problems. Although the context view and its associated query system were not developed to model the Universal Relation, they display many similar properties which can be used to better understand the foundations of the context view and may be used to develop similar query algorithms.

5.2 Previous Languages Used

Before we go further in describing our language and the query mechanisms it is useful to briefly describe the languages developed and used in previous multidatabase and federated environments.

In order to achieve multidatabase querying, several languages were developed like MSQL [Krishnamurthy R. et al. 1991] and its successor IDL [Litwin W. and Abdellatif A, 1987]. Those languages allow the user to define higher order queries and views by providing database variables that can range over metadata in addition to regular data. Metadata include database names, relational names, and attribute names.

The language allows queries across database systems in addition to regular expressions. Other MDBS query languages include DIRECT [Merz U. and King R. 1994] and SchemaSQL [Gingras F. et al. 1997]. The fundamental weakness in multidatabase query languages is the reliance on the user's knowledge of the database structure and semantics to construct queries. Further, data organization is optimized for efficiency and not understanding. Understanding the structure and semantics of one data source is complicated in itself, and the in-depth knowledge required to formulate queries on multiple databases is extremely rare. Although, previous languages may allow the construction of multidatabase queries, they do nothing to reduce the need of the user to thoroughly understand the semantics.

Several other languages have been developed that allow users to query by word phrases in order to simplify querying [Cohen W. 1998], [Konopnicki and Shmueli 1998], [Ogden and Brooks 1983]. These systems are not powerful enough for a general multidatabase environment because they do not allow the user to precisely define the exact data returned. Word systems that simplify query formulation by ignoring structure sacrifice query precision.

Other systems try to augment a relational database with logical rules or knowledge [Kuhn E. et al. 1994], [Motro A. 1990] or change or add to the database in some manner. This is done in order to enable advanced queries to be posed, but that violates database autonomy and thus it is not desirable.

A query system must isolate the user from structure and system details while at the same time should provide a query language powerful enough to produce precise, formatted results. SemQL [Lee J.O et al. 1999] attempts semantic querying using semantic networks and synonym sets from WordNet [Miller G.A et al. 1990]. Although their approach is similar to ours, using a large online dictionary such as WordNet in querying time, increases the complexity of matching word semantics. Our approach improves on SemQL by providing condensed term ontology, an integrated view to convey database semantics to the user, and a systematic method for SQL generation.

A fundamental database model is the Universal Relation Model which provides logical and physical query transparency by modeling an entire database as a single relation. Just as the relational model relieves users of the responsibility for navigation within the physical database, a universal relation system relieves them of

the responsibility for navigation among the relations. We will demonstrate the similarity of our context view with the Universal Relation Model [Maier et al. 1984], and thus argue that our system provides logical and physical transparency. There has been substantial work presented on querying a universal relation environment [Bressan et al. 1988] and more generally in theory of joins [Aho et al 1979] and querying [Korth et al. 1984], [Sagiv Y. 1983].

It is important to note that our architecture extends wrapper and mediator systems. Simple mediator systems either assume that an integrated view of data sources is constructed a priori by designers or do not construct an integrated view. If an integration view is constructed, it is a conventional, structural organization of the data into relations and attributes. This integrated view is then mapped to the local views of the mediators by logical rules or query expressions specified by the designer. Mediators do not perform schema integration. Schema integration or the actual construction of the integrated view is manually performed by designers. In our system although, an integrated view is automatically produced from data source specifications developed independently of other data sources and the global view itself.

5.3 Context View as a Universal Relation

The context view (CV) produced by the integration architecture models database schema knowledge as a hierarchy of contexts and concepts. In this section, we more formally describe the nature of the CV and its relationship to the Universal Relation. Firstly, it is necessary to define the concepts of a standardized ontology term, a semantic name and the context view.

An Ontology term is a single, unambiguous word or word phrase present in the standardized Ontology. Each term represents a unique semantic connotation of a given word phrase, so words with multiple definitions are represented as multiple terms in the Ontology. A context term is an ontology term used in a semantic name which describes the context of schema element associated with the semantic name. A concept term is a single ontology term used in a semantic name which provides the lowest level semantic description of a database field. Basically, a concept is a

semantic name which maps to a database field whereas a context is a semantic name which maps to a database table. For example, the semantic name *[Category] Id* is a concept because it maps to the database field *CategoryID*. The semantic name [*Category]* is a context because it maps to the database table *Categories*.

As we defined in the previous chapter, a semantic name S_i consists of an ordered set of ontology terms $T= T_1,T_2, \ldots,T_N,$ where $N >=1$, which uniquely describes the semantic connotation of a schema element. If $N=1$, then T_1 is a context term. The last term T_N is a concept name if S_i has a concept name; otherwise it is the most specific context of S_i. A semantic name is a hierarchy of contexts each of which has a meaning independent of the semantic name. When integrating semantic names into a context view it is necessary to match semantic names based on their associated terms. For this purpose it is useful to define the *context closure* of a semantic name:

Definition: The context closure of a semantic name S_i denoted S_i^*, is the set of semantic names produced by extracting and combining consecutive ordered subsets of the set of terms $T=T_1, T_2, \ldots, T_N$ of S_i starting from T_1.

For example, given a semantic name S_i = [A; B; C] D then S_i^*= {[A], [A; B], [A; B; C], [A; B; C] D}. Based on the above we can define a *Context View* as follows:

- If a semantic name S_i is in CV, then for any S_j in S_i^*, S_j is also in CV.
- For each semantic name S_i in CV which ends in a leaf node, there exists a set of one or more mappings M_i which associate a schema element (table field) E_j with S_i.
- A semantic name S_i can only occur in the CV once.

That is, for every semantic name that exists in the context view, all its associated semantic names formed by taking a consecutive subset of its terms are also in the context view. Moreover, each semantic name in the view can be mapped to physical fields and tables by the set of mappings provided by the system. The integration architecture combines schema elements into the context view by merging their associated semantic names with the semantic names currently present in the CV. Matching proceeds term-wisely until a complete match is found, or no further matches are found as we saw in the integration algorithm, described in the subsection 4.4.3.

Thus the CV is a graph of nodes N= N_1, N_2, …, N_n where each node N_i has full semantic name Si consisting of one or more ontology terms T_1, T_2, …, T_m. When a node is added, each of its corresponding terms are recursively added starting at the root.

There is an underlying similarity between a Context View and Universal Relation (UR). A Universal Relation contains all the attributes of the database where each attribute has a unique name and semantic connotation. The fundamental feature of UR is that all attributes are uniquely named with a unique connotation.

Lemma: A context view is a valid Universal Relation if each semantic name is considered an attribute.

Proof: In order to violate the Universal Relation assumption, a given semantic name must either occur more than once in the CV (non-unique attribute names) or two or more semantic names have identical connotations (non-unique semantic connotations). By definition of CV, each semantic name can occur only once. Hence each semantic name is unique. Moreover, the construction of a semantic name by combining terms defines its semantics such that two different semantic names cannot have the same semantic connotation. Thus, a context view is a valid Universal Relation**.**

Although, a given semantic name occurs only once in a context view, it is possible that there is more than one mapping to physical fields in a single data source. Consider for example two tables *Orders* and *OrderDetails* and one field called *OrderId* in both tables. That field is assigned the same semantic name in both tables (e.g [Order] Id) and this makes sense because each of these two fields has the same semantic connotation and is only represented in two different tables due to the normalization of the tables. When those two tables are combined into a UR, only one instance is retained. However, the query system must decide on the correct and more efficient mapping when generating query access plans.

A context view examined as a Universal Relation addresses several of the problems that have been studied for the UR model. First, the context view is automatically generated by the system combining the semantics of each database that administrators provide. The system uses the supplied semantics, schema and join information and automatically builds the context view. This process can be applied in

reverse to extract query results from normalized database tables given a query expressed on the context view according to ontology terms.

Furthermore, the context view resolves the issues of large and complex Universal Relations. Since the context view is organized hierarchically by context, there is an explicit division of the context view into semantically grouped topics as opposed to one, flat relation containing all attributes. Unlike a strict Universal relation implementation, the context view is never physically constructed. Rather, like a view, it is an outlook of the data stored in other structures which is built as needed. Thus, the focus of the rest of this chapter is demonstrating how queries posed through the context view can be physically realized by an automatic algorithm which maps from semantics to structure and produces relational calculus (SQL) expressions on the underlying data sources to extract the relevant data.

5.4 Query Parsing and Join Tree Construction

By isolating the user from database structure, the system becomes responsible for correctly formatting the query based on the user's intended semantics. The most important property the query system must provide is consistency, which means that the system must generate deterministic, repeatable, and semantically intuitive queries in all cases.

Given an Ontology, users can generate queries which contain a subset of context view's concepts. Since a query is just a subset of the context view, the query can be examined similar to a context view. There are two major requirements in mapping from semantic to structural querying. First, the system must select the appropriate fields to use for projection and selection, since multiple mappings to the same semantic name are possible within a given data source. The query result may be different for different mappings to the same semantic name because new joins may be introduced if the field is in another table. Second, the join conditions must be automatically be determined to combine the appropriate data source tables.

Regardless, if the field is being used in a selection or projection operation, all fields are treated uniformly by the query system. Determining the correct field instance to select if a given semantic name can be mapped to multiple fields in the

underlying database is complex. Fortunately, it is unlikely that a semantic name has multiple field mappings when the database is normalized if the field is not a key field. However, the choice of a key field with multiple mappings is especially important as it affects the join semantics. Depending on the field mapping chosen, different tables are joined together. For example as we noted previously the semantic name *[Order] Id* may map to two physical fields, *OrderId* in the *Orders* table and *OrderId* in the *OrderDetails* table. In both cases, the field has the same semantics. However depending on which of the two mappings is selected, a new join may be introduced into the query if the table is not currently in the query.

For a key field occurring in more than one database table, there are four cases to consider based on the interrelationships between the parent tables for field mappings. That is, if the key field is present in two or more tables, the inherent interrelationships between these tables determine the complexity in selecting the correct mapping. These cases are:

1-1: An one-to-one relationship between tables normally implies that the tables share some key. The mapping chosen in this case is uniquely determined by the user's choice of semantic name (*[Person] SSN* and *[Employee] SSN* determines that in the first case SSN will be selected from Person table, whereas in the second case from the *Employee* table.)

1-N: An one-to-many relationship between tables implies a foreign key from the N-side table to the one-side table. Consider the tables *Orders* and *OrderDetails*, where a record in *OrderDetails* table which contains information about the ordered products, cannot exist without an *Order* record. It is obvious that the *OrderDetails* table will have as a part of its key, the key for the *Orders* table and that both fields are assigned the semantic name *[Order] Id*. In this case, there are actually two field mappings to the same semantic name. Here the general heuristic is to choose the primary key instance (*Orders*) unless the user selects attributes from the *OrderDetails* table.

M-N and M-N dependent: Any many-to-many relationship will result in multiple field mappings to a single semantic name because the relationship is structured by constructing a joining table whose key is the combination of the keys of the two related tables. Consider, for example a database storing information on books and authors. Since a book may have multiple authors and an author may write

multiple books, a joining table *BookAuthor* ([*Book*; *Author*]) is necessary to implement the M-N relationship between books and authors. The *BookAuthor* table has mappings to both the *Book* ([*Book*] *Id*) and *Author* ([*Author*] *Id*) table keys. This table is not shown in the integrated view and the query engine must select the appropriate joins to be executed.

There is one special case when a semantic name may have multiple field mappings. When a database is not normalized, multiple fields in a single table may map to a semantic name. The semantically correct query should automatically normalize the data by splitting one record into many normalized records. A special case arises too when mappings exist to multiple fields that belongs to different tables within the same database. The query system first selects a field which is currently present into the tables already in the query. Otherwise, it chooses the mapping based on the shortest join paths to the current tables in query.

This is done to identify the most logical semantic choice for the field. Presumably, this identifies the most common occurrences of the field and often is the primary key of the parent table. The algorithm, that is executed for every database, is presented in the figure in the next page and constructs a set of fields (F) and tables (T) which best map to the set of query nodes $Q=Q_1, Q_2, \ldots, Q_n$ given by the user.

For example consider the query "SELECT *[Book]Price, [Book] Author, [Book] ISBN* " that is issued in a database with the following two tables:

Book (ISBN, Price, Author1, Author2)
BookDetails (ISBN, LibraryIndex)

A simple ontology is used with a class named *Book* with the attributes *ISBN*, *Author*, *Price* and *LibraryIndex*. The algorithm starts with *[Book] Price*. A mapping is found in *Price* column of *Book* table and *Price, Book* are added in the list of fields (*F)* and the list of tables (*T)* respectively. Then the [*Book] Author* element is going to be processed. Two mappings are discovered in only one table. So the fields *Author1* and *Author2* are added in *F* and the table *Book* is added in *T*. When the last element *[Book] ISBN* is going to be processed two mappings are found but in two different tables. The algorithm should decide which one of them will be added. As we can see the table *Book* already exists in T. According to our algorithm if a table already exists

in T then the mapping that involves this table is chosen. That's why the *ISBN* field from table *Book* is selected and added to *F*. If the table *Book* did not exist in *T* then our algorithm would select the table with the minimum join distance to the current tables in *T*. When we say the table with the minimum join distance we mean the table that has the minimum distance from any of the tables that already exist in T. The specific algorithm is shown in Figure 11.

```
Input: Query Nodes Q=Q1, Q2, …, Qn given by the user
Output: A set of fields (F) and tables (T)
1:      For each Qi
2:      {
3:              SNi = semantic name of Qi
4:              search_Xpec ( SNi, R )
5:              //search for SNi in X-Specs. Return results in R
6:              IF SNi has only one mapping in R
7:                      Add field Rk to F
8:                      Add parent table of Rk to T
9:
12:             IF SNi has multiple mappings all in one table
13:                     For each result Rk in R
14:                             Add field Rk to F
15:                     Add parent table of R1 to T
16:
17:             IF SNi has multiple mappings in several tables
18:                     IF mapping Rk is found that the parent table of Rk already in T
19:                             Add field Rk to F
20:                     Else select the mapping that leads ot the shortest join path to
21:                     the current tables in Query
22:                             Add field Rk to F
23:                             Add parent table of Rk to T
24:
25: return T,F
```

Figure 11. Field Selection Algorithm

In order to show how joins are handled, we have firstly to define a join graph. A *join graph* is an undirected graph where each node corresponds to a table in the database, and there is a link from node N_i to node N_j if there is a join between the corresponding two tables. For this discussion we ignore multiple joins between two tables on different keys. Moreover, a *join path* is a sequence of one or more joins interconnecting two nodes in the graph, and a *join tree* is a connected subset of the

join graph. Let's assume without loss of generality that the join graph is connected (otherwise, we apply the algorithm to each connected subset and connect them using a cross-product). Then, we can conclude to the following lemma.

Lemma 1. If a join graph is acyclic, there exists only one join path between any two nodes.

Proof. We will prove this lemma by using contradiction. Let's assume that two join paths exist between two nodes N_i and N_j. Then, we could take the first path from N_i to N_j and return on the second path from N_j to N_i. This implies that the graph has a cycle**.**

Moreover, we can conclude the following lemma too.

Lemma 2. If a join graph is acyclic, there exists only one join tree between any subset of its nodes.

Proof. For two nodes the statement is true as we proved in lemma 1. Given a subset of m nodes where the lemma holds we will try to prove that lemma also holds if we add one more node. So, given a subset of m nodes with only one join tree, we add another node N to the set. Assume that by adding N there exist more than one join tree in the new subset of m+1 nodes. Since there was only one join tree for the previous m nodes, this implies that N must be connected to more than one node in the subset. It is obvious that this produces a cycle. Thus, the statement holds for m+1 nodes and the result follows by induction**.**

The consequences of lemma 2 are really important. If the join graph for a database is acyclic, there exists only one possible join tree for any of its tables. That means that the query system does not have to make any decisions involving which joins to apply. It has to identify which joins are required to connect the required tables by constructing the proper join tree. The order in which the joins are applied is a problem of optimization that will be discussed later in this chapter.

From this result, it is possible to construct an algorithm which builds a matrix M where entry M [N_i, N_j] is the shortest join path between any pair nodes N_i and N_j.

Theorem. Given a subset of nodes from a matrix M which stores the shortest join paths for an acyclic join graph, and a set of tables T to join, a join tree can be constructed by choosing any table T_i from T and unioning the join paths in M [N_i, N_1], M [N_i, N_2], …, M [N_i, N_n] where N_1, N_2,…, N_n are the nodes corresponding to the set of tables T.

Proof. Since the graph is connected, the matrix entries M [N_i, N_1], M [N_i, N_2], …, M [N_i, N_n] represent join paths from N_i to all other nodes in the subset. Thus, there is a path from N_i to N_j and from node N_i to N_k. Unioning those paths together results in a path from N_j to N_k. Thus, all nodes are connected with the join tree, and it is the only possible join tree as we proved with lemma 2**.**

Normalized databases often have acyclic join graphs. However, we cannot assume that all databases would be acyclic, and the general case of a cyclic join graph must be considered. Cycles arise when joins are added for query convenience and when tables serve multiple semantic roles in a database. A given table can assume multiple semantic roles in a database, by acting for example as a lookup table for several others. For example, consider the tables *Orders* and *OrderDetails*. We can add another one table called *Employees* which will store information on the employee who entered each product in addition to what employee entered the overall order. In this case, *Orders* and *OrderDetails* have foreign keys to the *Employees* table. This produces a join, and according to the join path chosen, different semantic queries are represented. For example, the join path *Orders-Employees-OrderDetails* represents the orders entered by employee with their products whereas *Order-OrderDetails-Employees* represents the orders with their products along the employee entering the product. Moreover cycles often occur when a table stores a generalized concept which may have multiple sub-concepts, where several tables join to the different semantic instances in the generalized table.

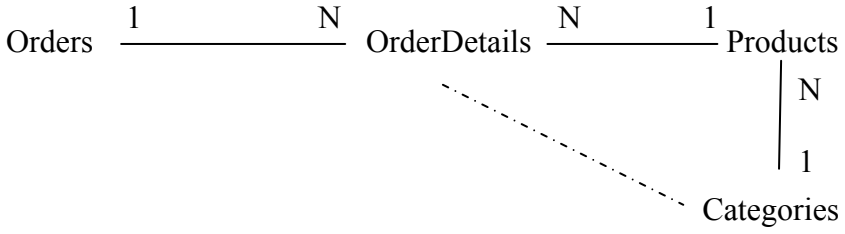


Figure 12. Join Graph Example

Finally, cycles may occur when redundant joins are added to the database. For example, the *CategoryId* field could be added to *OrderDetails* for a direct link to *Categories* instead of joining through *Products*. This results in a cycle involving

OrderDetails, Products and Categories. Note that joins of this nature may be lossy when used in combination with other, valid lossless joins. An invalid lossy sequence of joins results when a join with a N-1 cardinality is followed by a join with a 1-N cardinality where the join attribute is not a key. There may be other joins between those two joins. The result is a lossy join because it results in a M-N cardinality relationship between the merged tables. Effectively, this results in invalid information being created by using these joins. Also, a join of cardinality M-N between two tables, without using an intermediate table, is always lossy. Of course, such databases are not normalized and we expect that most of the databases today are normalized ones. Thus, the algorithm first should attempt to find join paths without using these types of lossy joins.

To handle cycles, the query system must make a determination of the best join paths between nodes. The query system uses join semantics, path length, and join properties such as total participation, lossless or lossy joins to determine best join paths. The breadth-first algorithm presented constructs the matrix M of best join paths and it works for both cyclic and acyclic join graphs. The algorithm selects the shortest join paths with no lossy joins and equal length join paths may be differentiated based on total participation or other join properties. Lossy joins are only used if there exists no other path between nodes (a cross-product would be necessary).The specific algorithm is shown in Figure 13.

For a specific example, we will try to build the matrix M for the graph shown in figure 12. Starting from node *Orders*, initially M[*Orders*, *Orders*] is zero, *count* is zero too and we do not accept lossy joins. Then we add the *Orders* node to our FIFO queue *NQ* and since *NQ* is not empty we remove the first node *N* from *NQ*. So, *N*= *Orders*. Since there is only one outgoing link from *Orders*, *LTN = OrderDetails* and since it is not visited and we have no lossy joins, it is added to *NQ*, it is marked as visited and *M[Orders, OrderDetails]= M[Orders, Orders] + OrderDetails.* Then, count is set to one, *NQ* is not empty and *N = OrderDetails*. The only outgoing destination from *OrderDetails* is *Products*, so *LTN = Products*. The *Products* node is not visited yet, so we add *Products* to *NQ*, we mark it as visited and *M[Orders, Products] = M[Orders, OrderDetails]+Products= OrderDetails + Products.* The algorithm goes on the same way until the full matrix M is constructed. We would like

to note that for each node *F* in *G* the algorithm will go through line 7 at most two times, since by finishing the second round *count* will be equal to *#of nodes* in *G.*

```
Input: G as a graph
Output: Matrix M              // N x N matrix where N the number of nodes in G
1:  For each node F in G
2:  {
3:      M [F, F] = Null // Empty join path to itself
4:      count = 0
5:      accept_lossy = false   // initially do not accept lossy joins
6:
7:      While count < # of nodes in G
8:      {
9:        add F to NQ //NQ is a FIFO queue structure
10:
12:       While NQ is not empty
12:       {
13:             remove first node N from NQ
14:             For each outgoing link L of N
15:                   LTN = destination node of link L from N
17:                   If LTN is not visited and (accept_lossy or the path has not
                      a lossy join)
18:                         add LTN to NQ
19:                         mark LTN as visited
20:                         M [F, LTN] = M [F, N] + LTN
21:                         count++
22:                   ElseIf accept_lossy or the path has not a lossy join
23:                         //you may want to replace a join path already
24:                         //constructed
25:                         //if new join path is the same length as current and
26:                         //new join path has better properties (total particip.)
27:       }
28:       clear_flags() //clear all visited flags for all nodes in G
29:
30:       accept_lossy = true
31:     }
32:  }
33: return M
```

Figure 13. Algorithm to Calculate Join Paths.

It would be ideal if we could use the algorithm of unioning join paths in the matrix to produce a join tree between any subset of nodes. However, if the graph is cyclic, there will be multiple join trees possible depending on the choice of starting

node. These join trees are all semantically valid depending on the query and the system cannot differentiate them for the user without more knowledge about the intended query semantics. Some work started to emerge in the area [Mason et al. 2005] but finding heuristics that could choose the best join tree based on the attributes chosen for the query is beyond the scope of this thesis and is included in our future work. So, our system cannot handle cycles and lines 23-25 of the algorithm in Figure 13 have not been implemented.

Whereas we cannot differentiate all semantic valid join trees when we have cycles, we can use "smart tricks" in order to avoid confusions. So, when the administrator constructs the X-Spec file, he can choose which valid paths to represent. It is not mandatory to represent the whole underlying schema and every relation across tables. He can choose only the parts that are of interest and if he wants later, he can add more relations or more tables. So when we have cycles we can choose which join tree to be constructed and we can declare an acyclic join tree.

Moreover, it is possible for the user to declare explicit joins in the *where* clause that denotes the join path that his query will use. Whereas usually, joins paths are hidden from the users and the user doesn't have to know the structure of the underlying database, it is possible if desired and if he knows the underlying schema to declare the explicit joins to be performed. Of course we do not expect from users to have in mind the underlying schema, but we give them the option to decide if such knowledge exists.

5.5. Join Algorithms

Except from determining the correct join path, an essential matter is to choose the proper join algorithm in order to efficiently answer the queries issued. Since the most costly operator is the join one, an important issue is to determine the more efficient algorithm to perform joins in each case. In centralized databases, this research area has been extensively studied and every database management system has an optimizer that chooses the best join algorithm (or nearly the best) to use in each case.

Whereas, a lot of work has been done in centralized databases, in distributed database systems there are a lot to be done. As we mentioned before, we can determine in most cases the proper join path needed to execute the query issued in our system. In that join path we can distinguish which joins need to be performed across tables that belong to the same data source or to different data sources. Having in mind the goal to encapsulate the heterogeneity of the component databases and to use existing homogenous distributed database techniques as much as possible we used underlying databases for applying specific joins.

Since our policy is to use existing database techniques we "push" the joins that interfere tables within the same database to local databases and we leave the joins that span across data sources, to be handled from our system. In order to join data across data sources, two algorithms have been implemented.

5.5.1 Main Memory Algorithm

After the query is issued in our system, and it is decomposed into subqueries, these subqueries are executed in parallel, independently in each data source. So the time to execute the individual queries depends on the query that takes more time to be executed and transferred. When all the results from the independent data sources are loaded into the memory of our system, the join algorithm is being executed and as soon as we have some results they are presented to the final user.

5.5.1.1 Nested Loops

The first join algorithm that was implemented in our system, in order to study the join implications was the simple nested loops algorithm. This join algorithm may not be the more efficient join algorithm, but it is really simple to implement in a mediator-based environment. If we want to join two relations with simple nested loops, for each tuple in the outer relation R, we scan the entire inner relation S as we can see in the following figure.

```
Input: R, S relations
Output: Join result
1:      for each tuple r in R
2:              for each tuple s in S do
3:                      if   r_i = s_i then
4:                              add <r, s> to result
5:      return result
```

Figure 14. Simple Nested Loops Join

If the smaller relation becomes the outer one, the algorithm is more efficient since its cost is: Total Cost = (tuples per page in R * #of pages in R) * #of pages in S + #pages in R. Of course this is in centralized databases. Here, in the total query cost we have to add the time to get the results from the individual sources (local query time + communication costs) and to load them into memory.

5.5.1.2 Result Processing

Since, we load the result of each subquery into the main memory; it is our task to process them further if order, group and union operations have to be applied.

If no join condition is specified the results of each database are being unioned according to their shared global key. Because each tuple presented, is constructed in our system, we can choose to accept unions of tuples that their schema does not fully match. If for example in one database a field is missing we can allow union to be performed with another database where that information exists, and whenever that field is missing is left blank.

Ordering operations should be considered before showing the results. In order to apply these, we use the *Quicksort* algorithm to sort the results according to the required criteria. Grouping operation has not been implemented yet but in our near future we are going to examine them.

5.5.2 Central Database Algorithm

Whereas simple nested loops were efficient for joining a small number of tuples, when the number of tuples that need to be joined increased, the algorithm became really slow. The first thing that came into our mind was to try and implement

several other algorithms, such as hash-join, sort/merge joins, and hash joins. After that we would build an optimizer to decide which join policy to use in each case. Doing all that, we would re-implement several well implemented algorithms (in central databases) and we would try to build something that is already well-done by several database vendors.

Having that in mind, in addition to the principle of using as much as possible of the existing homogenous distributed database techniques, we leaded to the construction of a new join algorithm that could exploit current DBMS systems. The algorithm consists of the following six steps and it is implemented in our system.

1. For every sub-query issued in each independent data source find the table that should be constructed in a central database in order to store the results of that sub-query.
2. Build those tables in a central lightweight database.
3. While executing sub-queries, store their results into those tables created in the first step.
4. Build a new join graph based on the results stored in the central lightweight database.
5. Build the global query that should be issued in the central database.
6. Execute that query, get the results, and present them to the final user.

The first three steps are being executed in parallel for every existing data source and they are implemented using threads. Parallel execution used since each subquery concerns only a single independent data source. After the results of each independent subquery are stored in a single database, we can build one single proper query based on the relations stored in memory. Thus, each join that needs to be performed across databases is performed within a DBMS. Of course, there is a payload to the whole procedure, which is the cost to build the proper tables in a central database, and the cost to store the results returned from each individual data source within the central database and the cost to build and execute the single query issued in the central database.

Using this algorithm not only improves the time to execute joins across databases, but also has valuable side effects. It can be used to implement several

caching policies since after executing each query; data remain in the central site and can be used to answer future questions concerning the same fragment of data. Of course, matters of caching are beyond the scope of this thesis and are indented to be examined in our future work.

5.5.2.1 Building the tables.

Building the tables needed to store the results of the individual subqueries is a rather trivial matter. The only thing that needs to be examined is the subquery issued in the individual source, and of course the information about the fields queried, that is stored in the appropriate X-Spec.

At first, each subquery is examined to define the returned fields in its select clause. Then a table is generated with a random name which is built in such a way that is unique in our lightweight database. The fields of that table are named after the fields in the select clause of each sub-query. Except from the field's name, their data types should also be known in order to build the proper table to store these results. This is really simple too, because in X-Spec we have all the information needed about the data type and the length of each field and we can use that information to build the proper tables. For example if the query issued in data source 1 is:

```
select B.bioAssay, R.ReporterID, B.Intensity1, B.Intensity2
from bioAssayData as B, reporter as R
where R.id=B.Reporter
```

the table TempTable12387986 (B_bioAssay int, R_ReporterID varchar(50), B_Intensity1 int, B_Intensity2 int) is being constructed. The field's data type corresponds to the data types of the selected fields in their individual data sources. After building one such table for each data source, the results returned from each individual subquery are being inserted in the appropriate table in our central database.

5.5.2.2 Building the Query.

When all data needed, are found in our central database the next task is to build one query that will combine them in a proper way. In order to build the query

that will be issued in our central database we need to examine the tables created in our database and the fields that correspond to those.

A general policy is to build joins across tables that come from vertical distributed tables and then to union tables that have the same schema or more properly, that their schema corresponds to the same semantics. Finally the Cartesian of the tables is produced and if they share the same global key, join conditions are applied. A global key is a key that has the same semantic meaning across two or more data sources, and as a result tuples coming from different databases should be joined.

Of course, we have to admit that these assumptions are valid, under the hypothesis that the fields mapped across data sources share the same domain where their values belong. So when in data source 1, the field *bookId* of table Writings with semantic name *[Book] Id* has value 1 and in data source 2, the field *Id* of table books, with the same semantic name *[Book]Id*, has value 1 too, we assume that we are referencing to the same element that is the same book instance.

Since all the operations are performed within our central database, ordering, grouping, etc. can be performed by the database itself and we do not have to implement algorithms for those operations. Of course, because databases cannot perform union of tuples that do not have the same schema, relations with partial schema cannot be unioined using this policy.

5.6 Considering Distribution

In our system, is possible to declare fragmentation vertical of horizontal. Both vertical and horizontal fragmentation may exist and should be declared when data sources with fragmented data are going to be integrated into our system. The benefit from fragmentation is that queries that involve only specific fragments of data don't have to involve the whole data of the table.

Moreover if a table is horizontally distributed across several data sources, the system will recognize selections on fragments that have a qualification contradicting the qualification of the fragmentation rule and will remove them, since they produce empty relations. If selections are made across fragments that do not contradict the qualification of the fragmentation rule then the union of two selections will be returned.

In the case of vertical distribution, when a query is issued containing information from these two tables, a join between these tables should be produced. This is done when results are being formulated in our central site. So, whereas the definition of horizontal fragmentation rules has as a result the elimination of empty queries contradicting those rules, defining vertical fragmentation rules assures that the correct joins will be made across databases that share vertical fragments of the same table. Consider for example the table *Reporters* (*Id*, *Name*, *Species*, *Date*) which is horizontally distributed in two databases, the first containing the tuples with *Id* less than 500 and the second one tuples with *Id* more than or equal to 500. Imagine now that someone decides to fragment vertically the table in the second database for reasons of performance. So, two tables are being created, *Reporters* (*Id*, *Name*) in the second database and *Reporters* (*Id*, *Species*, *Date*) in one third database. Consider now a query that asks for every field of table reporters. If knowledge about fragmentation exists, the tables from the second and the third database will be joined, and then because the resulting table will have the same schema with the results from the first database a Union will correctly produced. But if no such knowledge exists, results from the first database and the second one may be joined since they share the same global key, and because they belong to the same horizontally distributed table no results will be returned.

5.7 Example

Consider for example that we have two databases which store information about books. In the first database there are the tables *Library*, *Book*, *Publisher* and *Author*. A library has many books and each one of them has a publisher and an author. The second database stores information about the location of books that also have only one author. A simple ontology is built describing books, and we use that ontology to annotate the X-Spec produced from the two distinct databases. Assume that after building the integrated view in memory the following semantic query is issued:

SELECT [Library] Name, [Book] Id WHERE [Book; Author] Name = "X"

Using the field selection algorithm for the first database we conclude that the fields *Library.Name*, *Book.Id*, *Author.Name* and the tables *Library*, *Book* and *Author* will be used. For the second database, the fields *Location.Name*, *Copy.Id* and *Author.Name* and the tables *Location*, *Copy* and *Author* will be used. In each case the first two fields will be used for projection and the *Author.Name* field will be used to form selection criteria. For the two databases, the join graph is shown in the following figure. Having the projection fields, the fields that will be used for constructing the selection criteria and the join graph, we can build the proper sub-queries that will be issued in our two districts databases. Those queries are:

Db1: *Select Library.Name, Book.Id From Library, Book, Author*
Where Author.name = "X" and Library.Id = Book.LibraryId and
Book.Author = Author.Id

DB2: *Select Location.Name, Copy.Id From Location, Copy, Author*
Where Author.name = "X" and Location.Id = Book.LocationId and
Copy.Author = Author.Id

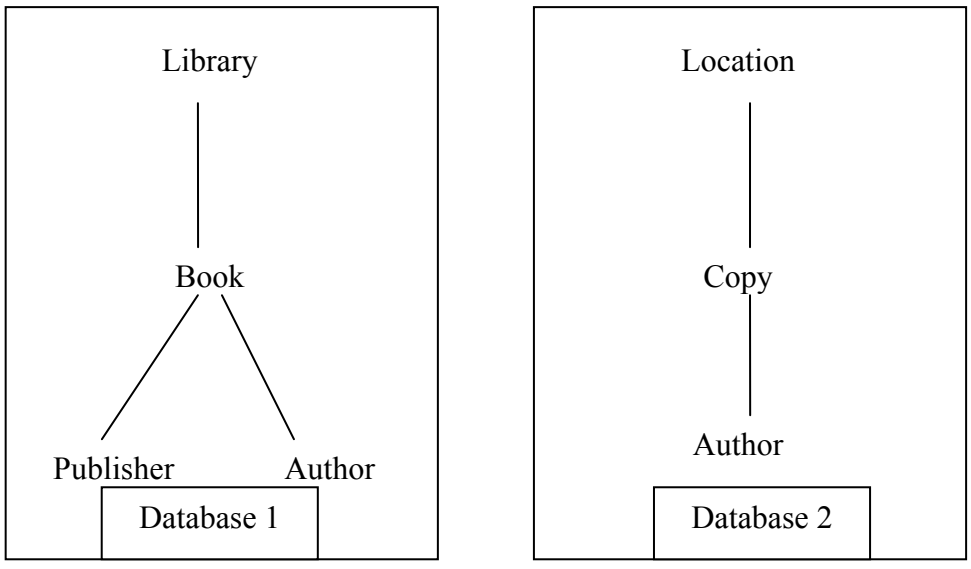


Figure 15. Join Graphs for Database 1 (left) and Database 2 (right)

Those sub-queries are issued in the two distinct databases, using threads. Assume that the Database algorithm is used. From these two sub-queries and using the information about queried fields from X-Spec, we can conclude that the following two tables should be constructed in our central database.

TempTable123123 (Name varchar (50), Id int)
TempTable321321 (Name varchar (50), Id int)

The results from the two sub-queries are stored in those tables and then one global query should be constructed to be issued in our central database. Since the schemata of these two tables correspond to the same semanctics, the UNION operator should be used. As a result the final query to be issued is

Select Name, Id from *TempTable123123*
UNION
Select Name, Id from *TempTable321321*

That query is issued in our central db and the results are presented to the final user. If the memory algorithm is used the system will recognize that the results coming from the two sub-queries correspond to the same semantics and as a result a union operation is required. That operation will be performed in memory after the results from the two sub-queries are returned and then the final result will be presented to the user.

Chapter 6

QueTe Implementation and Evaluation

"Knowledge is of two kinds. We know a subject ourselves, or we know where we can find information on it"

-Samuel Johnson

Contents

In this Chapter we are going to give an overview of our implementation and show the decisions made while developing QUETE. We will give a simple example of integrating two data sources and we will describe the necessary steps that need to be performed. After the implementation has been fully understood we are going to evaluate QueTe based on the demands of the project PROGNOCHIP.

6.1 QueTe Implementation

As we have shown, our architecture is capable of handling large-scale integrations in evolving environments, where the specific databases participating in the whole system change frequently and their schema evolves over time.

Moreover, the system's database engine integrates distributed data sources without requiring middleware or database server support and allows programmers to access easily, several integration algorithms. The whole system is implemented in Java. Java was used because it is currently the standard language to develop web applications. It supports native multithreading and provides several distributed programming facilities. Moreover, a program written once in Java can run in any platform and in any operating system desired.

Furthermore, we adopted Unity's policy to implement the whole system within a standard JDBC driver, because providing a standard interface is essential for unified querying of heterogeneous databases. The JDBC standard allows the execution of queries in a general programming environment by providing library routines which interfere with the database. Most users and programmers are familiar with using the standard JDBC driver in order to interact with a single data source. We are using the same functions, and the same API to provide transparent access to multiple data sources instead of just one. In particular, JDBC has a rich collection of routines which make the interface simple and intuitive and provides portability since users are allowed to develop their own programs and interfaces using our driver. An example application using the driver is shown in the Appendix. In every application built, our driver should be explicitly declared to be used initially. Moreover the *URL* of the configuration file that will be described later in this chapter must be declared. Then one can use our driver exactly as the common JDBC driver. Note that the driver could be used even when no Ontology is used and no conceptual querying is performed. In this case, all relations from all databases are imported into the global view but not matched. Thus, at the lowest level, the driver functions as a standard federated system allowing distributed access to the data sources. However, its true benefit is abstracting away the challenges of building joins and matching schema constructs manually.

The whole system, besides the JDBC driver has several components that should be described.

6.1.1 X-Spec Specification Documents

As mentioned before, a standardized ontology is not enough to achieve integration, because a standard schema for communication is not defined. Data concepts can be represented in vastly different ways in various data sources thus we need a system for describing the schema of a data source using ontology terms and additional metadata. We use X-Spec to store all that relevant information.

An X-Spec consists of the relational database schema being described along with additional information about keys, relationships, and field semantics. More importantly, each table and field in the X-Spec has an associated name built from terms in the standardized ontology.

```
<TABLE>
      <semanticTableName>SAMPLE</semanticTableName>
      <tableName>sample</tableName>
   <FIELD>
      <semanticFieldName> [SAMPLE] ID</semanticFieldName>
      <fieldName>id</fieldName>
      <dataType>4</dataType>
      <dataTypeName>int</dataTypeName>
      <fieldSize>10</fieldSize>
      <decimalDigits>0</decimalDigits>
      <numberRadixPrecision>10</numberRadixPrecision>
      <remarks>null</remarks>
      <defaultValue>null</defaultValue>
      <characterOctetLength>0</characterOctetLength>
      <ordinalPosition>1</ordinalPosition>
      <isNullable>NO </isNullable>
    </FIELD>
     <PRIMARYKEY>
      <keyScope>4</keyScope>
      <keyScopeName>Global</keyScopeName>
      <keyName>PK_sample</keyName>
      <keyType>1</keyType>
      <FIELDS>
      <fieldName>id</fieldName>
      </FIELDS>
   </PRIMARYKEY>
   <JOIN>
    <joinName>sample->extract</joinName>
    <fromKeyName>PK_sample</fromKeyName>
    <fromTableName>sample</fromTableName>
    <toKeyName>FK_extract_sample1</toKeyName>
    <toTableName>extract</toTableName>
    <joinType>2</joinType>
   </JOIN>
```

Figure 16. Example X-Spec

An example X-Spec is given in the previous figure. As we can see the document is an XML document. In the beginning, we can see that table *sample* is annotated with the semantic name *SAMPLE* from our ontology. After annotating each table, we have to describe table fields too. In our example field *id* is being described. Firstly a semantic name is given to that field ([*SAMLE] ID*) and then information is shown about the type of the field. The current field has a data type no 4 as given in java.sql.Types and it is an integer (*int*) with size of ten. The *dataTypeName* is data source dependent and it is not enough for the specification of each field because each database may represent differently its own data types. Furthermore, the number of fractional digits (*decimalDigits*) and its radix (*numberRadixPrecision*) are defined and any comments about the field are given (*remarks*). Moreover, it is declared whether the field can accept null values (*isNullable*), its default value (*defaultValue*) and the index of column in table's definition (*ordinalPosition*). Finally if the field is a char, the maximum number of bytes in the column is given (*characterOctetLength*)

Except from specifying the specific attributes of each field in a database independent way, the relations across tables should also be declared. As shown in figure the primary key of each table should be declared. In primary key declaration, each field participating in primary key is shown, a unique semantic name is given for that primary key (*keyName*) and as well the type of that key (*keyType*) - 1-primary, 2-foreign, 3-alternate or candidate. Moreover, the scope of the key is declared (*keyScope*) along with the name of the scope that this key participates (*keyScopeName*). Those declarations specify the scope of the keys they are valid and are used to match global keys across databases. If the primary keys within several databases have the same semantic name and the same scope, then the same global key is used and it will be used to join the subquery results. Whereas in our implementation every field is annotated using one single ontology and as a result they belong to the same scope, it is possible several ontologies and scopes to be used. Information is given for foreign keys too the same way with the primary ones.

Finally if joins exist, they should also be explicitly declared as shown in the figure. The type of the join (*joinType*) is essential (1-1, 1-N, M-N) and the keys (*fromKeyName*, *toKeyName*) and tables (fromTableName, toTableName) that participate in the join should be given. More examples are shown in the cd that accompanies this thesis.

6.1.2 X-Spec Extractor

It is obvious that the construction of an X-Spec with a lot of tables is really time consuming whether it is really simple and trivial for each administrator. That's why an Extractor is provided with the whole system and the only thing that is required in order to be executed efficiently is the connection string (database, username and password) of each database.

The Extractor will create an X-Spec automatically for a specific database in the proper format. All information relying in underlying data sources will be gathered and recorded in the output X-Spec. Of course, in order for the Extractor to create fully formed X-Specs, primary and foreign keys must be specified within the databases being extracted.

After extraction, the annotation of each field and each table using reference ontology remains to the hands of the administrator. He should give afterwards semantic names in all fields of interest, that will be integrated using our system, and decide which tables, fields and joins should participate in the integrated schema. The tables that are not going to participate can be removed from each X-Spec, and if cycles exist, specific join paths can be removed by eliminating joins among tables within the same X-Spec.

6.1.3 Configuration File

After the extraction and the annotation process, all files generated should be placed in a central directory from where our system will use them, in order to properly answer the queries issued. One final configuration file has to be created that describes what data sources are being integrated and where their X-Spec files are stored. An example is shown in the following figure.

```
<SOURCES>
        <DATABASE>
                <URL>jdbc:odbc:Base</URL>
                <DRIVER>sun.jdbc.odbc.JdbcOdbcDriver</DRIVER>
                <XSPEC>xspec/Base.xml</XSPEC>
        </DATABASE>
</SOURCES>
```

Figure 17. Configuration File for Base

As we can see in the example, the configuration file is an XML file too. In the *URL* tag is written the connection string used from the system to connect with each data source. In the current example, the specific data source is connected through *ODBC* and is given the name *Base* (note that this source should be declared in the *System DSN* of the *ODBC data sources* of the machine where our system is installed). Since our system is using ODBC, the proper ODBC java driver should be used to interact with that data source. This is declared in the *DRIVER* tag. Finally, the place of the X-Spec corresponding to the specific data source must be declared, and this is done in *XPEC* tag.

The goal of *ODBC* is to make possible to access any data from any application, regardless of which database management system is handling the data. *ODBC* manages this by inserting a middle layer, called a *database driver*, between the application and the DBMS. The purpose of this layer is to translate the application's data queries into commands that the DBMS understands. This specific characteristic of ODBC makes it ideal for integrating different databases under a common API.

6.1.4 Vertical and Horizontal Distribution

As shown in the previous chapter, the system is optimized for horizontal and vertical, distributed, relational data sources. In order to benefit from these optimizations, somehow the distribution must be declared from the administrator.

System can support horizontal fragmentation based on simple selection predicates. Fragmentation rules are declared as:

Data_Source: Table: Field *predicate* value

For example consider table *Sampes*(*SampleId*, *SampleData*, *SampleDate*). A possible horizontal fragmentation across two databases denoting that samples with id higher than 500 are stored in db1 and the rest in db2 could be declared as.:

DB1: Samples: SampleId > 500 and DB2: Samples: SampleId<=500

Selection predicates could be <, >, =, <=, >=, <>, and values could be either numeric or strings within ‘ or “. The system will recognize selections on fragments that have a qualification contradicting the qualification of the fragmentation rule and will remove them, since they produce empty relations. If selections are made across

fragments that do not contradict the qualification of the fragmentation rule, the union of two selections should be returned.

Moreover, system can support vertical fragmentation and the fragmented tables are denoted to belong to the same table. For example, if a table has been fragmented into two tables Diagnosis and Samples that remain into different data sources we can declare:

```
Vector fragment1=new Vector();
fragment1. add("Data_Source1:Diagnosis");
fragment1. add("Data_Source2:Samples");
```

Those tables are vertical fragmented according to their primary keys. If selections are made across vertical fragmented tables, then the join of these tables should be produced.

6.2 Evaluation

After providing X-Specs, configuration file, and the fragmentation rules, the system is ready to answer every question issued transparently and efficiently. To show system efficiency exhaustive testing and evaluation has been performed. Here we will only present, the evaluation based on the needs of project PROGNOCHIP. Detailed experiments were performed in order to study the performance of the system on the previous listed algorithms.

Our resources were limited, so we used three machines with an Intel Pentium III processor on 1.0 GHz, and 256 MB of RAM. Our system achieved good performance even in these slow machines and we expect great results when more powerful machines are used. Those machines were on a 10/100 Mbps LAN.

6.2.1 Starting Point - Simple Database Case Study

In the beginning of our evaluation, we built two simple databases that were placed in the same machine. Those databases were in Microsoft Access and because they were placed on the same machine with our system there were no communication costs. The schema of those databases is shown in the following figure.

Database 1

Patients

PatientId	PatientName	PatientSurname

Extractions

PatientId	SampleId	ExtractionProtocol

Database 2

Samples

SampleId	SampleData	SampleDate

Clinical.Extractions.SampleID == Biological. Samples.SampleID

Figure 18. Example Database Schema

In this simple case study the focus was to examine the performance of our system using the memory algorithm only, against the JDBC driver created by SUN. Of course our system provides transparency to users and ontology based queries whereas JDBC is a simple driver for single database access.

We annotated our schema using a really simple ontology built only for those tables. Our ontology consisted of two classes: *Patients* with the attributes *PatientId*, *PatientName* and *PatientSurname*, and *Samples* with the attributes *SampleId*, *ExtractionProtocol*, *SampleData*, *SampleDate* and *PatientId*. All those attributes were mapped in the underlying data sources at X-Spec creation.

We firstly tried a simple query against a single database. The query was to select all the Patient Ids from Patients. We run each query 10 times using our implementation and then the JDBC driver. As we can see in the following figure, JDBC had a better performance than our implementation, as we expected. Our system had to load into memory the schemata, to build the correct paths and to transform the semantic query to SQL, things that add a little overhead to our implementation. Moreover, our system is implemented in a way that after selecting the tuples and loading them into memory, every tuple has to be examined for checking if more actions have to be performed on our central site, even if our result comes from a single database. This adds an overhead relative to the number of tuples returned. In our future plans is to optimize the whole procedure.

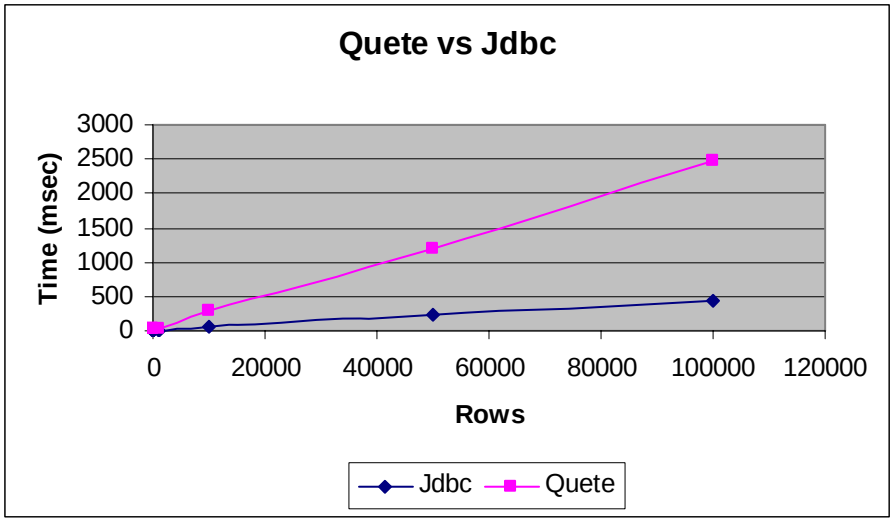


Figure 19. Quete versus Jdbc in a single select query

The real advantage of our system is the transparent access to multiple, heterogeneous databases. In order to check the performance of our system in such an environment we tried to issue a query that would involve a join between two tables across databases. Of course this action cannot be performed by the JDBC driver, who can only ask separate databases. That's why, in order to make estimation about the JDBC driver we issued "hard-code" the decomposed subqueries in the two data sources that our system would automatically produce. The results of these subqueries where then stored in a local database, and then another hard-coded query was sent to ask the local database for the final results. We have to note that all results from the two separate databases were joinable. The query issued in our system was:

```
Select  [Samples]PatientId, [Samples] SampleId, [Samples] SampleData;
```

That query, decomposed into the two following subqueries issued in the two underlying databases:

```
Select E.PatientId, E.Sample From Extractions as E
Select S.SampleId, S.SampleData From Samples as S
```

As we can see, in the following table our system has a better performance in cases where a small amount of rows is selected and joined. But when a lot of tuples appear the performance degrades quickly

Rows		Jdbc-Odbc	Quete
DB1 Extractions	DB2 Samples		
5	**5**	**50 msec**	**30 msec**
10	**10**	**50 msec**	**30 msec**
100	**100**	**330 msec**	**71 msec**
1.000	**1.000**	**2750 msec**	**3198 msec**
10.000	**10.000**	**26533 msec**	**308945 msec**
50.000	**50.000**	**134841 msec**	**A lot of sec**
100.000	**100.000**	**267198 msec**	**A lot min**
10.000	**10**	**80 msec**	**1081 msec**
10.000	**100**	**380 msec**	**3455 msec**
10.000	**1000**	**3475 msec**	**30958 msec**
50.000	**10**	**161 msec**	**4345 msec**
50.000	**100**	**471 msec**	**16771 msec**
50.000	**1000**	**3194 msec**	**145892 msec**

Table 2. Joining rows across databases

Having those experiments in mind we started developing the Central Database algorithm we implemented. The memory algorithm was not efficient when a lot of tuples had to be joined.

6.2.2 Prognochip Case Study

Having the second algorithm implemented, the system was tested in real world applications and challenges. Since the motivation for this thesis was the project PROGNOCHIP, measuring the performance of the system when deployed in those databases was really important.

In the beginning, the schema of each database participating in the project was collected and the Extractor tool was used, to capture the properties of each database. Then a trivial, plain ontology was built that was focused on the two databases participating in our project, and the fields of interest where annotated using terms from that ontology. The two databases participating were developed to fulfill different, separate requirements.

The Genomic database stores information about the execution and the result of microarray experiments. Protocols, procedures, and measurements occurring from

several experiments are stored and the whole process of a microarray experiment is modeled and stored. The results of such experiments are then analyzed using statistical methods and are stored in a different partition of the same database. The database has about 85 tables, but according to our ontology only 15 of them are needed to be used in our integration scheme, so only those tables were annotated. The whole database schema is really big to be presented here and can be found in the cd that comes with this thesis. The Genomic database is stored in MySQL, and since the join relations are not shown in MySQL we had to fully understand the design and the relations of those tables and to describe them in the X-Spec files.

Whereas the Genomic database is dedicated to microarray experiments, the Clinical database was built in order to capture all the information needed in a Hospital. So the clinical database has about 500 tables, but in our project only 70 of them are needed. The clinical database is in SQLServer but the relationships among tables are not captured within the database because of implementation and multilingual reasons (-SQLServer provides the capability to store table relationships within database and several constraints coming from these relations are checked when data are updated or inserted). So, we had to understand the whole schema related to the information that our project needed, and to capture the relationships across tables in the X-Spec files. Because cycles existed, whenever a table could be reached from many tables, we chose the more efficient and correct path. This was performed by eliminating the necessary relations from the appropriate X-Specs. The schema and the X-Spec files can be found in the cd and in the end of this thesis.

6.2.2.1 No fragmentation

The performance of the two algorithms implemented was initially tested. The Quete Database algorithm is the one where all the results are stored in a local database and joined there, whereas the Quete Memory algorithm loads the subquery results into memory and joins them using simple nested loops.

In Database 1 was stored the Genomic Schema without the tables that are produced from data analysis, Database 2 contained the tables produced from Genomic data analysis, and Database 3 followed the clinical schema. The central database used, was SQLServer but any DBMS accessed by standard ODBC protocols could be used

as well. We built a benchmark program that was able to load all three databases with a prefixed number of rows that all could be joined within and across sites.

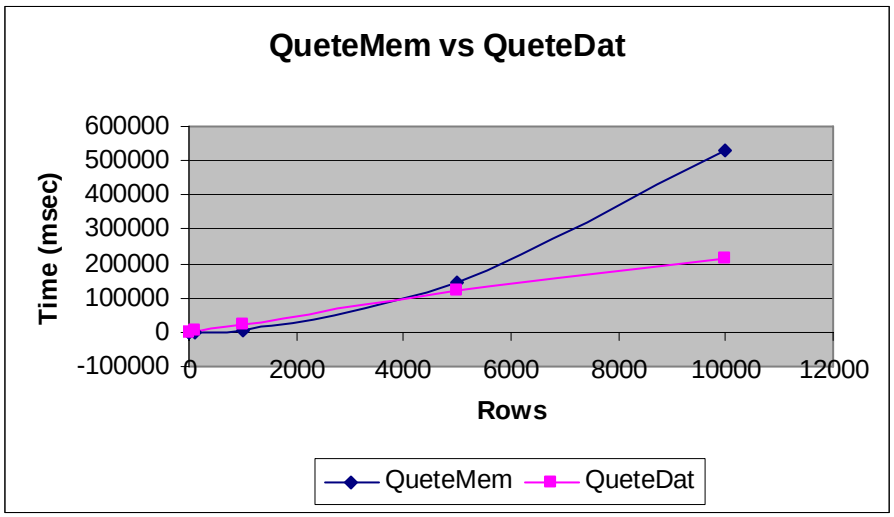


Figure 20. Memory algorithm VS Database Algorithm

The previous figure summarizes data shown in the Appendix and we can observe that the Memory algorithm performs well when a small amount of tables is being joined, whereas the Database algorithm outperforms the Memory one when the data grows. These results confirm that current DBMS can handle heavy-load situations more efficiently than every implementation we might have. The query issued, involved all the tables in the Clinical and the Genomic database, and queries like that will be issued in the final stage of the project.

6.2.2.2 Horizontal fragmentation

After checking the performance of the system in the previous two cases, the performance of the system when horizontal fragmentation existed, was checked. So, we fragmented one large table in db1 such that half of it was put in a new table in db2 and a small fraction of the initial table was put in a new table in db3. Then we submitted fragmentation rules to our system and we issued a query that could exploit fragmentation to achieve better performance. As shown in the figure and its corresponding table in the Appendix, when fragmentation rules are considered, we have a better performance. The performance gained from fragmentation knowledge is

optimal when Memory algorithm is used as shown, whereas in Database algorithm the performance gained is too small. As we can see in the figure, Database algorithm outperforms Memory algorithm.

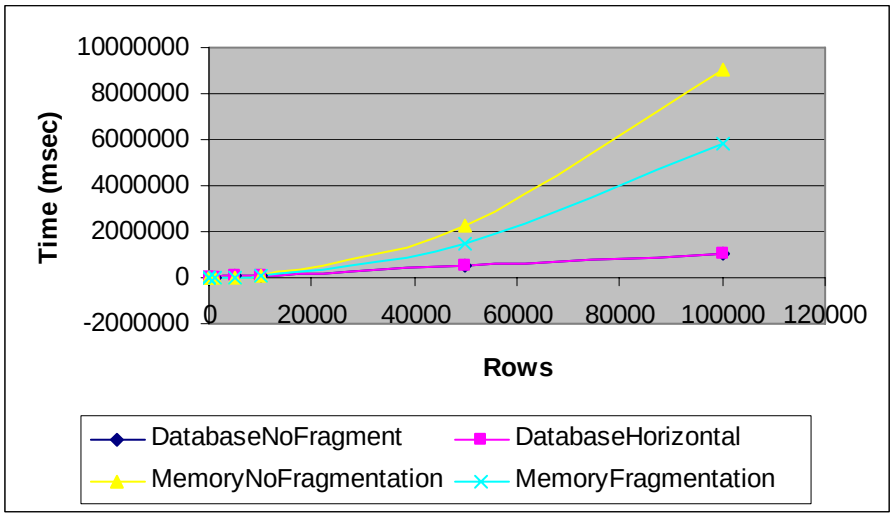


Figure 21. Considering fragmentation rules

Moreover we can conclude from the figure that in this simple case, Database Algorithm shows the same performance whether data are horizontally fragmented or not (the line of *DatabaseNoFragment* is under the *DatabaseHorizontal* line in the graph) . This happens because as we said the subqueries are executed in parallel. So, when the network is not congested the overall time of the initial Query to be executed is the time for the slowest query to be executed, that overlaps the time to query and fetch the zero data into our central database. When communication links are highly congested, of course, using Horizontal Fragmentation rules achieves a better performance.

6.2.2.3 Hybrid fragmentation

In hybrid fragmentation except from defining horizontal fragmentation rules, we defined vertical fragmentation rules too and we fragmented a table across two databases. Then a Query that exploited the fragmentation rules was issued. The results are shown in the following graph.

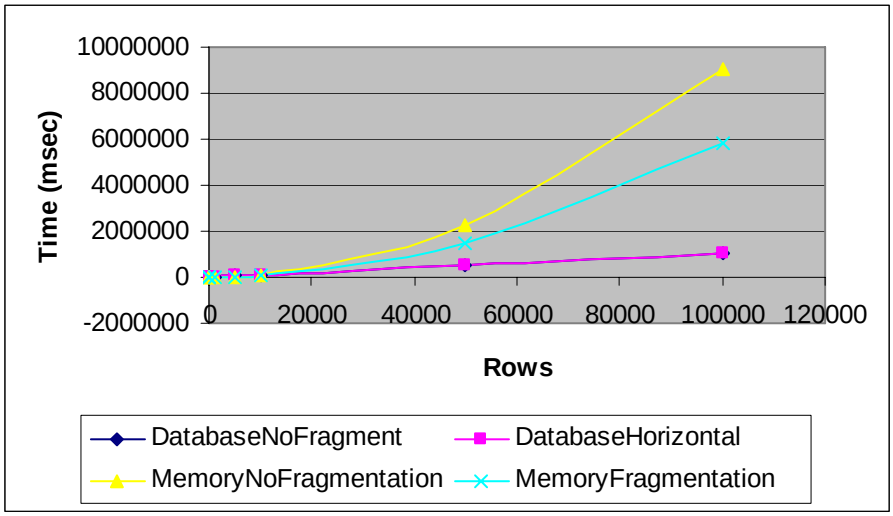


Figure 22. Database vs Memory Algorithm with Hybrid Fragmentation

As we can see in each case, the Memory algorithm is faster for a small number of tuples whereas Database algorithm is better when we have a lot of rows to join across databases. Moreover, as fragmentation knowledge exists our system can use that knowledge to achieve a better performance. We notice that the graph is similar to the Horizontal case one. This is because vertical fragmentation rules are only used to assure that the correct joins are applied, whereas horizontal fragmentation boosts the whole system performance. Of course, if a large table is vertical fragmented in two tables and a query concerning only the data of the one sub-table is issued, the cost is smaller than querying the whole large table.

The previous experiments show, that Quete has an acceptable performance even when a lot of data are going to be queried. Of course, there are some trade-offs in our system. We sacrifice speed in order to be able to integrate answers from multiple sources and in order to be able to query them using a global reference ontology. Furthermore, we can conclude that when difficult operations with a lot of data are going to be performed, current database systems perform better than our implementation.

Chapter 7

Conclusions

"Everything should be as simple as it is, but not simpler"

-Albert Einstein

Contents

In this chapter we will present the conclusions gained from our research concerning the area of query processing in data integration systems. Then we are going to present the directions for our future work since there are a lot to be done in the area.

7.1 Conclusions

The focus of research in information integration is currently changing. While previous approaches concentrated on the integration of a given set of well-structured

databases, the Internet age is about providing a certain type of information to a user, independently of which information source is used.

Examples of the new type of information services are companies that sell information integrated from autonomous web sites, interfaces that provide researchers with experimental results produced and managed in hundreds of laboratories, and bargain finders that harvest hundreds of data sources to find the cheapest offer for a certain good. In these scenarios, integration is provided by a third party, and the task of integration is to satisfy a source independent information requirement. Underlying data sources remain completely autonomous and may evolve independently over time.

Despite the growing importance of this new wave in information integration, few successful solutions are known that are not ad-hoc, hard-coded "hacks". We believe that this is because of several reasons. Firstly, information integration is difficult. The main source of difficulty is heterogeneity and independent evolution, which both are consequences of autonomy. Moreover, virtual information integration is prone to bad performance. It is inherently inefficient compared to homogenous, monolithic systems because in involves the execution of remote methods or queries, and as a result is almost defenseless to bandwidth limitations. Communication costs that arise between distinct data sources and their unknown availability over time limit the capabilities of integration systems. Complicated structures have to be used and many complex problems arise that can only be partly solved in many cases.

Our system is a typically Local-as-View system and is really flexible in addition/deletion of the local sources that participate in the integration system. Moreover underlying sources can evolve at will without any changes to the global schema. Whereas in LAV system, query processing is a difficult task we managed to build a processor that can easily decompose semantic queries to structured queries that will be answered from the underlying databases. Of course in order to achieve efficiency and good performance we sacrifice complexity and expressiveness and complex rules cannot be declared in our system. Only rules concerning table fragmentation can be declared and used and these optimizations make our system unique.

7.2 Extensions

Our system tries to integrate several underlying databases by providing the user with the capability to transparently query them. Of course, our work does not claim to be complete. There are a lot to be done, since the area of data integration is a large and complex. Some of our future plans are presented in this section.

7.2.1 Implementing more Querying algorithms

First of all, our near future plan is to implement several other join algorithms and to build an optimizer that will decide which method to use based on cost estimates. Those cost estimates could be based on statistics kept, or by other cost functions based on predefined knowledge. By obtaining information about the data sources including selectivity and relation size, the global join strategy could be optimized.

Moreover, in many cases nested queries need to be issued which are not currently being supported. Strategies to effectively implement those nested queries should be extensively studied. Except from nested queries, the “Group By” operator needs to be examined in order to be efficiently implemented.

7.2.3 Database Cycles

As we noted before, in many cases schemas may have multiple sets of joins that are equivalent in their semantic meaning. Trying to identify and reduce these duplicate join paths to a single core path will reduce the ambiguity. Heuristics and smart tricks are not always applicable, because user demands may change over time and the administrator cannon always predict the join path desired by users. So, an algorithm should be implemented that will be capable of finding the best join path in each case.

7.2.2 Non – Relational Data Sources

Whereas relational data sources are the most common ones, the evolution of the internet and the web has brought forth opportunities to connect information sources across all types of boundaries. Examples of such information sources include XML and RDF databases, multimedia and object databases.

A major challenge is to extend our implementation in order to integrate such sources with the relational ones. Uniformly querying those sources should add new dimensions to the query planning and execution across those data sources.

7.2.2 Exploiting Systems for Automatically Schema Matching.

In our system, the mapping between ontology and relational data sources is performed by each database administrator, and it is stored in a XML file, called X-Spec. Mapping ontology terms into data sources, is in many cases really time consuming and requires a good knowledge of the underlying schema.

Extending our approach, we could replace the administrator with a tool that would automatically generate the mappings between ontology and schema and would store them in a pre-defined structure. Several algorithms and tools exist [Aumueller et al. 2005], [Bernstein et al. 2004] for that purpose, which perform rather well in most cases and that could be done fully or semi automatic. The predefined structure that stores information about underlying schemata could be XML, or even tuples stored in our lightweight database under a specific schema. We believe that it is really trivial to port one of those systems in our approach, so that human evolvement in the configuration phases can be highly reduced.

7.2.2 The Web Service approach – Grid approach

The system we implemented builds subqueries that are being executed in the underlying data sources and pushes to them all operations, concerning only their distinct schema. Then all operations concerning the final results (ordering, join across databases, etc.) are being executed in our central site and the whole system is implemented in a JDBC driver.

A better approach would be to access data through a web service interface and to distribute the work done in central site, in several other sites according to specific parameters, building something like a grid. This ability to access the data stored in the several relational databases transparently, with mechanisms that will distribute the load, is likely to be a very powerful one, especially for scientists wishing to collate and analyze data distributed over the grid. The first steps in this direction have already started to emerge and several good implementations exist with one of them to distinguish, because it uses the same starting point with us [Arshad A. et al. 2005].

7.2.3 Caching Data

Furthermore, since the results of each subquery are stored in our local lightweight database, it is possible for frequent subqueries, all the information needed to be stored and results to be returned without even querying underlying data sources. Caching could really boost the whole system performance since communication costs will be omitted in many cases.

Of course, if some data are cached during the query processing it is essential to detect whether the query can be answered with the data stored in the cache. Furthermore, the cache replacement policy is really important since data can be invalid after a short period of time. Moreover, calculating missing data and getting them from underlying sources is another aspect of caching.

7.2.4 Updating underlying data sources.

Finally, future work also involves expanding the query processor to handle updates. Several constraints have to be met in the underlying data sources, in order to execute updates in the heterogeneous underlying data sources. The implementation of our system makes it ideal for updating sources too since we have a mechanism that can be easily extended to support updating. In the current state of the system, all the information needed to produce correct and efficient updates exist, since we know the mappings from ontology terms to local fields, the structure and the requirements of the underlying schemata that are captured in the X-Spec files. Update declarations

expressed using our ontology can be decomposed into data source specific update operations the same way queries are decomposed into sub queries issued to local databases.

Bibliography

ABITEBOUL, S., BUNEMAN, P., AND SUCIU, D. 1999. *"Data on the Web, from Relations to Semistructured Data and XML"*. MORKAU, MKADDR.

ACM Computing Surveys. 1990. Special issue on heterogeneous databases. ACM Computing Surveys, 22, 13.

ADALI, S., CANDAN, K., PAPAKONSTANTINOU, Y., AND SUBRAHMANIAN, V. S. 1996. "*Query caching and optimization in distributed mediator systems."* In Proceedings of the ACM SIGMOD Conference on Management of Data (Montreal, Canada, June), 137–148.

AHO A.V., BEERI C., ULMAN J.D., 1979, *"The theory of joins in relational databases",* ACM Transactions on Database Systems, 4(3):297-314

ANTONIOU G., HARMELEN F. V., 2004, *" A semantic Web Primer",* ISBN 0-262-01210-3

APERS, P. 1988. *"Data allocation in distributed DBMS."* ACM Transactions on Database Systems 13, 3 (Sept.), 263–304.

ARSHAD ALI, ANJUM ASHIQ, AZIM TAHIR, BUNN JULIAN, IQBAL SAIMA, MCCLATCHEY R., NEWMAN H., SHAH S.YOUSHAF, SOLOMONIDES TONY, STEENBERG C., THOMAS M., LINGEN F., WILLERS I., 2005, " *Heterogeneous Relational Databases for a Grid-enabled Analysis Environment",* Workshop on Web and Grid Services for Scientific Data Analysis at the Int Conf on Parallel Processing

AUMUELLER D., DO H.H, MASSMANN S., RAHM E., 2005, *"Schema and Ontology Matching with COMA++",* SIGMOD, Baltimore

BABB, E. 1979. *"Implementing a relational database by means of specialized hardware."* ACM Transactions on Database Systems 4, 1 (March), 1–29.

BAKER P., BRASS A., BECHHOFER S., GOBLE C., PATON N., STEVENS R., 1998, *"TAMBIS: Transparent Access to Multiple Bioinformatics Information Sources"* , In proceedings of the Sixth International Conference on Intelligent Systems for Molecular Biology

BARU C., GUPTA A., LUDASHER B., MARCIANO R., PAPAKONSTANTINOU Y.,VELIKHOV P, CHU V., 1999, *"XML-based information Mediation with MIX".*

In proceedings of the ACM SIGMOD International Conference on Management of Data, p 597-599

BATINI C., LENZERINI M., NAVATHE S.,1986 *"A comparative Analysis of Methodologies for Database Schema Integration"*, ACM Computing Surveys 18, 323-364

BELLO, R. G., DIAS, K., DOWNING, A., JR., J. F., NORCOTT, W. D., SUN, H., WITKOWSKI, A., AND ZIAUDDIN, M. 1998. *"Materialized views in oracle."* In Proceedings of the Conferencce on Very Large Data Bases (VLDB) (New York, Aug.), 659–664.

BEN MILED Z., LI N., BAUMGARTNER M. LIU Y., 2003 *"A Decentralized Approach to the Integration of Life Science Web Databases",* Informatica. 27(1)

BERNSTEIN, P., GOODMAN, N., WONG, E., REEVE, C., AND ROTHNIE, J. 1981. *"Query processing in a system for distributed databases"* (SDD-1). ACM Transactions on Database Systems 6, 4 (Dec.), 602–625.

BERNSTEIN, P., MELKIN S., PETROPOULOS M., QUIX C., 2004, *"Industrial strength Schema Matching",* SIGMOD Record, 33(4), 38-43

BESTAVROS, A. AND CUNHA, C. 1996. *"Server-initiated document dissemination for the WWW."* IEEE Data Engeneering Bulletin 19, 3 (Sept.), 3– 11.

BOGLE, P. AND LISKOV, B. 1994. *"Reducing cross domain call overhead using batched futures."* In Proceedings of the ACM Conference on Object- Oriented Programming Systems and Languages (OOPSLA) (Portland, OR, Oct.), 341–354.

BUCK-EMDEN, R. AND GALIMOW, J. 1996. *"SAP R/3 System, A Client/Server Technology."* Addison- Wesley, Reading, MA.

BROSDA V., VOSSEN G,, 1988, *"Update and Retrieval in a relational database through a universal schema interface",* ACM Transactions on Database Systems, 13(4):449-485

BUTTLER D., COLEMAN M., CRITCHLOW T., FILETO R., HAN WEI, LIU LING, PU CALTON, ROCCO D., XIONG LI, 2002 *"Querying Multiple Bioinformatics Data Sources: Can Semantic Web Research Help?",* ACM Sigmod Record, 31(4)

CAREY, M. AND KOSSMANN, D. 1998. "*Reducing the braking distance of an SQL query engine."* In Proceeding of the Conference on Very Large Data Bases (VLDB) (New York, Aug.), 158–169.

CAREY, M., HAAS, L., SCHWARTZ, P., ANYA, M., CODY, W., FAGIN, R., FLICKNER, M., LUNIEWSKI, A.,NIBLACK, W., PETKOVIC, V., THOMAS, J., WILLIAMS, J., AND WIMMERS, E. 1995. "*Towards heterogeneous*

multimedia information systems". In Proceedings of the International Workshop on Research Issues in Data Engineering (March), 124–131.

CAREY, M. AND LU, H. 1986. "*Load balancing in a locally distributed database system."* In Proceedings of the ACM SIGMOD Conference on Management of Data (Washington, DC, June), 108–119.

CERI, S. AND PELAGATTI, G. 1984. *"Distributed Databases—Principles and Systems."* McGraw- Hill Inc., New York, San Francisco,Washington, D.C.

CHAMBERLIN, D., ASTRAHAN, M., KING, W., LORIE, R.,MEHL, J., PRICE, T., SCHKOLNIK, M., SELINGER, P.,SLUTZ, D., WADE, B., AND YOST, R. 1981. *"Support for repetitive transactions and ad hoc queries in System R."* ACM Transactions on Database Systems 6, 1 (March), 70–94.

CHAUDHURI, S. AND GRAVANO, L. 1996. *"Optimizing queries over mulitmedia repositories."* In Proceedings of the ACM SIGMOD Conference on Management of Data (Montreal, Canada, June),91–102.

COHEN W., 1998 *"Integration of heterogeneous databases without common domains using queries based on textual similarity".* SIGMOD Record, 27(2):201-212

COLE, R. AND GRAEFE, G. 1994. *"Optimization of dynamic query evaluation plans."* In Proceedings of the ACM SIGMOD Conference on Management of Data (Minneapolis, MI, May), 150–160.

D'ANDREA, A. AND JANUS, P. 1996. *"UniSQL's nextgeneration object-relational database management system."* ACM SIGMOD Record 25, 3 (Sept.), 70–76.

DATE C. J., 1994, *"The SQL standard"* Addison Wesley, Reading, US, third edition

DAVIDSON S., OVERTON C., BUNEMAN P.,1995, *"Challenges in Integrating Biological Data Sources"* , Journal of Computational Biology. Vol 2, No 4

DAVIDSON S., CRABTREE J., BRUNK B., SCHUG J. TENNEN V., OVERTON C., STOECKERT C., 2001, *" K2/Kleisli and GUS: Experiments in Integrated Access to Genomic Data Sources "* IBM Systems Journal, 40(2), 512-531

DEWITT, D., FUTTERSACK, P., MAIER, D., AND VELEZ, F. 1990. *"A study of three alternative workstation server architectures for object-oriented database systems."* In Proceedings of the Conference onVery Large Data Bases (VLDB) (Brisbane, Australia, Aug.), 107–121.

DESHPANDE, P., RAMASAMY, K., SHUKLA, A., AND NAUGHTON, J. 1998. *"Caching multidimensional queries using chunks."* In Proceedings of the ACM SIGMOD Conference on Management of Data (Seattle, WA, June), 259–270.

DESSLOCH, S., H¨ARDER, T., MATTOS, N., MITSCHANG, B., AND THOMAS, J. 1998. *"KRISYS: Modeling concepts, implementation techniques, and client/server issues."* The VLDB Journal 7, 2 (April), 79–95.

DU, W., KRISHNAMURTHY, R., AND SHAN, M.-C. 1992. *"Query optimization in heterogeneous DBMS."* In Proceedings of the Conference onVery Large Data Bases (VLDB) (Vancouver, Canada, Aug.), 277–291.

DU, W., SHAN, M.-C., AND DAYAL, U. 1995. *"Reducing multidatabase query response time by tree balancing."* In Proceedings of the ACM SIGMOD Conference on Management of Data (San Jose, CA, May), 293–303.

EICKLER, A., KEMPER, A., AND KOSSMANN, D. 1997. *"Finding data in the neighborhood."* In Proceedings of the Conference on Very Large Data Bases (VLDB) (Athens, Greece, Aug.), 336–345.

ENTREZ – Search and Retrieval System . http://www.ncbi.nlm.nih.gov/Entrez

EVRENDILEK, C., DOGAC, A., NURAL, S., AND OZCAN, F. 1997. *"Multidatabase query optimization."* Distributed and Parallel Databases 5, 1 (Jan.), 77–114.

FAGIN, R. 1996. "*Combining fuzzy information from multiple systems."* In Proceedings of the ACM SIGMOD/SIGACT Conference on Principle of Database Systems (PODS) (Montreal, Canada, June), 216–226.

FLORESCU, D., KOSSMANN, D., AND MANOLESCU, I. 2000. *"Integrating keyword search into XML query processing"*, In Proceedings of the WWW Conference (WWW9) (Amsterdam, The Netherlands, May).

FLORESCU, D., LEVY, A., AND MENDELZON, A. 1998. *"Database techniques on the worldwide web: A survey."* ACM SIGMOD Record 27, 3 (Sept.), 59–74.

FRANKLIN, M., CAREY, M., AND LIVNY, M. 1993. *"Local disk caching for client-server database systems."* In Proceedings of the Conference on Very Large Data Bases (VLDB) (Dublin, Ireland, Aug.), 543–554.

FRANKLIN, M., J´ONSSON, B., AND KOSSMANN, D. 1996. *"Performance tradeoffs for client-server query processing."* In Proceedings of the ACM SIGMOD Conference on Management of Data (Montreal, Canada, June), 149–160.

FRIEDMAN M., LEVY A., MILLSTEIN T., 1999, "*Navigational Plans For Data Integration ",* In proceedings of the National Conference on Artificial Intelligence (AAAI), 67-73

GARDARIN, G., GRUSER, J.-R., AND TANG, Z.-H. 1996. *"Cost-based selection of path expression processing algorithms in object-oriented databases."* In Proceedings of the Conference on Very Large Data Bases (VLDB) (Bombay, India, Sept.), 390– 401.

GINGRAS F., LAKSHMANAN L., SUBRAMANIAN I., PAPOULIS D., SHIRI N., May 1997, *"Language for multidatabase interoperability"*. In Proceedings of the

ACM SIGMOD International Conference on Management of Data, vol 26,2 of SIGMOD record, 536-538

GRAEFE, G. 1993. *"Query evaluation techniques for large databases."* ACM Computing Surveys 25, 2 (June), 73–170.

GRAEFE, G. 1995. *"The cascades framework for query optimization."* IEEE Data Engeneering Bulletin 18, 3 (Sept.), 19–29.

GRAEFE, G. 1996. *"Iterators, schedulers, and distributed-memory parallelism."* Software Practice and Experience 26, 4 (April), 427–452.

GRAEFE, G. ANDDEWITT,D. 1987. *"The EXODUS optimizer generator."* In Proceedings of the ACMSIGMOD Conference on Management of Data (San Francisco, CA, May), 160–172.

GRAEFE, G. AND MCKENNA, W. 1993. *"The Volcano optimizer generator: Extensibility and efficient search."* In Proceedings of the IEEE Conference on Data Engineering (Vienna, Austria, April), 209–218.

GRAEFE, G. AND WARD, K. 1989. *"Dynamic query evaluation plans."* In Proceedings of the ACM SIGMOD Conference on Management of Data (Portland, OR, May), 358–366.

GRAVANO, L., CHANG, C.-C., GARCIA-MOLINA, H., AND PAEPCKE, A. 1997. *"STARTS: stanford proposal for internet meta-searching."* In Proceedings of the ACM SIGMOD Conference on Management of Data (Tucson, AZ, May), 207–218.

GRAVANO, L. AND GARCIA-MOLINA, H. 1997. *"Merging ranks from heterogeneous internet sources."* In Proceedings of the Conference on Very Large Data Bases (VLDB) (Athens, Greece, Aug.), 196–205.

GUPTA, A., HARINARAYAN, V., AND RAJARAMAN, A. 1997. *"Virtual data technology."* ACM SIGMOD Record 26, 4 (Dec.), 57–61.

GUPTA, A.,LUDASCHER B., MARTONE M.E. 2000, *"Knowledge Based integration of Neuroscience Data Sources."* , In Intl. Conference on Scientific and Statistical Database Management.

HAAS, L., KOSSMANN, D., WIMMERS, E., AND YANG, J.1997. *"Optimizing queries across diverse data sources."* In Proceedings of the Conference on Very Large Data Bases (VLDB) (Athens, Greece, Aug.), 276–285.

HAGMANN, R. AND FERRARI, D. 1986. *"Performance analysis of several back-end database architectures."* ACM Transactions on Database Systems 11, 1 (March), 1–26.

HAMMER J., SCHNEIDER M., 2003 *"Genomics Algebra: A new Integrating Data Model, Language, and Tool Processing and Querying Genomic Information"*, In proceedings of the 2003 CIDR conference.

HASAN, W. AND MOTWANI, R. 1995. "*Coloring away communication in parallel query optimization.*" In Proceedings of the Conference on Very Large Data Bases (VLDB) (Z ¨ urich, Switzerland, Sept.), 239–250.

IOANNIDIS, Y., NG, R., SHIM, K., AND SELLIS, T.1992. *"Parametric query optimization."* In Proceedings of the Conference on Very Large Data Bases (VLDB) (Vancouver, Canada, Aug.), 103–114.

IVES, Z., FLORESCU, D., FRIEDMAN, M., LEVY, A., AND WELD, D. 1999. *"An adaptive query execution engine for data integration."* In Proceedings of the ACM SIGMOD Conference on Management of Data (Philadelphia, PA, USA, June), 299–310.

JENQ, B., WOELK, D., KIM, W., AND LEE, W. 1990.*"Query processing in distributed ORION."* In Proceedings of the International Conference on Extending Database Technology (EDBT) (Venice, Italy, March), 169–187.504.

KIM, W., GARZA, J., BALLOU, N., AND WOELK, D. 1990. *"Architecture of the ORION next-generation database system."* IEEE Transactions on Knowledge and Data Engineering 2, 1 (March), 109–124.

KONOPKI D. AND SHMUELI O., 1998, *"Information gathering in the World-Wide Web: The W3QL query language and the W3QS system",* ACM Transactions on Database Systems, 23(4):369-410

KORTH H, JUPER G., FEIGENBAUM J, GELDER A, ULMAN J, 1984, *" Sustem/U: A database system based on the universal relation assumption",* ACM Transactions on Database Systems, 9(3):331-347

KOSSMANN, D., FRANKLIN, M., AND DRASCH, G. 2000. *"Cache Investment: Integrating query optimization and dynamic data placement."* ACM Trans. Data Syst.

KUHN E., TSCHERNKO T., SCHWARZ K, 1994, *"A language based multidatabase system",* SIGMOD Record, 23(2):509

HAAS, L., FREYTAG, J. C., LOHMAN, G., AND PIRAHESH, H. 1989. *"Extensible query processing in starburst."* In Proceedings of the ACM SIGMOD Conference on Management of Data (Portland, OR, USA, May), 377–388.

HAAS, L., KODALI P., RICE J.E., SCHWARZ P., SWOPE W.C. 2000, *"Integrating Life Sciences Data – With a Little Garlic."* IEEE International Symposium on Bio-Informatics and Biomedical Engineering

HAAS, L., SCHWARZ P., KODALI P., KOTLER E., RICE J.E., SWOPE W.C. 2001, *"DiscoveryLink: A system for Integrated Access to Life Sciences Data Sources",* IBM Systems Journal, 40(2), 489-511

KABRA, N. AND DEWITT, D. 1998. *"Efficient mid-query re-optimization for sub-optimal query execution plans."* In Proceedings of the ACMSIGMOD Conference on Management of Data (Seattle, WA, June), 106–117.

KOSSMANN, D. AND STOCKER, K. 2000. *"Iterative dynamic programming: A new class of query optimization algorithms."* ACM Transactions on Database Systems 25, 1 (March).

KRISNAMURTY R., LITWIN W., KENT W., June 1991, *"Language features for interoperability of databases with semantic discrepancies",* SIGMOD record, 20(2), 40-49

LEE J.O, BAIK D.K, 1999, *" SemSQL: A semantic query language for multidatabase systems.",* In proceedings of the 8th International Conference on Information Knowledge Management CIKM'99, 259-266

LEVY, A. 1999. *"Answering Queries Using Views: A Survey."* In preparation.

LEVY, A., RAJARAMAN, A., AND ORDILLE, J. 1996. *"Querying heterogeneous information sources using source descriptions."* In Proceedings of the Conference on Very Large Data Bases (VLDB) (Bombay, India, Sept.), 251–262.

LITWIN W. ABDELLATIF A., May 1987, *"An overiview of the database manipulation language MDSL",* In Proceedings of the IEEE. 69-73

LOPEZ R., 2001, "*SRS – Sequence Retrieval System "* .Presentation http://www.pdg.cnb.uam.es/cursos/BioInfo2001/pages/-SRS/, Universidad Autonoma de Madrid

LU, H. AND CAREY, M. 1985. *"Some experimental results on distributed join algorithms in a local network."* In Proceedings of the Conference on Very Large Data Bases (VLDB) (Stockholm, Sweden), 229–304.

LUDASCHER B., GUPTA, A., MARTONE M.E. 2001, *"Model – Based Mediation with Domain Maps",* 17th Intl. Conference on Data Engineering

MACKERT, L. AND LOHMAN, G. 1986. *"R* optimizer validation and performance evaluation for distributed queries."* In Proceedings of the Conference on Very Large Data Bases (VLDB) (Kyoto, Japan), 149–159.

MAIER D., VARDI M., ULMAN J. D., 1994, *"On the foundations of the universal relation model",* ACM Transactions on Information Systems, 12(4):339-359

MASON T., LAWRENCE M., 2005 *"Dynamic Database Integration in a JDBC Driver",* 7th International Conference on Enterprise Information Systems - Databases and Information Systems Integration Track, Miami, FL

MASON T., WANG LIXIN, LAWRENCE R., 2005, *" Autojoin: Providing freedom from Specifying Joins",* 7th International Conference on Enterprise Information Systems - Human-Computer Interaction Track

MCHUGH, J. AND WIDOM, J. 1999. "*Query optimization for XML*." In Proceedings of the Conference on Very Large Data Bases (VLDB) (Edinburgh,GB, Sept.), 315–326.

MERZ U., KING R. , October 1994, *"DIRECT: A Query Facility for Multiple Databases" ,*ACM Transactions on Information Systems, 12(4):339-359

MILLER G.A, BECKWITH R., FELLBAUM C., GROSS D., MILLER K., *"Five papers on WordNet.",* Technical Report CSL Report 43, Cognitive Systems Laboratory, Princeton University 1990

MISHRA, P. AND EICH, M. 1992. *"Join processing in relational databases."* ACM Computing Surveys 24, 1 (March), 63–113

MELTON, J. AND SIMON, A. 1993. *"Understanding the New SQL:A Complete Guide."* Morgan Kaufmann Publishers, San Mateo, CA.

MORK P., HALEVY A., TARCZY-HORNOCH A. 2001, *" A model for Data Integration Systems of Biomedical Data Applied to Online Genetic Databases",* In proceedings of the Symposium of the American Medical Informatics Association

MOTRO A., YUAN Q., 1990,*"Querying database knowledge",* SIGMOD Record, 19(2): 173-183

O'TOOLE, J. AND SHRIRA, L. 1994. *"Opportunisic Log:Efficient Reads in a Reliable Object Server."* Technical Report MIT/LCS-TM-506 (March), Massachusetts Institute of Technology, Cambridge, MA 02139.

OGDEN W., BROOKS S., 1983,*"Query languages for the casual user: Exploring the ground between formal and natural languages",* In Proc. Annual Meeting of the Computer Human Interaction of the ACM, 161- 226

OZCAN, F., NURAL, S., KOKSAL, P., EVRENDILEK, C., AND DOGAC, A. 1997. *"Dynamic query optimization in multidatabases."* IEEE Data Engineering Bulletin 20, 3 (Sept.), 38–45.

OZSU, T. AND VALDURIEZ, P. 1999. *"Principles of Distributed Database Systems (second ed.)"*. Prentice Hall, Englewood Cliffs, NJ.

PAPAKONSTANTINOU, Y., GARCIA-MOLINA, H., AND WIDOM, J. 1995a. *"Object exchange across heterogeneous information sources."* In Proceedings of the IEEE Conference on Data Engineering (Taipeh,Taiwan, 1995), 251–260.

PAPAKONSTANTINOU, Y., GUPTA, A., GARCIA-MOLINA, H.,AND ULLMAN, J. 1995b. *"A query translation scheme for rapid implementation of wrappers."* In Proceedings of the Conference on Deductive and Object-Oriented Databases (DOOD) (Dec.),161–186.

PAPAKONSTANTINOU, Y., GUPTA, A., AND HAAS, L. 1996. *"Capabilities-based query rewriting in mediator systems."* In Proceedings of the International IEEE Conference on Parallel and Distributed Information Systems (Miami Beach, FL, Dec.).

PATON N, STEVENS R., BAKER P., GOBLE C., BECHHOFER S., BRASS A. 1999*"Query Processing in the TAMBIS Bioinformatics Source Integration System"*, In proceedings of SSDBM, 138-147, IEEE press

PIRAHESH, H., HELLERSTEIN, J., AND HASAN, W. 1992. *"Extensible/rule based query rewrite optimization in starburst."* In Proceedings of theACMSIGMOD Conference on Management of Data (San Diego, CA, June), 39–48.

POTAMIAS G., ANALYTI A., KAFETZOPOULOS D., KAFOUSI M, MARGARITHS T., PLEXOUSAKIS D., POIRAZI P., RECZKO M., TOLLIS I.G, SANIDAS M.E, STATHOPOULOS E., TSIKANKIS, VASSILAROS S. I, 2005, *"Breast Cancer and Biomedical Informatics: The PrognoChip Project",* Proceedings of the 17th IMACS world Congress Scientific Computation, Applied Mathematics and Simulation, Paris, France

QUASS, D. AND WIDOM, J. 1997. *"On-line warehouse view maintenance."* In Proceedings of the ACM SIGMOD Conference on Management of Data (Tucson, AZ, May), 393–404.

ROTH, M. T., OZCAN, F., AND HAAS, L. 1999. *"Cost models DO matter: Providing cost information for diverse data sources in a federated system."* In Proceedings of the Conference on Very Large Data Bases (VLDB) (Edinburgh,GB, Sept.), 599–610.

SAGIV Y. 1983, *"A characterization of globally consistent databases and their correct access paths.",* ACM Transactions on Database Systems, 8(2):266-286

SHAN, M.-C., AHMED, R., DAVIS, J., DU, W., AND KENT, W. 1994. Pegasus: A heterogeneous information management system. InW.KIMED., Modern Database Systems, Chapter 32. Reading, MA. ACM Press (Addison-Wesley publishers).

SHETH A.P., LARSON J.A. 1990, *"Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases",* ACM Computing Surveys, 22(3) 183-236

SIDELL, J., AOKI, P., BARR, S., SAH, A., STAELIN, C., STONEBRAKER, M., AND YU, A. 1996. *"Data replication in Mariposa."* In Proceedings IEEE Conference on Data Engineering (New Orleans, LA, Feb.), 485–494.

STEINBRUNN, M., MOERKOTTE, G., AND KEMPER, A. 1997. *"Heuristic and randomized optimization for the join ordering problem."* The VLDB Journal 6, 3 (Aug.), 191–208.

SRINIVANSAN, V. AND CAREY, M. 1992. *"Compensation based on-line query processing."* In Proceedings of the ACM SIGMOD Conference on Management of Data (San Diego, CA, June), 331–340.

STONEBRAKER, M. 1985. "*The design and implementation of distributed INGRES."* Reading, MA.
Addison-Wesley.

SUJANSKY W. 2001, *"Heterogeneous Database integration in Biomedecine Methological Review*", Journal of Biomedical Informatics, 34, 285-298

TANENBAUM, A. 1992. "*Modern Operating Systems*." Prentice Hall, Englewood Cliffs, NJ.

THOMAS, J., GERBES, T., H¨ARDER, T., AND MITSCHANG, B. 1995. *"Implementing dynamic code assembly for client-based query processing."* In Proceedings of the International Symposium for Advanced Applications, (DASFAA) (Singapore, April), 264–272.

TOMASIC, A., RASCHID, L., AND VALDURIEZ, P. 1998. *"Scaling acccess to distributed heterogeneous data sources with DISCO."* IEEE Transactions on Knowledge and Data Engineering 10, 5 (Oct.), 808–823.

URHAN, T. AND FRANKLIN, M. 1999. *"Xjoin: Getting Fast Answers from Slow and Bursty Networks."* Technical report CS-TR-3994 (Feb.), University of Maryland, College Park.

URHAN, T., FRANKLIN, M., AND AMSALEG, L. 1998. *"Cost based query scrambling for initial delays."* In Proceedings of the ACM SIGMOD Conference on Management of Data (Seattle, WA, June), 130–141.

VALDURIEZ, P. AND GARDARIN, G. 1984. *"Join and Semijoin algorithms for a multiprocessor database machine."* ACM Transactions on Database Systems 9, 1 (March), 133–161.

WIDOM, J. 1995. *"Research problems in data warehousing."* In Proceedings of the International Conference on Information and Knowledge Management (Baltimore, MD, Nov.), 25–30.

WIEDERHOLD, G. 1993. *"Intelligent integration of information."* In Proceedings of the ACM SIGMOD Conference on Management of Data (Washington, DC, May), 434–437.

WILLIAMS, R., DANIELS, D., HAAS, L., LAPIS, G., LINDSAY, B., NG, P., BERMARCK, R., SELINGER, P., WALKER, A., WILMS, P., AND YOST, R. 1981. R§: *"An Overview of the Architecture."* IBM Research, San Jose, CA, RJ3325. Reprinted in: M. Stonebraker (ed.), Readings in Database Systems, Morgan Kaufmann Publishers, 1994, 515–536.

WOLFSON, O., JAJODIA, S., AND HUANG, Y. 1997. *"An adaptive data replication algorithm."* ACM Transactions on Database Systems 22, 42 (June),255–314.

ZHU, Q. AND LARSON, P. 1994. *"A query sampling method of estimating local cost parameters in a multidatabase system."* In Proceedings IEEE Conference on Data Engineering (Houston, TX, USA, Feb.), 144–153.

Appendix

List of Symbols and abbreviations.

Symbol	Explanation
$\cup$	The union operator
π (A)	The projection operator
σ (A)	The selection operator
$\triangleright\triangleleft$	The join operator
$\triangleright<$	The Semijoin operator
DBMS	Database Management System
DSN	Data Source Name
CV	Context View
GAV	Global as View
LAV	Local as View
ODBC	Open DataBase Connectivity, standard database access mehtod
SQL	Structured Query Language
UR	Universal Relation

Sample JDBC Application

```
1:      import java.sql.*;
2:
3:      public class JDBCApplication
4:      {
5:              public static void main(String[] args)
6:              {
7:
8:                      String url = "jdbc:QueTe://sources.xml";
9:                      Connection con;
10:
11:                     // Load QueTeDriver class
12:                     try { Class.forName("Quete.jdbc.QueTeDriver"); }
13:                     catch (java.lang.ClassNotFoundException e)
14:                     {System.exit:}
15:
16:                     try { // Initiate connection
17:                             con = DriverManager.getConnection(url);
18:                             Statement stmt = con.createStatement();
19:
20:                             ResultSet rst = stmt.executeQuery("
```

```
21:                     SELECT Part.Name, LineItem.Quantity, Customer.Name
22:                     WHERE Customer.Name='Customer 25'");
23:
24:                             System.out.println("Part , Quantity, Customer");
25:
26:                             while (rst.next())
27:                             {
28:                                     System.out.println(rst.getString("Part.Name")
29:                                     +","+rst.getString("LineItem.Quantity")
30:                                     +","+rst.getString("Customer.Name"));
31:                             }
32:                             con.close();
33:                     }
34:                     catch (SQLException ex) {System.exit(1); }
35:             }
36:     }
```

Evaluation Measurements

No Fragmentation

Rows			Jdbc-Odbc	QueTe Memory	QueTe Database
Db1	Db2	Db3			
5	5	5	9 + 1 + 97	443	897
10	10	10	3 + 3 + 95	427	921
100	100	100	11 + 7 + 102	410	3017
1000	1000	1000	109 + 47 + 156	6508	24831
5000	5000	5000	505 + 215 + 886	144500	119521
10000	10000	10000	1031+432+654	529895	212513
50000	50000	50000	6032+2218+3920	-	-
100000	100000	100000	11470+4440+21668	-	-
1000	100	100	110+5+46	1248	11237
1000	1000	100	108+61+5	3871	23666
100	1000	1000	12+48+83	5343	22438
100	100	1000	12+26+70	2046	10683
10000	10000	100	1045+430+8	266304	194603
50000	50000	100	5955+2215+12	-	940119

Table 3. Results with when no fragmentation exists

Horizontal Fragmentation

Rows			Jdbc-Odbc	Quete Database Normal	Quete Database Horizont	Quete Mem Normal	Quete Mem Horizont
Db1	Db2	Db3					
5	5	5	1+1+1	724	471	233	223
10	10	10	2+0+0	484	460	220	223
100	100	100	0+2+2	1255	1362	243	240
1000	1000	1000	6+9+21	12555	12629	974	734
5000	5000	5000	27+26+26	56415	56122	26410	15169
10000	10000	10000	55+48+207	109848	111990	88948	59858
50000	50000	50000	245+221+749	508391	516580	2233303	1499815
100000	100000	100000	447+428+1724	1009590	1015382	9005540	5783919

Table 4. Results When Horizontal Fragmentation exists

Vertical Fragmentation

Rows			Jdbc-Odbc	Quete Database	Quete Memory
Db1	Db2	Db3			
5	5	5	1+1	1248	260
10	10	10	2+1	714	226
100	100	100	4+5	2526	320
1000	1000	1000	32+18	23650	3387
5000	5000	5000	68+64	104719	68390
10000	10000	10000	123+124	229074	287871
50000	50000	50000	572+597	1015868	6446310
100000	100000	100000	1095+1192	-	-

Table 5. Results when Vertical Fragmentation exists

Hybrid Fragmentation

Rows			Jdbc-Odbc	Quete Database Normal	Quete Database Horizont	Quete Mem Normal	Quete Mem Horizont
Db1	Db2	Db3					
5	5	5	1+1+1	861	1051	327	250
10	10	10	1+1+1	797	701	237	223
100	100	100	0+3+1	1295	1275	267	263
1000	1000	1000	24+8+2	16377	13487	1054	951
5000	5000	5000	27+26+21	59412	58464	20809	18149
10000	10000	10000	49+45+42	111474	109679	71384	71747
50000	50000	50000	251+209+303	520847	533516	1689826	1684386
100000	100000	100000	458+424+368	1018832	1027887	6691803	6765256

Table 6. Results when Hybrid Fragmentation exists