A Visual Query Language for a Federation of Databases

by

Martin Andreas Jacobs

A thesis
presented to the University of Manitoba
in partial fulfilment of the
requirements for the degree of
Master of Science
in
Computer Science

Winnipeg, Manitoba, Canada, 1996

ISBN 0-612-13208-0

Name _____

Dissertation Abstracts International and *Masters Abstracts International* are arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation or thesis. Enter the corresponding four-digit code in the spaces provided.

Computer Science
SUBJECT TERM

0987
SUBJECT CODE

UMI

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

THE UNIVERSITY OF MANITOBA

FACUTY OF GRADUATE STUDIES

COPYRIGHT PERMISSION

A VISUAL QUERY LANGUAGE FOR A FEDERATION OF DATABASES

BY

MARTIN ANDREAS JACOBS

A Thesis/Practicum submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Abstract

This thesis presents a system for querying a federation of databases easy enough to use so that even people with very limited computer knowledge can produce meaningful queries. To achieve this goal a simple Visual Query Language (VQL) has been designed.

Most VQLs simplify access to database systems by displaying the contents or schema of the underlying databases graphically. The two most common flavours of VQLs, diagrammatic and iconic, will be described, as well as arguments given, why iconic languages are more suitable for novice users. Examples of different VQLs are provided.

Several problems of federated database systems will be addressed. Among those are schema level integration, instance level integration and the need for a common data model. Some solutions to these problems are outlined.

A detailed description of the implemented system is given. The query language designed for this system is implemented in the Hyper Text Markup Language (HTML) which is generated based on user input. The HTML approach combines ease of use with ease of programming.

Acknowledgments

There are many people that I would like to thank. First I would like to thank my supervisor Dr. Ken Barker for having the basic idea for this thesis and being there when it was necessary. I would also like to thank the members of my thesis committee Dr. David Blight, Dr. Mark Giesbrecht, Dr. Peter Graham, and Dr. Ken Barker for making the thesis defense possible at a very early date. Another "Thank You" to Len Dacombe and Dr. Clint Gibler of TR*Labs* for providing me with a great research environment (and a steady supply of doughnuts). Thank you also to Glenda Stark for reading and correcting the thesis. Other people that were helpful in the creation and design of the program described in this thesis are Dr. Ken Ferens, Martin Meier and Darryl Dueck. A miscellaneous thank you to Heather Hnatiuk, Dan Erickson and Shamit Bal. A very big Thank You to my girl friend Christine Crisan for waiting a very long time until the M.Sc. degree was finally achieved. Also to be thanked in general are my parents. Finally a thank you to TR*Labs* as an organization for providing the necessary financial support.

Contents

List of Figures

Chapter 1

Introduction

Ever since the advent of the first commercial computers in the early 1950s computers have been used to process, store and retrieve information. Over time the techniques for storing data evolved and developed. The earliest method of storing information was the use of application specific storage files, which did not have a standard interface, so they could only be accessed by the one special application they were developed for.

The next step in information storage techniques was the introduction of databases. Databases are specialized applications that store information independent of any particular application program. They have a well defined interface through which applications can store or retrieve information. The *network model* was one of the first data models used in databases. It stores the relationships that exist between the information explicitly. This means that only relationships that are predefined can be extracted from the system. The relationships between the data are of a form where

one record type "owns" another record type. For example the record type "student" owns the record type "course", which models a student taking a particular course. The *hierarchical model* is a subset of the network model and emerged at about the same time. In contrast to the network model, it only allows a record type to be owned by exactly one "parent" record type. The most popular data model today is the *relational model*, which was developed next. It stores the information in tables that aggregate related information, rather than as individual records. The main benefit of this data model is that the relationships that exist between the information do not have to be predefined. Through the *Structured Query Language* (SQL) it is possible to associate the information in different tables without explicitly hard coding the relationship in the database. Another benefit is that the access to information is less dependent on the physical storage compared to the older models. The most recent data model is the *object-oriented model*. This model more readily stores and manipulates complex information than previous ones because of its properties of inheritance, polymorphism and encapsulation.

All of these methods of data storage have a common disadvantage, in the way the store and retrieve information. In all cases a special access method or query language has to be utilized. If the database access is embedded in an application program, then the programmer has to know the query language and the structure of the database. If the query is posed directly to the database, in an *ad-hoc* query, then the end user has to have substantial experience. In either case, training in the use of a query language

in general, and the structure of the database under consideration in particular, is necessary. This poses a problem if the information contained in the database is to be made available to the general public, where it cannot be expected that every person trying to gain access to the information has the necessary background to formulate correct queries.

Visual query languages (VQLs) have been developed to remedy this problem. VQLs can be subdivided into two classes: graphical query languages and iconic query languages. Of these two classes, iconic query languages are especially well suited for novice users. Iconic languages allow the user to point-and-click on an icon representing certain information, so more detailed data can be retrieved.

One example of where such a query language is needed is an *information kiosk* in a shopping mall. Information kiosks are interactive displays, that guide the shopper towards a store that carries an item of interest. In contrast to a conventional mall directory, which can only display the location of the stores in the mall, an information kiosk can also display an inventory of the merchandise each individual store has to offer. To effectively use the kiosk access must be easy and intuitive and it must provide up-to-date information on the merchandise available.

Accuracy of the data describing the merchandises' availability and pricing, combined with the stores' reluctance to hand over their data to a central repository, rules out a centralized approach. Instead the information must remain in each individual store's database. This presents the problem that while the information is retained

in different databases it should be accessible to the information kiosk user without undue overhead or the need for substantial user expertise.

This problem can be decomposed into two subproblems. The first problem is that of simplifying access to a database. The second is that of integrating multiple independent databases into one coherent system, while maintaining the independence of the databases. The first problem can be solved with the use of an iconic query language, while the second problem can be overcome by building a federation of databases. Sheth and Larson describe a *federated database management system* as : "A collection of cooperating but autonomous component database systems".

This thesis describes the Community Data Network Architecture (CDNA), which is an approach to supporting user access to multiple data repositories. An example of this approach is an information kiosk system which has been developed where the data is not transferred to a single data repository, but remains at each store's database. The store databases therefore form a federation of databases. The central CDNA system contains only a "directory" of the information available. The user interface is simple, so an average computer-naive shopper having no database knowledge can use it effectively. The user interface is a simple visual query language and the information about the merchandise is structured in a hierarchical fashion, so the user is first presented with a very general selection of product categories. Selecting any of these categories yields more detailed information about the categories. The information about an actual product is retrieved from the store databases.

1.1 Thesis Organization

The thesis is organized in the following manner:

- **Chapter 2** introduces visual query languages. It begins by describing graphical query languages, which are more suitable for the experienced user, since they manipulate the schema of the database. The chapter also describes iconic query languages, which are more useful to inexperienced and naive users, since the concepts contained in the database are represented by familiar small images. Finally some examples of visual query languages are given.

- **Chapter 3** outlines the problems encountered in the construction of a federation of databases. It describes schema level integration as well as instance level integration. It also outlines the need for a common data model to have a common representation of the data present in the federated database system.

- **Chapter 4** describes the implementation of the CDNA system. This includes the development of a quasi iconic query language, and a method of support for the construction of a federation of databases, based on relational databases. The user interface is based on the Hyper Text Markup Language (HTML). This makes the CDNA system portable from one shopping mall like application to another and scalable to a virtual mall on the Internet.

- **Chapter 5** contains concluding remarks and suggestions for future extensions of the CDNA project.

Chapter 2

Visual Query Languages

The trend in operating systems is to use graphical user interfaces (GUIs) because they are more intuitive to both novice and expert users. GUIs present the user with choices in a pictorial form thereby increasing ease of use. Most text based systems on the other hand require users to memorize complex commands that must be carefully entered to avoid errors.

Traditional database query languages operate without a GUI. They are text based and not easily accessible to the uninitiated. An example is SQL which was designed to be a user friendly query language used by end-users to directly access database management systems (DBMS). While it is structured and fairly easy to learn for technical users, it is not intuitive to novice database users. There are many reasons for this. One is the need to memorize the commands of the query language. Another reason is that the structure of the database has to be known in advance. Structural knowledge includes table names and column names in the database and their inter-

relationship between the different tables, including foreign keys. This requirement to know the exact database structure presents a problem for expert users because of the database's complexity, that is further compounded for novice or naive users.

Visual Query Languages (VQL), as described in depth by Batini, *et al.* [BCCL91], try to overcome some of these problems. VQLs can be subdivided into several different classes. One is the class of diagrammatic query languages. These query languages represent the schema, or logical structure, of the database under consideration using diagrams. An example of these kinds of diagrams is the Entity-Relationship (E-R) diagram, as described by McFadden and Hoffer [McHo91].

E-R diagrams are often used to model relational databases. In an E-R diagram database entities are visualized with boxes, while the relationships among entities are shown as diamonds. These are connected through lines that show how the entities and relationships are related to each other.

Other forms of visual query languages include iconic languages. Here, the concepts of interest, are shown in the form of icons, small pictures that capture and symbolize a specific abstract idea or concrete entity.

Both approaches to visual query languages are described in more detail in the following sections. Additional examples are provided for these query languages. Finally, a comparison between SQL and diagrammatic query languages is presented.

2.1 Diagrammatic Query Languages

As with all visual query languages, the main goal of diagrammatic query languages (DQLs) is to make accessing a database easier for the user. In this section, first the visual representation of the database is described, then the mechanisms of query formulation are discussed and finally the suitability of diagrammatic query languages for different classes of users is outlined.

Diagrammatic query languages usually display the database schema using a diagram. This makes complex schemas much more accessible to the user. The visualization is achieved by assigning a limited set of geometrical figures to the different components of the database. As mentioned above, the E-R diagram is a good example of this, where only rectangles, diamonds, lines and circles are used. The number of geometric figures should be limited so that the user is not overwhelmed. Other possibilities for representing the database include the visualization of the actual content of the tables in contrast to the schema of the database.

There are two slightly different representations used to display geometric figures on the screen. One is the grid standard, where the lines, connecting different concepts are allowed to have corners in them. Figure 2.1 is an example of a diagram which conforms to the grid standard. The advantage of this representation is that it is very compact and can fit many concepts in a limited area. The disadvantage is that more concepts make it difficult to differentiate concepts, especially if the lines connecting the concepts have many corners in them. Therefore the straight line standard seems

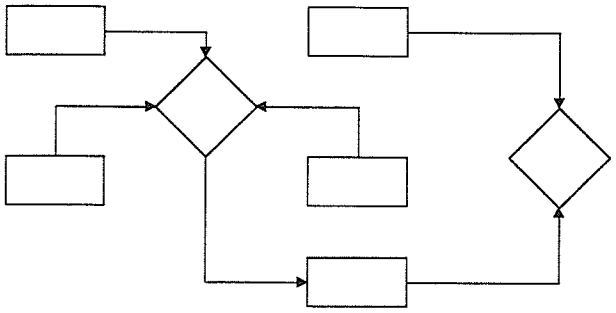

Figure 2.1: A diagram conforming to the grid standard

more appropriate. Figure 2.2 depicts a line standard diagram. Another important

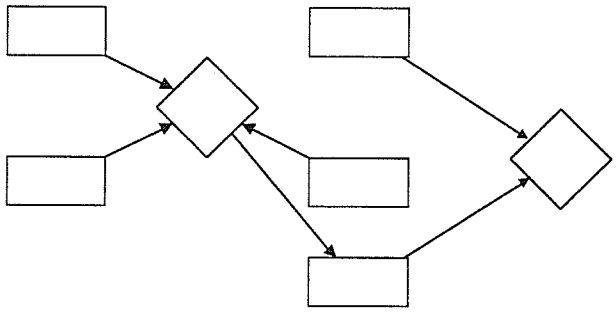

Figure 2.2: A diagram conforming to the line standard

aspect is the reduction of line crossings. It enhances diagram readability so it is easier to distinguish which concepts relate.

Queries are formulated through pointing and clicking with a mouse. A user clicks on a concept of interest to examine it more closely. For example in a relational database a concept can be a table. Query formulation for diagrammatic languages can be thought of as a three step process. The first step is to understand the concept of interest, the second step is the actual query formulation, and finally the query is tested to ensure the results are those the user desires.

2.1.1 Understanding the Concept of Interest

The process of understanding the concept of interest can be done in different ways. One possibility is to take a *top-down* approach. Here, the user selects a high-level concept of interest in the database. After a high-level concept has been chosen, it is further examined. For example, the examination can be based on finding out which columns in the table are of interest. Finally the rows are selected. Each time a selection is made, the process zooms in on the final goal. The top-down approach is really a method of iterative refinement, where, with each iteration, the user gets closer to the final goal of the query.

Another possibility is to use *selective zooming*. Here too the concepts that the user is interested in are zoomed in on, but other concepts remain visible on the screen but minimized. The advantage is that the user is still shown the context in which the query is being formulated, while avoiding overcrowding on the screen.

The *hierarchical zoom* is yet another approach to database querying. Here different concepts of interest can be looked at with varying degrees of zooming incrementally increasing the degree of detail. For example in object-oriented databases, the user might be examining one instance of an object while looking at the type structure of a different class.

An example of the top-down approach is presented in Figures 2.3, 2.4, and 2.5. The query that the user wants to pose on the Database Management System (DBMS) is: "What are the names of the employees working on projects that have a deadline

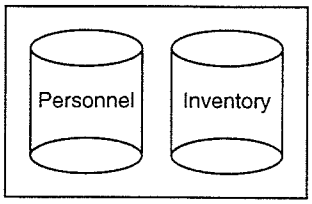

Figure 2.3: Initial screen of a DQL system

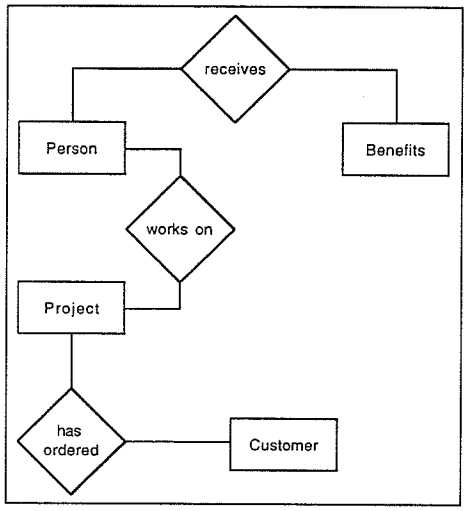

Figure 2.4: The schema of the personnel database in a DQL

of May 15, 1996 ?".

Figure 2.3 shows the initial graphical representation of the DBMS. There are two databases in the DBMS, one dealing with *Personnel* data and the other containing data on the inventory of the company. Since the user is interested in employee data the personnel database is clicked on. The user is then presented with a high level representation of the contents of the *Personnel* database (see Figure 2.4). Since the user is interested in names of employees working on a certain *Project*, this part of the database is selected (see Figure 2.5). This diagram contains the necessary detail

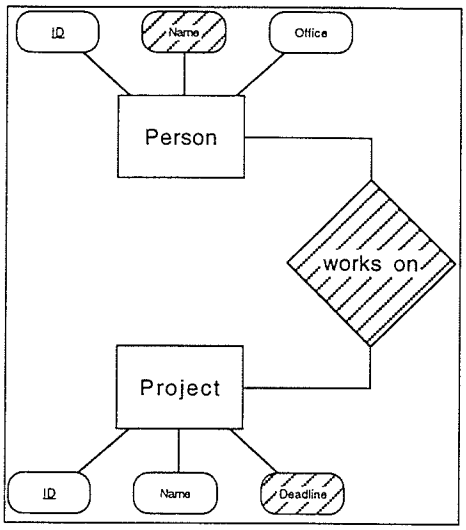

Figure 2.5: Query formulation in DQL system

to pose the query that answers the user's question. To obtain the desired result, the *Deadline* attribute of the project relation is chosen, as well as the relationship between project and person, in addition to the *Name* attribute in the person relation. Finally, the user supplies the system with the deadline of interest.

Another way of finding the concept of interest, instead of the top-down approach outlined above, is to browse the graphical representation of the database. Browsing is appropriate when users have little knowledge about a database and have to familiarize themselves with the concepts and structure of the database. Since they are not familiar with the database, these users usually do not have a predetermined goal, so they have to be provided with a method of exploring the different concepts and their relationships. As with the top-down approach, there are several different strategies for browsing. Intensional browsing on the database schema shows the user how the

different concepts are interrelated. An example from relational databases is showing the user which tables have foreign keys into other tables. Another kind of browsing involves inspecting the actual contents of the database. Here the user can examine the different concepts more closely. Finally, there is the possibility of mixing both kinds of browsing, so the user first browses the schema of the database and then when an interesting concept has been located, examines the content of that concept.

The difference between the two approaches is that in the top-down approach the user has a clear understanding of what type of information is desired, and consequently tries to locate this information immediately. With browsing, the user explores more concepts and their connections to locate the information of interest.

2.1.2 Query Formulation

After the concept of interest has been located, either through the top-down approach or through browsing, the query itself must be formulated. Here again several mechanisms exist.

The first is to specify the query in a top-down fashion. This means that the user first locates the concept of interest and then narrows down to the exact instances of interest. For example, the rows of a table in a relational environment, can be selected by pointing and clicking.

Another possibility is to formulate a query in a bottom-up fashion. The user creates query libraries that form the lowest level of the query. These libraries are

reused whenever the query is issued. In effect the user only has to choose which concepts to pose a query on and let the query library generate the appropriate low level queries. Other strategies for formulating a query are closely related to browsing while finding the concept of interest. The user specifies a concept of interest and uses it as a starting point to relate other concepts to it.

Finally, it is also possible to combine different forms of query formulation. An example would be the combination of the top-down and bottom-up approaches. The user first zooms in on the concept or instances of concepts of interest and then invokes one of the previously created query libraries.

2.1.3 Query Assessment

The final step of the three step process of constructing a query for diagrammatic languages is to determine whether the query generated is correct and does what is desired. This can be done by translating the query, given that a relational database is being queried, into SQL, which is then inspected by the user. This is only useful if the user knows SQL so the system designer would make use of this approach. A more user friendly approach for the testing stage is to translate the query into natural language. Here the query system reformulates the diagram into an English sentence stating the names of the tables and columns involved.

Diagrammatic query languages are more directed at the expert user. The reason for this is that diagrammatic query languages work on the database schema predom-

inantly. Thus this class of visual query languages is not suitable for the Community Data Network (CDNA) project.

2.2 Iconic Query Languages

As with diagrammatic languages, iconic query languages (IQL) are designed to simplify the database querying process. Many similarities exist between the two visual querying strategies. The differences that exist between the strategies will be described in this section.

Iconic query languages, in contrast to diagrammatic query languages, usually do not display the database schema, but rather the actual data values. The icons are visual symbols of the entity or idea they represent. An example would be an icon depicting a person, which represents a personnel record in a company's database. The user clicks on this icon and is provided with additional icons, representing different levels of employees and the actions that can be performed on their records.

Understanding the concept of interest in iconic languages is mostly achieved through browsing. Here browsing does not mean browsing the schema but browsing the available data. One data item might lead to another data item. In the example with the personnel records, the record of a specific employee might lead to another record representing the employee's spouse. This record in turn might lead to yet another record representing their child, for example. In this fashion, it is possible to give the user suggestions for a non-goal directed search when the user is not sure

what is needed.

Queries in iconic languages are usually formulated by associating the icon of the concept of interest with an icon representing an action that is supposed to be carried out on the concept of interest. By way of example, a personnel record might be selected and then an icon representing "delete" might be selected. This permits the employee record to be deleted from the database. The icon for the action "delete" might be depicted by a trash can or a shredder which are natural representations of similar activities in the user's physical environment.

The main difficulty with iconic query languages is finding icons that are generally understandable. It is very difficult to find iconic representations that carry the same meaning for all users without further explanation. Even for concrete objects it might be difficult to find a self-explanatory icon. The reason is that the users may not be familiar with the object represented by the icon. For example, the icon can have a clear visual representation of a Manila folder on it, but it will not be understandable to a user who is not familiar with this way of organizing sheets of paper.

The first principle for creating meaningful icons is to draw ideas from an environment surrounding the intended user. For a database in an office environment this might be office tools. Thus a shredder can be used to represent deleting instances from the database, a Manila folder can represent the aggregation of certain instances of the database, and so forth.

Other ways of making the icons more meaningful include the use of special colours

and/or highlighting parts of the icon. This way the user's attention can be directed towards certain icons. They can be used, for example, to inform the user that the status of a query has changed from *processing* to *completed.*

Finally, by combining icons with descriptive text, they become much more understandable. Thus, the user has a character based reminder of what the icon stands for. The disadvantage of this approach is that, the user must be at least rudimentarily literate. This can pose a problem in certain circumstances, as for example in a shopping mall environment, where all prospective customers must be able to use the system. Also, once text is introduced to the system it becomes more culturally dependent than with icons. If the system is exported to a different country the text describing the icons has to be translated.

Iconic languages are aimed at a different class of users than diagrammatic query languages. Iconic languages are more suited to the casual user who does not have any training in using databases. For these users the language and its functionality have to be immediately obvious. Since these users usually do not specify very complex queries, it is acceptable that iconic languages sacrifice some expressiveness for ease of use.

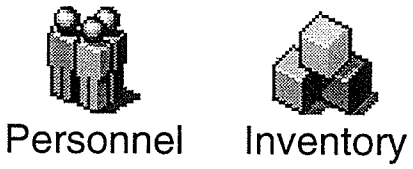

Figure 2.6: Initial screen of an IQL system

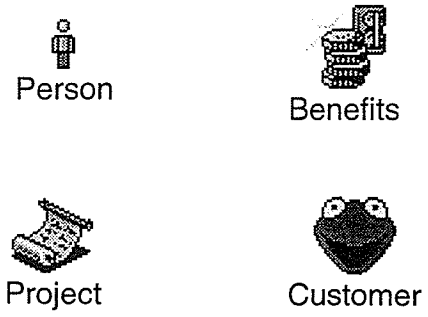

Figure 2.7: Contents of the personnel database

An example of an iconic query is shown in Figures 2.6, 2.7 and 2.8. Both the database and the query are the same as with the example presented for diagrammatic query languages. The query is again: "What are the names of the employees working on projects that have a deadline of May 15, 1996 ?".

The user is first presented with the content of the DBMS. Figure 2.6 shows that there are two databases to choose from. Since the user is interested in data on certain employees, the *Personnel* icon will be chosen. After the user clicks on the icon representing *Personnel*, the content of this database will be displayed. There are four tables in this database, a table on people in the organization, a table on the benefits these people receive, a table on the projects that the employees work on, and finally a table on the project's customers.

The query asks for the the names of employees, so the user will select the *Person* icon, as well as the project icon. Then a text dialog box will appear, in which the user will have to type in the date of the deadline. As well the user has to mark which attributes of the selected tables are of interest. This is shown in Figure 2.8. The

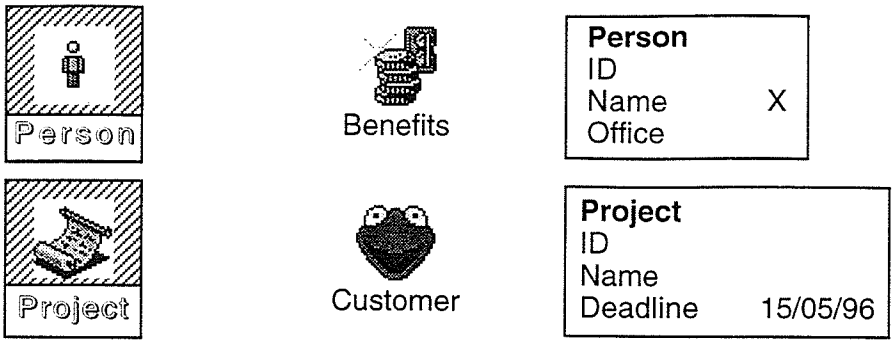

Figure 2.8: Query formulation in an IQL system

result of the query will also be presented in a textual form, stating the names of the employees working on projects whose deadline is on May 15, 1996.

2.3 Examples of Visual Query Languages

In this section several different visual query languages will be briefly described. The first query system to be described is QBD*, which is a diagrammatic query language that can store queries in a query database. The second system is LOOKS, a user interface generator for the object-oriented database system O_2. HQL, a hybrid query language whose main concept is to restrict the visual interaction to more complex queries, is presented, as well as QBIC, a system for querying image databases. Finally the Hyper Text Markup Language (HTML), the language in which World Wide Web pages are written, will be presented. HTML can be used as a special form of an iconic query language. The queries that it allows cannot be as complex as a full featured iconic query language, but it is also easier to use, as fewer choices can be made by

the user.

2.3.1 QBD*

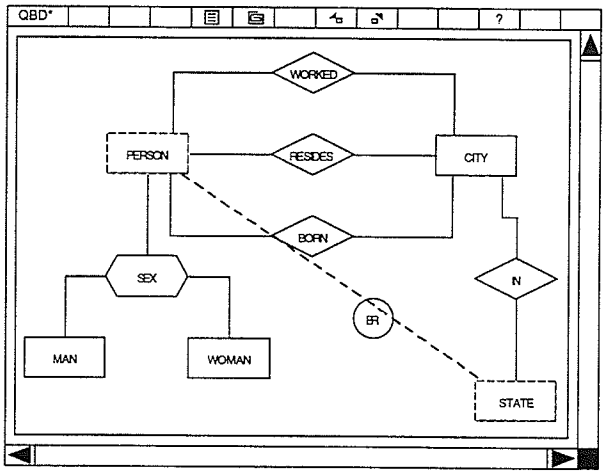

Figure 2.9: Identification of main concept of interest in QBD*[SaSo93]

Query by Diagram (QBD*) described by Santucci and Sottile [SaSo93] is a diagrammatic query system. It is based on a relational database and uses the E-R diagram of the database schema as the database's graphical representation. The system is fully visual so that the need for the user to type in commands on the keyboard is reduced greatly.

The concept of interest is found through top-down browsing as described in Section 2.1. The user zooms in on the part of the schema that contains the data of interest.

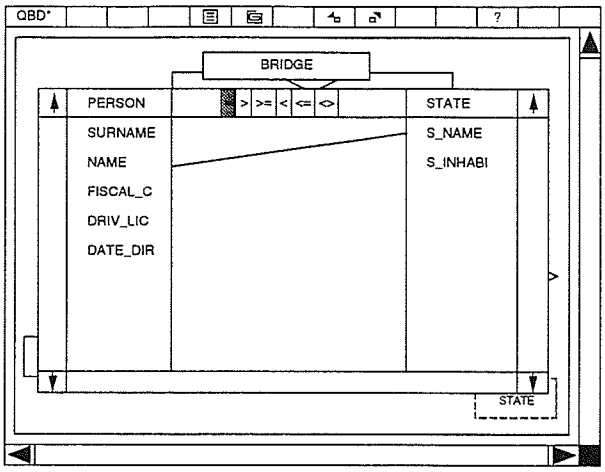

Figure 2.10: Query formulation in QBD*[SaSo93]

Once the concept of interest has been located, it can be further refined so that only particular instances of this concept participate in the query. These particular instances can then be related to other concepts by connecting the concepts through edges. Finally the results of the query specification can be stored in a library so that they can be reused.

Figures 2.9 and 2.10 show an example query in QBD*. After the main concept of interest has been identified, a second concept is selected, and a condition for the query is specified. In Figure 2.9 the selection of the concepts PERSON and STATE is shown. The specification of the condition is shown in Figure 2.10. In SQL, it could be specified as `select * from PERSON, STATE where PERSON.NAME = STATE.S_NAME`.

2.3.2 LOOKS

LOOKS is a user interface generator used to build visual query interfaces for the object-oriented database O_2[O2+90]. The interfaces generated by LOOKS act very much like iconic languages, even though they are not purely iconic. The generated interfaces also make extensive use of character based explanation to make the icons more understandable.

The user finds the concept of interest by browsing the available icons. Each icon represents a certain object in the database. A mouse click on one of the icons reveals more information on the concept represented by that icon. Associated with each icon are the methods that can be invoked on the objects. This defines the results of the mouse clicks.

The query is formulated by associating icons with each other, or through text input when desirable. This might be used when restricting the range of a query to an age group in a personnel database. Other possibilities that LOOKS provides are cut, copy, paste and create operations. For example if, a user wants to enter a new employee into the personnel data structure, an icon for that new employee, will be pasted onto the personnel icon after its creation. These operations are restricted by the methods associated with the icon, so it would be impossible to paste an item belonging to an inventory data structure into a personnel structure.

LOOKS does not provide a specific facility for testing the correctness of the query that was formulated. The user is expected to know whether the results obtained

correspond to the needs.

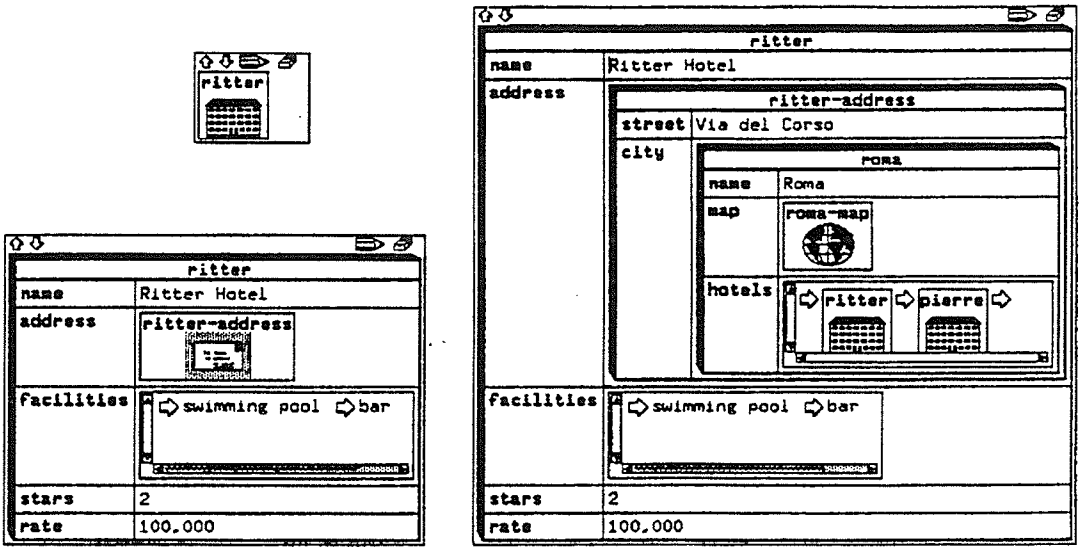

Figure 2.11: Three stages of icon examination in LOOKS [O2+90]

Figure 2.11 illustrates iconic exploration using LOOKS. The icon represents a hotel object that can be examined at three different stages. Each of these stages yields more information about the hotel.

2.3.3 Hybrid Query Language

Andries and Engels [AnEn95] describe the *Hybrid Query Language* (HQL), which addresses query language problems that arise when queries become too complex to express graphically. This directly contradicts the reason for developing visual query languages which is to simplify user interaction with the database.

HQL is a query language which offers diagrammatic and textual formulation of

queries. The capabilities of the textual query language remain but are supplemented with visual querying. For complex queries the user can switch at any point between visual and character based querying. It is not possible to specify very basic operations like aggregate functions graphically so these operations must be typed. The diagrammatic query language used in HQL is based on an extension of the E-R diagram. This extension allows the E-R diagram to contain, among other things, specializations and generalizations.

Figure 2.12 shows an example query in HQL. The user navigates the E-R diagram of the underlying database until a concept of interest is reached. The user selects this concept of interest and copies it to the bottom query window, where the constraints for the query are entered.

2.3.4 QBIC

Query By Image Content (QBIC) is described by Niblavk, *et al.*[NBEF+93]. It is a visual strategy that is totally different from the other strategies outlined in this chapter. While the other strategies are mainly concerned with a visual representation of the underlying database schema (the diagrammatic languages), or with a visual representation of the database instances of the database (the iconic languages); QBIC is a visual query language in the truest sense of the word. QBIC is a query system that allows *image* retrieval in a database based on the colour, shape or texture of that image. QBIC tries to overcome the inadequacies of current large scale image

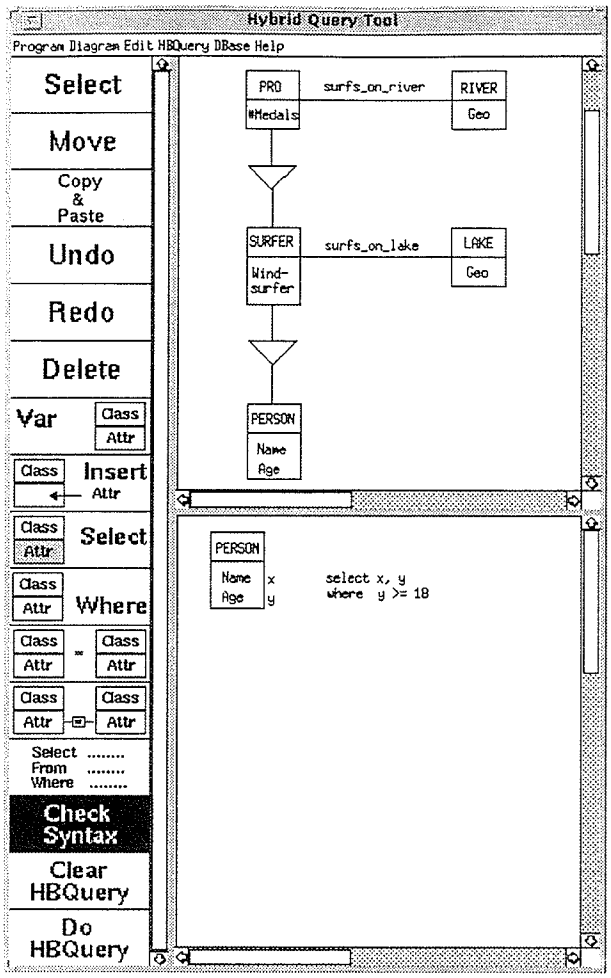

Figure 2.12: Example query in HQL[AnEn95]

databases, where the only way of retrieving an image of interest is based on the textual description of the image that is stored alongside it. For smaller scale image databases the authors recommend the use of thumbnail images for selecting a picture. In contrast to other querying systems, QBIC does not provide a definitive answer, but a range of images that might fit the current query, so the user has the final choice as to which image best fits the query.

The system consists of two parts. The first is the database population part, while the second is concerned with the formulation of a query. During database population the user can highlight certain parts of the image to be subsequently queried. There can be multiple regions highlighted for each image. For each highlighted area, their shape, texture and colour scheme will be analyzed and stored with the image. In the querying part of the system, it is possible to specify shapes, textures and colouring schemes that should be present in the retrieved images. Each of the conditions can be used alone or in conjunction with each other. It is, for example, possible to request an image that contains a certain shape and a certain colour. To specify shapes that should be present in the retrieved images, a user can sketch a shape using a mouse or other input device. A *colour picker* is provided to select the desired colours. Finally, these inputs are matched against the images in the database. They can either be matched against the complete image or against the regions highlighted in the database population part.

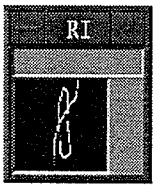

Figure 2.13: Query formulation in QBIC[NBEF+93]

An example of the query by sketch part of the system is shown in Figure 2.13 and 2.14. For this query, a shape that should be present in the desired images is sketched. Figure 2.13 shows a sketch that a user entered.

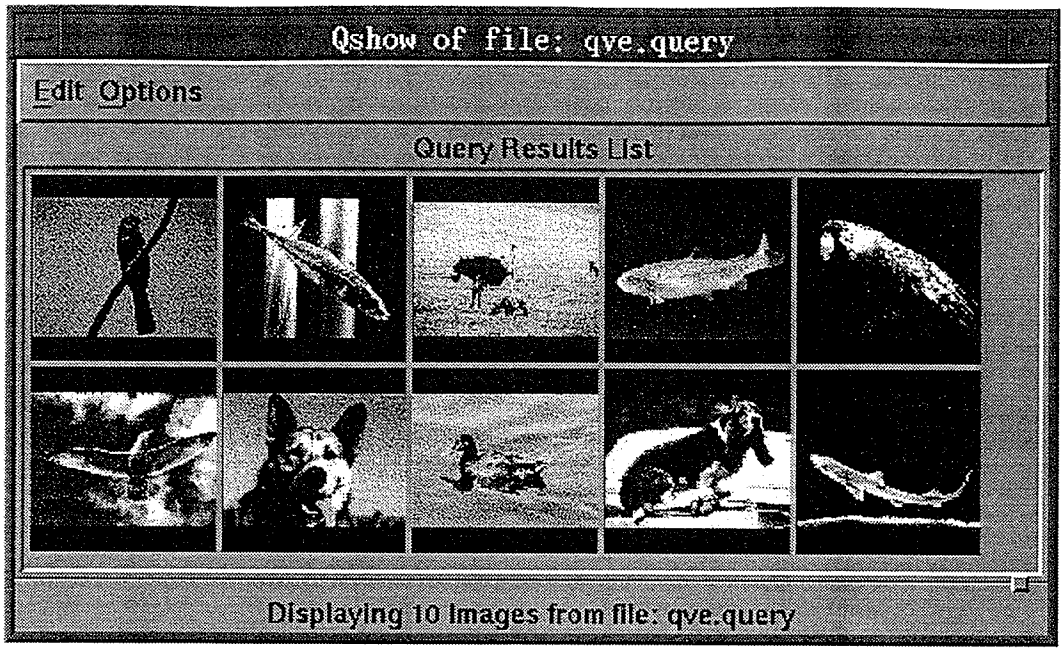

Figure 2.14: Result of the QBIC query[NBEF+93]

Figure 2.14 shows the result of the query. The images that are returned are those images in the database that most closely match the sketch entered by the user, with the best match in the top left hand corner.

2.3.5 SUPER

The *SUPER* visual querying facility by Auddino,*et al.*[AADD+92] has a diagrammatic query language, based on an extended E-R diagram model (ERC+). The main benefit of the system is the modularity of its implementation.

The ERC+ is extended to encompass object-orientedness. In addition to the usual projection and selection operators, ERC+ supports generalization and specialization as well as object identity and complex objects.

The SUPER system is designed to be very modular. It has a kernel, which is a layer between the database accessed and the display facility. The kernel can access either relational or object-oriented databases. On top of the kernel a display facility for both UNIX and a Macintosh has been implemented. The display facility consists of several different tools, including the design editor and the query editor. With the design editor it is possible to create a new database schema. In the query editor new queries for the database can be created.

The query editor consists of several different windows. One window is for the display of the E-R diagram. The query editor does not allow browsing of the schema, it presents the complete schema to the user. The user copies the concepts of interest to a query composition window. The instances of interest are then selected from the concepts, which have been selected in the previous step.

2.3.6 DOODLE

DOODLE by Cruz [Cruz92] is a visual query language for object-oriented database. Its most distinguishing feature is its extensibility. It allows the users to adapt the visual representation of the data to their individual needs, although some representations for the data are already predefined. These representations can be modified using an object-oriented language, or completely overwritten with different data representations.

To overwrite the predefined visualizations, the user has to define a mapping be-

tween the objects in the database and their visual representation. These modifications can be done in a graphical fashion.

F-Logic [Cruz92], a language for reasoning about objects, is used as a theoretical representation of the system. The visual representation, both the already implemented features, as well as the representations given by the user are translated into F-Logic.

2.3.7 HTML

Although the *Hypertext Markup Language* (HTML) [BeCo95], is not a visual query language by itself, it creates a user interface, which can be used to create querying tools that act much like the iconic languages described above. Actions can be associated with icons in HTML.

HTML is a simplified version of the page description language SGML [Gold90]. HTML allows the programmer to specify the general characteristics of the page. It is possible, for example, to format the text on the page in paragraphs and to create headings for each paragraph. Most of the final formating is done by the HTML browser so it is impossible to predict the exact appearance of a page when creating HTML code. This browser (eg. Netscape, Mosaic, or Arena) is an application that interprets the HTML and displays it as pages. The most important feature of HTML is that it is possible to link certain words or phrases in a page to other HTML pages. The user clicks with the mouse on these phrases and a new HTML page is retrieved.

Through the use of the *HyperText Transfer Protocol* (HTTP) it is possible not only to create a link to a page that resides in the same computer but to one on any computer on a network that runs an *HTTP daemon* (HTTPd). By linking together many pages on computers connected via the Internet, the World Wide Web (WWW)[BeCa92] is formed.

Through the growing popularity of the Internet and the WWW in particular many users already work with HTML based systems. Users who are not familiar with the WWW concept, will find the interface very intuitive. The only action that users can take is to click the mouse on a certain phrase of interest, and they will be presented with material related to the item.

So far WWW pages, as described in this section, have been assumed to be static pages created by a human programmer. Another approach creates WWW pages using the Common Gateway Interface (CGI)[CGI96, BeCa92]. The CGI permits users to execute programs through a mouse click on a WWW page. CGI facilitates WWW pages creation "on the fly" by an application that runs on the same server as the HTTPd.

One of the applications that could be provided by the CGI is a front-end to a text based database. This enables the user to interact with the database through simple pointing and clicking on phrases or words in the generated pages. Another feature of HTML is that not only can words be linked to other pages or program executions through the CGI, but graphics can be linked as well. The combination

of these two features (CGI and links through graphics) makes an HTML-based user interface very similar to the iconic querying systems described above. The advantage of an HTML-based user interface is that it is very easy to use because the user only chooses whether to click on an icon or not.

2.4 Comparison of a DQL with SQL

A study by Catarci and Santucci [CaSa95] has shown that both novice and expert users benefit from the use of visual query languages. SQL and QBD* were compared for their ease of use.

The study measured both the effectiveness of the query language, and the efficiency achieved while using a particular style of query facility. The effectiveness was measured by the accuracy with which the users managed to formulate the queries necessary to answer the test questions. The efficiency measures how much time the users needed, on average, to complete a certain task. Other less tangible differences such as the level of users' contentedness with a specific query system were not considered.

In the study, three different classes of users were selected: novice, intermediate, and expert. The users were differentiated by whether they had no prior knowledge of computer science whatsoever, programming experience but no knowledge of databases, or whether they had a good knowledge of databases. The users were subdivided into two groups. One group had to solve problems in SQL, while the other group solved problems in QBD*. Each group was taught the respective query

language extensively before the experiment so that any differences in the levels of effectiveness or efficiency could not result from lack of preparedness.

In the experiment itself three categories of questions, with increasing levels of difficulties, were prepared. Novice users were only expected to solve the easiest category of questions, intermediate users had to solve questions of easy and medium difficulty, while expert users were asked to solve all three categories of questions. The level of difficulty was measured by assigning a weight to each of the SQL constructs necessary to solve the query. "Joins" were assigned the highest weight followed by nested queries and so forth. The questions themselves were natural language descriptions of problems that had to be solved. The descriptions were the same for both users of SQL and QBD*.

The results of the experiment were that all levels of users can benefit from a visual query language like QBD*. In both measures of effectiveness and efficiency, users of QBD* surpassed the test candidates expressing the queries in SQL. Even expert users expressing simple queries fared better using QBD* than using SQL. The reason for this is that with a visual query language the user is freed from remembering cumbersome details about the database, like the exact names of tables and columns involved in the queries.

Chapter 3

Non Local Data Repositories

Traditionally enterprises had one mainframe-type computer. These computers were accessed by terminals and all processing was done on the mainframe, since the terminals had no processing power of their own. With the advent of PCs the situation changed. Instead of using them as a mere replacement for terminals, the processing power of these machines is utilized. The mainframe is used as a data repository, while the data are either totally or partially processed by the PC. The advantage of this client/server architecture, where the PC is the client and the mainframe is the server, is that the workload is shared. This reduces the need for costly mainframe upgrades. Ideally, the mainframe can be completely replaced by a cheaper kind of machine, like a high end workstation or a cluster of workstations.

If an enterprise is very large it will have regional headquarters or plants in different parts of the country. It is advantageous to the enterprise if the data that is stored at different locations is accessible throughout the organization. This is one of the

reasons for distributed databases. Anther reason is that reliability can be increased through the distribution of the databases, since there is no single point of failure.

Both client-server computing and distributed database management systems are briefly described in the subsequent sections. The main part of this chapter is the description of federated database management systems (FDBMSs) and how disjoint DBMSs can be integrated into FDBMSs. An FDBMS is a system that consists of two or more independent DBMSs, which remain independent, but still cooperate to give the user of the FDBMS the impression that a homogeneous system is accessed. Finally two examples of FDBMSs are given.

3.1 Client-Server Computing

Client-server computing [Sinh92] means that the task of producing a certain result is shared by different machines. Usually this task is shared between personal computers, which are relatively cheap and therefore on every employee's desk, and a more powerful, central machine. Different types of servers exist. For example database servers only distribute data to the clients and compute servers, may pre-process the data in some way for the clients. The following discussion considers database servers only.

The database resides on a central computer system, connected via a network to the clients. The clients access the central system if they want to retrieve or update data from the database. Figure 3.1 shows a schematic of the client/server model

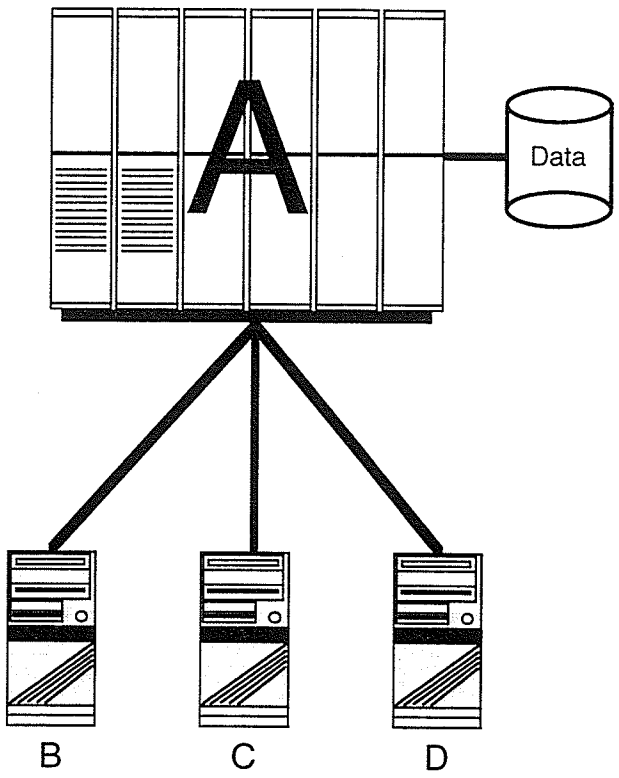

Figure 3.1: A client-server database system.

described. Machine A is the server and the other machines are the clients.

An example of client/server computing is described in the following way: The user on machine D needs data to complete a task. Machine D then sends a request to server A, this request will be in the server's query language. After the server has processed the query, the result is sent back to machine D, which processes it further as specified by the user.

In a single database environment, the database is characterized by the three level ANSI/SPARC schema architecture as depicted in Figure 3.2. An explanation of this can be found in Desai [Desa90]. The three levels of the architecture are:

- *Internal Schema* is the lowest level of abstraction in the database. It describes the physical location of the data on the disks. This schema is needed to optimize the data access speed on the disks. For example speed improvements can be made by clustering related data in close physical proximity, if it is often accessed together.

- *Conceptual Schema* is the next highest level in the hierarchy. Its function is to describe the data structure in the database and the relationships that the data has with each other. The conceptual schema can be used to reason about the data on a more abstract level.

- *External Schema* is a still higher abstraction of the database. It provides subsets of the conceptual schema, customized for a specific user or group of users. The external schema can be used to simplify reasoning about the database because only the relevant details for a specific database use are presented.

This architecture can be extended for the case of distributed databases.

3.2 Distributed Database Management Systems

In a distributed database management system (DDBMS)[OzVa91] there is no central database, as in a client/server environment, the data is instead distributed among all participating computer systems. This makes processing different than in a client/server architecture. To make the architecture truly distributed, each partic-

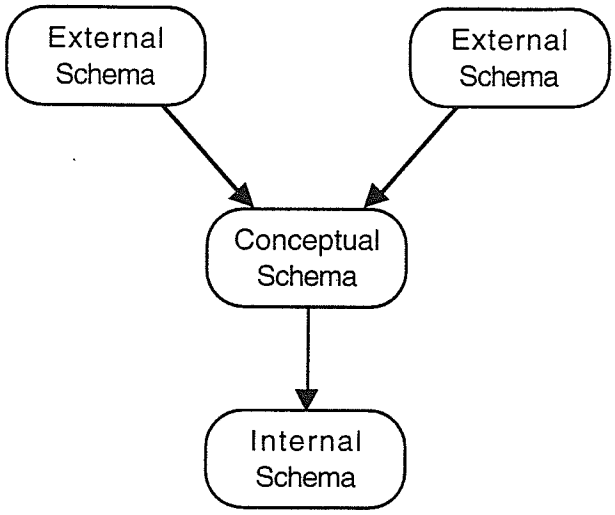

Figure 3.2: The three levels of the ANSI/SPARC architecture

ipating site has to store part of the database. This does not preclude the possibility that the distributed database also has clients attached to it but this is not the central characteristic. In this scenario all machines participating in the distributed database can be collectively considered to constitute the server.

A DDBMS can be characterized by a four level schema architecture. Figure 3.3 shows such an architecture. In addition to the components of the three level schema architecture used for a single database, a *Global Conceptual Schema* (GCS) is needed. The GCS consists of the conceptual schemas of the component databases. It is an integration of the local conceptual schemas and is used to reason about the collection of data contained in all participating local databases. The external schemas provide a simplification of the *collection* of databases which are integrated into a DDBMS.

Building a centralized database management system requires a data directory,

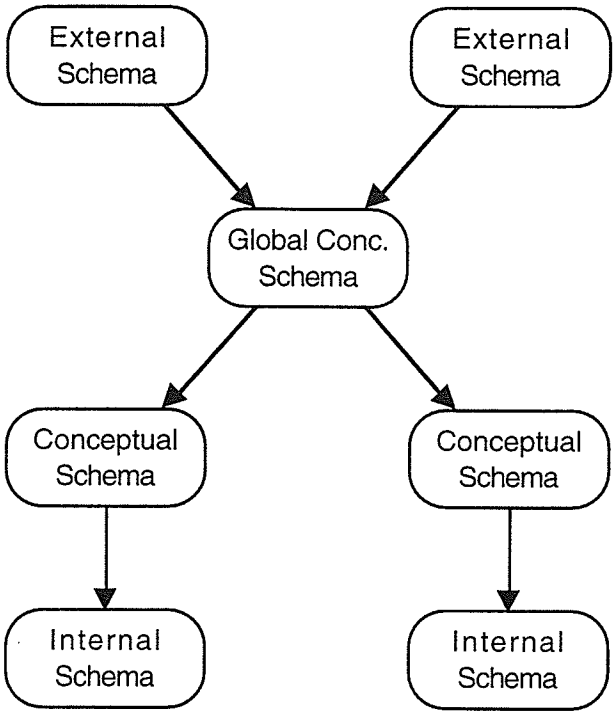

Figure 3.3: Four level schema architecture for distributed database systems

containing information about the concrete database. It tells a stand-alone system which part of a disk contains the required information. For a distributed system a *global* data directory (GDD) is necessary. It provides the distributed database system with information about which of its component databases contains the desired data item.

Both the distribution of data and the distribution of the GDD can be implemented in diverse ways. The data can be fully replicated so that each site has its own copy of the data, it can be non-replicated, so that each site has a distinct set of data, or it can be partially replicated so that there is some degree of overlap between the contents of the local databases at each site. Each of these different ways of distribution has its

advantages and disadvantages. The easiest to manage is a fully replicated or a non-replicated database system. For the fully replicated system, updates to the database have to be propagated to all sites, while updates for the non-replicated databases, do not have to be propagated at all. For the partially replicated database, updates have to be propagated only to some of the sites. The difficulty is to find an efficient way to determine which sites require propagating of updates. The disadvantage of non-replication is its lack of robustness because there is a single point of failure in the system. Full replication wastes a lot of disk space and is not particularly better than a non-distributed database. The increase in reliability offered by a fully replicated distributed database over a non-distributed database can be achieved via local replication. This can be the replication of disks (disk mirroring) up to the complete replication of the computer system (clustering).

Choices must be made about the placement of the GDD. The GDD can be stored on a single site or it can be distributed. The problems with the distribution of the GDD are the same as with the distribution of the data. A central GDD presents a single point of failure, while a distributed GDD is more difficult to manage.

For a pictorial representation of one type of distributed database system see Figure 3.4. It is a distributed database with a central GDD, which acts as a server to several PCs. A query might be issued at machine E. This query is first processed by machine B, since it contains the GDD. By querying machine B, it is found that machines A and C contain the data necessary to answer the query. After these sites

have been queried, the result is returned to machine E.

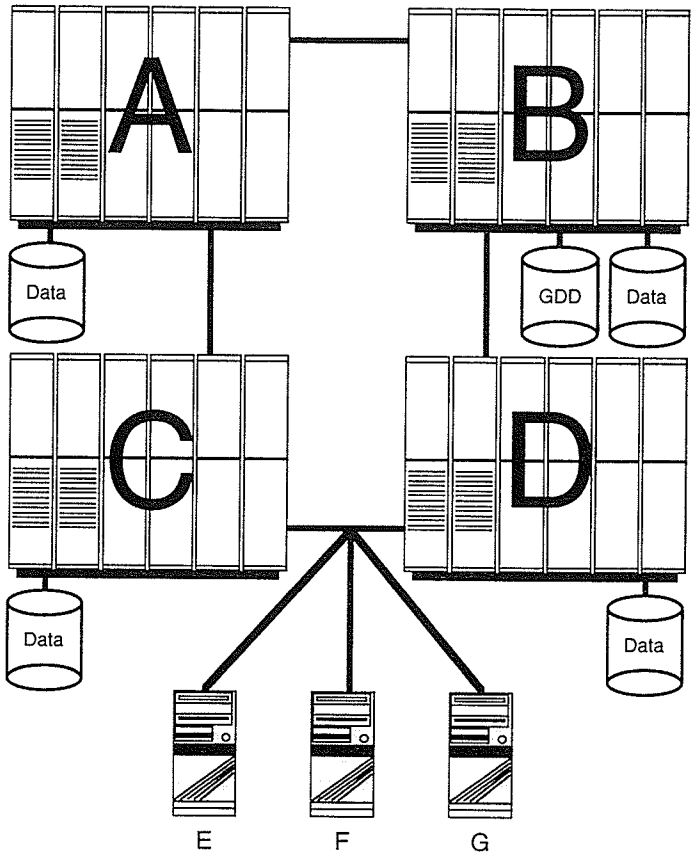

Figure 3.4: A distributed database system

Another aspect of distributed databases is the heterogeneity of the systems that participate in the database. For a distributed relational database system, heterogeneity poses significant problems, but for a distributed object base system, the problems of heterogeneity are even more severe because different systems have to be able to interpret the objects and run methods attached to the objects.

3.3 Federated Database Management Systems

A federated database is a collection of databases that were developed independently but must subsequently cooperate while remaining relatively independent. Reasons for this might be the merger of two companies whose databases now have to interact so that management can make decisions based on the contents of both databases. Joining the two distinct databases systems is necessary but it is too costly and time intensive to build a new database containing data from both companies from the ground up.

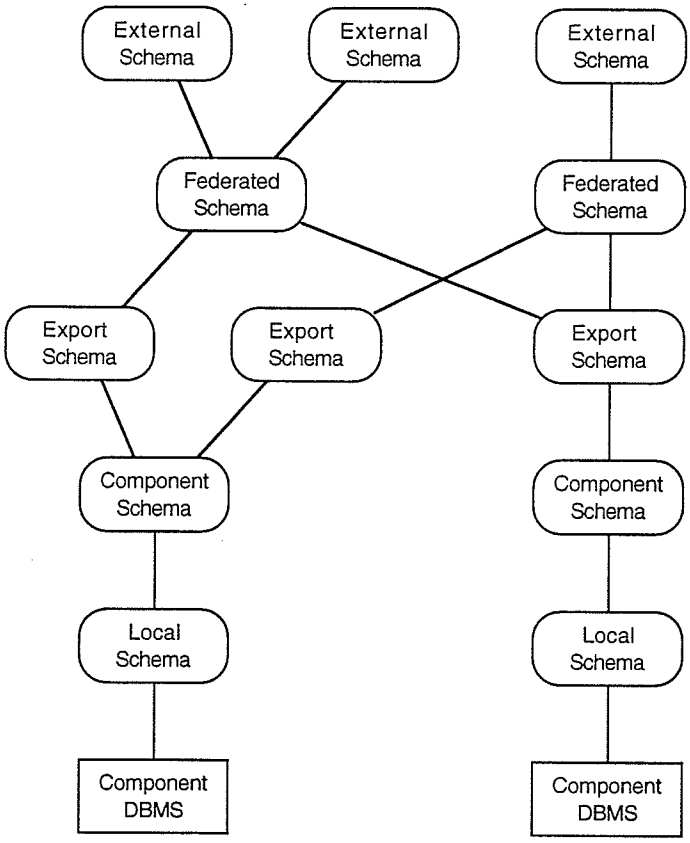

Figure 3.5: Five level schema of a federated database system according to Sheth and Larson[ShLa90]

Sheth and Larson[ShLa90] suggest that for a federated database the 3 level ANSI/SPARC schema is not enough. Instead, they propose a 5 level schema as depicted in Figure 3.5. The five levels are the *local schema*, the *component schema*, the *export schema*, the *federated schema* and the *external schema.* Each will be described briefly below.

The *local schema* is the conceptual schema of the underlying database system as defined in the ANSI/SPARC architecture. It describes the logical structure of the database.

The *component schema* is derived by transforming a local schema into a common data model. A common data model is a common representation of the different schemas. Common data models will be further discussed later. The transformation from a local schema to a component schema is achieved by a mapping function which translates commands for the component schema to commands for the local schema. Additionally, the results of the local schemas are transformed into the component schema representation.

The *export schema* restricts access by the federation's users to the data of the component schema. This is necessary, as there might be situations where not all the data in the local schema should be made public. The export schema is an easy way to control access to the component schema.

The *federated schema* integrates the export schemas of different databases. The federated schema is responsible for providing distribution information. Distribution information is the knowledge about which of the underlying databases contain the

required information.

The *external schema* provides a restricted schema on the complete federation. The external schema can be used to simplify the federation schema, or it can be used to restrict the schema for certain users.

The databases participating in a federation can have very different data models. While one database system might be constructed according to the hierarchical data model another database might be relational. These systems have both differing query languages and differing ways of presenting query results. To still be able to perform queries in a consistent fashion on such divergent systems a common data model is needed. The data model will act as an intermediate layer between the users of the federation and the actual databases and queries will be translated into the appropriate query languages and the results translated into a canonical form. The final goal is a federation that appears as one single database to the user.

A GDD is needed when a federated database system is built so a query can be directed at the appropriate component databases. One approach to accessing the federated system as a single database is the use of a so called mediator stage as described by Chakravarthy, *et al.* [CKTL93]. The mediator examines the query and decides, based upon the entries in the GDD, which of the databases contains the necessary data to satisfy the query. The mediator is also responsible for translating the query from the common data model representation into the appropriate data models and back.

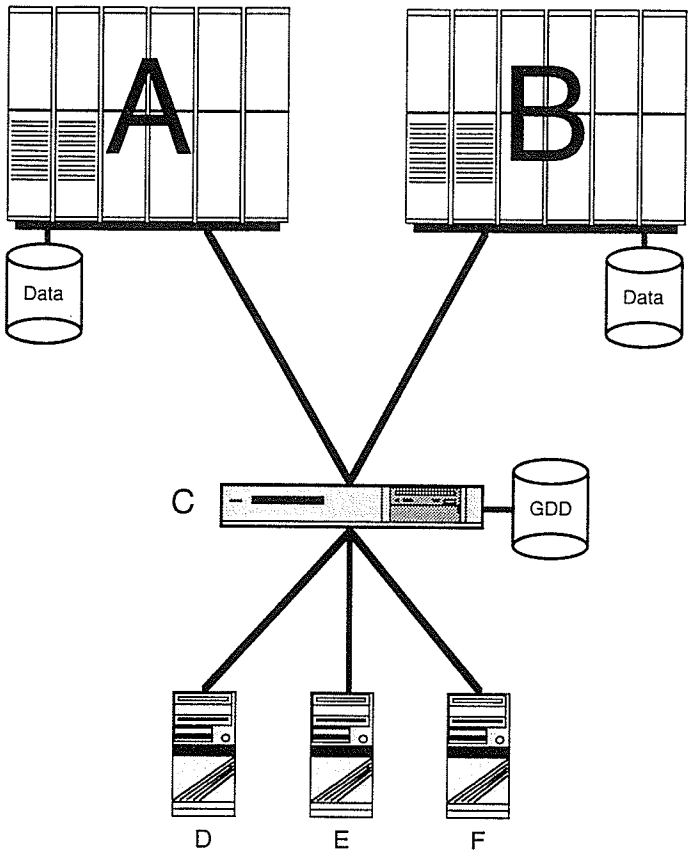

Figure 3.6: A federated database system

Figure 3.6 is an example federation of databases. It consists of two disjoint database systems, A and B. If a query to the system is issued by machine D, a client to the federation, the GDD will be queried first. The GDD resides on machine C. Based on the result of the query to the GDD, the query will have to be formed appropriately for the target system. This query formulation is also done by system C.

3.4 The Integration of Disjoint Databases

There are two levels of integration to be considered when trying to amalgamate two or more independent databases into a federated database system, as described by Lim, *et al.* [LSPR93]. The first level of integration is schema integration. Here the schemas of the databases are aligned, so that the structure of the tables and the naming of the columns is understood by the mediator stage. The second level of integration is the integration of instances in the databases. This is necessary to determine if two instances, occurring in the component databases, are actually referring to the same or to different, real-world entities. Each of these two integration approaches will be described in more detail in the following subsections.

3.4.1 Schema Level Integration

Schema level integration is the process by which the schemas, or logical descriptions, of two or more databases are aligned so that it is possible to reason about the different databases as if they were components of one, global database. During this process, it is necessary to identify the common concepts, if they are present at all, in the schemas of the participating databases. For example, a concept in the relational model is an entity or a relationship between two entities. Concepts here do not represent instances, whose integration will be described in the next section, but rather a grouping of instances.

The common concepts can be classified in several different ways [BaLN86]. The

concepts can be identical, in that two entities describe exactly the same collection of real world objects with exactly the same attributes. The next possibility is that two concepts are equivalent. Here there are different types of equivalence. There is behavioural equivalence, which means that for each relation in one database there is a relation in another database, that will produce the same answer for any given query. Then there is mapping equivalence. This equivalence means that for relations in one database there is one corresponding relation in another database. Another kind of equivalence is transformational equivalence. In this case it is possible to obtain a relation through a number of transformations from a relation in another database. If two concepts do not fulfill the criteria for identity or equivalence then they can still be compatible. This means that the two concepts are not directly contradictory to each other. The last category that common concepts can fall into is that of incompatibility. Here two concepts from different databases contradict each other.

Integrating the schemas of participating databases generally requires the four steps described by Batini *et al.* [BaLN86]. These steps are pre-integration, comparison of schemas, conforming the schemas, and merging and restructuring.

In the pre-integration step the schemas are analyzed and it is decided in which sequence the schemas are to be integrated. The sequence of integration can be handled in a binary fashion, where two schemas are integrated at one time to form intermediate schemas, which in turn are integrated two at a time, until the final schema emerges. Another possibility for binary integration is to integrate each new schema

with the existing intermediate schema. This method has the advantage that the schemas, which are considered to be the most important, can be integrated first. Another possibility is to integrate the schemas in a non-binary fashion, as for example, integrate all schemas in one step. The advantage of the binary methods is that the complexity of integration is reduced, because only a limited number of concepts have to be examined in each step. The disadvantage of the binary method, is that the integration steps have to be repeated quite often. For the non-binary method there are fewer integration steps involved but each step is more complex.

The comparison of schemas is the step after pre-integration. In this step all conflicts that occur in the representation of the different schemas are found. It is important to discover conflicts between the schemas, because only after the conflicts have been removed is it possible to determine the equivalence between them. The conflicts can be subdivided into two different classes: naming conflicts and structural conflicts. Naming conflicts arise, because the schemas for the different databases are created by different people, who refer to the same real world object by different names. On the one hand there is the problem of homonyms; where the same term is used in the schemas to refer to different objects. This can be solved by fully qualifying the attribute or entity name by prefixing it with an identifier for the schema it comes from. This way all objects with the same name in different schemas are uniquely identified. Further, synonyms occur when two or more names are given to the same object. This problem can only be found and solved through manual inspection of the

schemas.

Structural conflicts are based on differences in modeling "real-world" entities when the schemas were developed. There are four different kinds of structural conflicts to be considered. The first is a *type conflict*, which occurs when different abstractions are used to model the same real world object. An object could be modeled in one schema as an entity and in another merely as an attribute. The next kind of conflict is the *dependency conflict.* This means that the schemas differ in the way entities relate to one another. An example is when one schema models a relationship as 1:1 while another schema models the same relationship as $m : n$. *Key conflicts* must also be considered. These occur when the schemas use different attributes, or even a differing number of attributes as keys to an entity. Finally, there are *behavioural conflicts.* These are conflicts that occur when schemas have differing insertion or deletion rules associated with them. For example, this could mean that a certain attribute of an entity might be left blank in one schema, while its value was mandatory in another schema.

The third step in schema integration is conforming the schemas. In this step the participating schemas are remodeled in such a way that they are easily integrated. This means that conflicts identified in the previous steps are resolved. In the remodeling of the schemas, the concepts under consideration (i.e. entities and relationships) are transformed to a common canonical representation. The goal of this step is to arrive at a common schema representation for the participating schemas.

The last step in the schema integration process is merging and restructuring the schemas by superimposing the common concepts. The resulting schema is restructured for instance by finding generalizations and subset relationships in the schema. There are three qualities that the final schema should have: completeness, minimality and understandability. Completeness in the resultant schema is achieved among other methods by finding subset relationships in the new schema. These relationships can occur when one schema contains an entity that is a subset of an entity described in another schema. For example, there could be a person entity in one schema, and an employee entity in another schema. In this case, employee is clearly a subset of person. Another method is a join between concepts in two schemas to produce a common sub-concept. Minimality refers to the desire to create a schema with the least number of concepts possible. This is achieved by identifying and deleting redundant relationships and entities from the schema. Finally, it is preferable to produce the most understandable schema possible. This means that the graphical representation of the resultant schema should be easy to read by a human analyst. The same properties that are described in the chapter on visual query languages with regards to readability of a diagram apply here too.

Hammer, *et al.*[HaMS94] suggest a different approach to schema level integration. They describe a system for the integration of object-oriented databases, which is less geared towards producing a federation that is to be used by external users, but more towards building a peer-to-peer federation, where participants in the federation in-

corporate parts of the federation into their local schemas. The integration of different object-oriented databases is achieved with the help of a *sharing advisor*. The sharing advisor has knowledge of all export schemas of the different component DBMSs. It can be divided into four parts. These parts are *registration*, *discovery*, *semantic heterogeneity resolution*, and *unification*. The registration part is responsible for admitting new databases into the federation. The discovery part is used to decide which of the federation's data to integrate with the local schema of the database joining the federation. Semantic heterogeneity resolution resolves the differences in semantics between the local schema and the data types of interest found by the discovery part. The unification part finally integrates data of the federation into the local schema of the component database.

The databases that want to participate in the federation make themselves known to the federation in the registration part. They register the types that are in their export schema. The registration part builds up a semantic dictionary describing the registered types of the component databases. In the semantic dictionary, concept hierarchies are built. A concept hierarchy contains related type information of the registered types. If two types from different databases contain information about a similar subject, a concept is formed that contains the attributes that both types have in common, and the types form sub-concepts of the newly formed concept. The system determines if a new type is similar to existing concepts by comparing it with already registered concepts in a top down fashion (i.e. it compares it with the most

general concept first, if similarities are found with the descendents of that concept and so forth until its place in the concept hierarchy has been found). User interaction is required, if the registration part cannot determine by itself whether a type in a newly added database is related to any of the concepts that are already present. The similarity between the concepts is established by attribute names and types.

The discovery part of the sharing advisor is responsible for finding concepts of interest that can be incorporated into the local schema of one of the component databases participating in the federation. To find suitable concepts to integrate, Hammer, *et al.* define three types of relationships that the new concepts might have with the data in the local schema. These types of relationships are similarity, complementation and overlap. Similar concepts contain information that is related to the information already present in the local schema. Similar concepts are found by examining the semantic dictionary built up in the registration phase. All concepts that are at the same level in the concept hierarchy and have the same ancestor as a concept in the local schema are similar schemas. Complementary concepts are those that contain additional information to the information already contained in the concepts in the local schema. These concepts are found by examining the concept hierarchy and choosing those concepts that have different attributes than a local concept, but still having a common ancestor in the hierarchy. Overlapping concepts have some attributes in common. These concepts can be found by examining all descendents in the hierarchy and comparing them.

After concepts have been discovered that are suitable for integration into the local schema, the semantic differences between the attributes of the discovered concepts and the attributes of the concepts in the local schema have to be resolved. This is done in the semantic heterogeneity resolution (SHR) part of the sharing advisor. To achieve this, there is a lexicon present in each of the component systems which describes the relationships between the concepts present in the component system and the concepts known to the federation through the registration part. The relationships that can be specified are identity, equality, specialization, etc. The knowledge about the relationships is provided by the semantic dictionary. With the help of the SHR part of the sharing advisor it is now possible to find out more about concepts that were discovered as similar and how they are related.

The unification part of the sharing advisor is responsible for the integration of the newly discovered concepts into the local schema. The demands on the resultant schema are, as described by Batini, *et al.* [BaLN86]: completeness, minimality and understandability. To integrate a concept that is equivalent with one of the other concepts already present in the local database, the new one is made a subtype of the already present concept. This keeps all attributes that the two concepts have in common, while the new subtype has the attributes that are not present in the local concept but are present in the new concept. If a new concept is only related to the local concept, then a new supertype is created that has the attributes that the two concepts have in common. Both concepts will be descendents of the newly created

supertype and will retain the attributes that they do not have in common.

3.4.2 Instance Level Integration

After the schema level integration has been achieved, instance level integration may be necessary. Several different methods to perform instance level integration are described in [LSPR93]. The first possibility is the use of key equivalence. Here instances in the component databases are assumed to be the same if the key values in these component databases are the same. The problem with this approach is that there can be situations in which the keys of the respective tables do not match, but the instances in the tables still refer to the same real world entity. If the keys in the tables are composite keys, then it possible that, even though one attribute of the candidate keys matches, it still cannot be assumed with absolute certainty that the relations refer to the same entity. A method that will always provide the correct mapping between the two databases is *user specified equivalence.* Here the user or integrator of the databases specifies the mapping between the keys of the different tables in a separate table. The disadvantage of this method is that this table has to be constructed manually.

Other possibilities described in [LSPR93] are probabilistic approaches. In these approaches, the composite keys are considered equivalent even if only the components that describe the same characteristic of an entity match. It must have been established during the schema integration process, that these components describe the same

characteristic. For example, consider two tables, one an employee table, the other a customer table. The employee table's composite key is defined by the person's name and employee number. For the customer table, the key is defined by a name and a customer number. If during schema integration it has been determined that the two name columns of the composite key describe the same characteristic, (i.e. a person's name) then probabilistic key equivalence will define the relations in the two tables to be equivalent since they have a common name. Probabilistic attribute equivalence works in a similar fashion, only in this case not only the key attributes are considered, but all attributes that describe the same characteristic are taken into consideration. Both of these probabilistic methods have the disadvantage that they are not guaranteed to produce correct results. It is quite possible that specific keys or attributes have the same value but still do not describe the same entity.

For the method that Lim, *et al.* [LSPR93] suggest, they first define the notion of an extended key (K_{EXT}). The extended key is the minimal key that can uniquely identify the described entity in the federated database. Usually this extended key is the union of the keys of the tables which describe the entity. Furthermore, they develop the idea of an instance level functional dependency (ILFD). The ILFD specifies that an instance in one table is dependent on an attribute of an instance in another table. This is best explained with an example.

Table 3.2 does not contain the attribute `specialty` as Table 3.1 does. If it is known that *Football Shoes* are a product used for a sporting event rather than sold

name	specialty	street
JoeBobs	Sporting Goods	Portage Ave.
JoeBobs	Electronics	Main St.

Table 3.1: Table R

name	items_sold	city
JoeBobs	Football Shoes	Winnipeg

Table 3.2: Table S

in an electronics store, then it can be inferred that the JoeBobs store in Table 3.1 is the same as the JoeBobs store in Table 3.2. In this way the tuple in Table 3.1 has an ILFD on the tuple in Table 3.2 based on the `items_sold` attribute.

To identify which tuples are equivalent Lim, *et al.* suggest extending the participating table by the attributes of K_{EXT} that are not present in the respective tables. In the example, Table 3.1 is to be extended by K_{EXT-S}, or the attribute **items_sold**, and Table 3.2 is extended by K_{EXT-R} or the attribute **specialty** with the values of these attributes being NULL. Now each table has been extended to have K_{EXT} as the key. The next step is to fill in the columns for the newly created attributes. These are derived by applying the ILFD to the tuples. Now each tuple that agrees in K_{EXT} is deemed equivalent. This process is used for entity identification only, so that the original tables remain unchanged.

While this instance level integration approach has the advantage over the proba-

bilistic approaches in that it guarantees correct results, it still has the disadvantage that it is based on the semantics of the contents of the component databases. Therefore, it is also very time consuming since the ILFD have to be determined manually.

3.4.3 A Common Data Model

In the previous sections a need for a common data representation has been established. To reiterate, a common data model is necessary to overcome the possible divergence between systems that are to be integrated. There are a multitude of different data models in existence. The most common ones are the hierarchical model, employed by such DBMS as IBM's IMS, and the relational model, used by DBMS like DB/2, ORACLE and Sybase. A data model just beginning to emerge in the market place is the object based model [KiLo89]. It has many advantages over both the hierarchical and the relational model. It is capable of capturing very complex data much more easily and modeling it more naturally than the others. Such data exist in geographic information systems and CAD systems [BiOr94](among others). In addition to being able to model these complex kinds of data, it is also possible to model the same data structures that exist in relational databases using an object model. Thus an object based model is a superset of the relational and hierarchical models.

The capability to form a superset of many data models is important if several heterogeneous data models are to be integrated. With an object based model it becomes possible to provide consistent access to both a DBMS based on the relational model,

as well as an object-oriented DBMS, since an object model is able to capture the features of both data models. A relational model on the other hand is not capable of modeling an object based data model without substantial difficulties and extensions.

One data model that might be used as a common data model for a federation of databases is the RISC object model described by Manola, *et al.* [MaHe93]. It is designed as a common object model that can be used to map one object model to another. It has only a few fundamental building blocks upon which the mapping can be based, hence the name RISC in reference to *reduced instruction set computers.* The components of the model are the object state, object methods, object interfaces, object identity, types, and object construction. Some of these will be summarized below.

Objects have a state in most object models. This state is essentially the private memory of the objects. In the RISC model this state is represented through another object, which in turn has another object to represent its own state. This recursion ends when the state is of some primitive type defined in the model.

Methods are the functions that manipulate the objects. For the RISC model the methods are objects themselves. This feature gives them greater flexibility. Each has an invoke operation in its behaviour so that it can be called to perform its function.

The object interface describes how an object can communicate with other objects and methods in the system. The RISC object model does not have predetermined interfaces for the objects. The interfaces will be modeled after the interfaces of the

object models that are to be integrated. Therefore, it is left to the user of the model to determine which style of object interface fits best.

New objects are created in some object based systems by explicit object constructors (and deleted by object destructors). Object construction in the RISC model is also achieved via a `new()` method that creates objects for the primitive objects (e.g. the object state) that make up a RISC object.

3.5 Examples of Federated Database Management Systems

This section focuses on work described in the literature which is similar to the work proposed in this thesis. The first approach discussed is described [CKTL93], which outlines an integration approach for medical databases at the University of Florida. Another system similar to what is being proposed in this thesis is presented in [Meda95]. It is a graphical querying facility for a multi-media news server.

The challenge described in [CKTL93] is to create a federated multi-media database for medical research. The federation consists of three component databases, that contain very different kinds of data. One database stores the patients records and histories; the next database holds X-rays in digital format; while the last participating database is a general imaging database. The limitation of the initial setup was that these databases did not cooperate. They had to be queried separately to retrieve a

patient's medical history and the accompanying X-rays and other diagnostic images. The authors identified the requirements for a system that proposes to unify these three systems as follows: uniform access to all component databases, ad hoc query capabilities, and a customizable user interface. Since the component databases are also needed in their original form, it is not feasible to fully integrate all the components into a unified database system, but rather a federation of databases has to be developed. Differences in schemas among the components or the lack of a schema must be overcome by partitioning requests based on the type of data needed. In other words, the mediator knows where to find the data, based on their type.

The mediator providing the integration has two layers. The first layer is a mapping layer based on a global data directory that determines the database holding the needed data and translates the query from the federated system to a query that is appropriate for the specific database. The second layer is an access layer that interacts directly with the component databases and returns the results to the mediator layer. The clients of this database federation can also be subdivided into two categories: real-time clients and regular clients. Real-time clients are machines that might be used in an operating room. Due to the nature of their use, these clients need immediate answers to the queries, but have only a narrow query domain (i.e. the patient currently being operated on). Additionally the queries will be mostly browsing queries and not update queries. The proposed solution is to use clients that cache the data from the federation. In other words, the clients can be preloaded with the relevant data.

Regular clients, which are the other variety of clients considered by the authors, need access to broader ranges of data that may be updated and which therefore can not be practically preloaded. These clients have to retrieve the data as it is needed from the federation of databases. For performance reasons these clients also store data locally for processing. Therefore, this data has to be updated if a change in the federation occurs. The user interface on these clients has to be customizable so different kinds of applications are accommodated. It must also be capable of supporting the display and browsing of multi-media content.

The system proposed in [Meda95] provides a visual interface for distributed, multi-media capable news servers. The system follows the client/server model with the client having the visual querying capabilities and running the actual database client. The visual query facility allows the user to specify the kind of news article of interest. It is also possible to specify the desired medium of the news article. For example, video clips can be excluded from the returned results. The results can have hypertext links embedded in them, that lead to other articles. Therefore a browsing capability exists in addition to directly specifying the area of interest. The underlying servers can be continuous media servers (i.e. servers providing audio and video) or they can be non-continuous media servers for text. It is possible to specify a quality of service parameter for continuous media news articles. This can be used as one of the parameters for billing the user of the service. Unfortunately an explicit explanation of how the schemas of the underlying databases are integrated is not given by the

author of [Meda95].

Chapter 4

Implementation of the CDNA Project

The system described in this thesis models an information kiosk in a shopping mall. An information kiosk is a device located at central points in the mall that informs shoppers which stores carry desired merchandise. This system is an improvement over the currently existing signs in a mall, that only have maps telling the customer where a certain store is located, but cannot provide a complete inventory list of the different stores. The information kiosk tries to remedy this problem by providing the shopper with information about where to find an item of interest. The shopper is provided with a selection of items that the shops in the mall carry. A necessary condition for this is that all stores that participate in this system have a database that can be queried over a network.

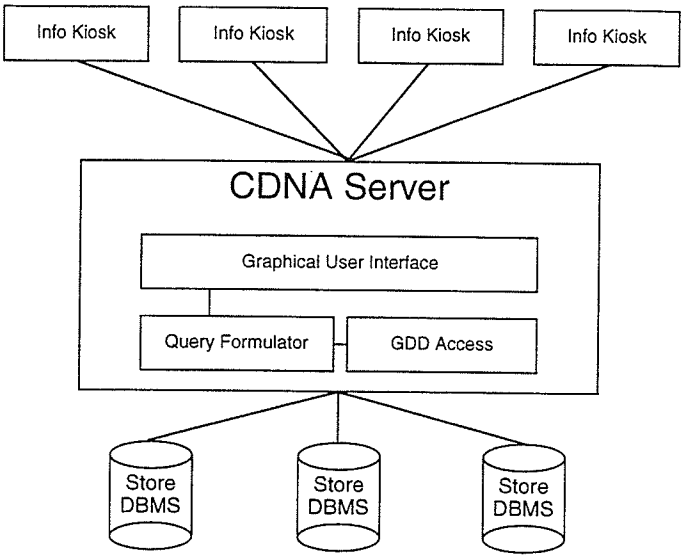

Figure 4.1: A conceptual overview of the architecture of the CDNA system

Figure 4.1 illustrates the system. The user enters the questions at one of the kiosk terminals, the questions are then processed by the Community Data Network Architecture (CDNA) system, which transforms the questions into queries for the database management systems of the stores, and displays the results that are obtained on the terminal.

This chapter will describe the implementation of a CDNA system. The main goals in the construction of the system were modularity, ease of use for the user, and scalability. The chapter is subdivided to reflect these goals.

4.1 Modularity

Modularity is desirable because this makes the system easier to maintain. This section describes ways in which modularity is achieved in the implemented system.

4.1.1 The System

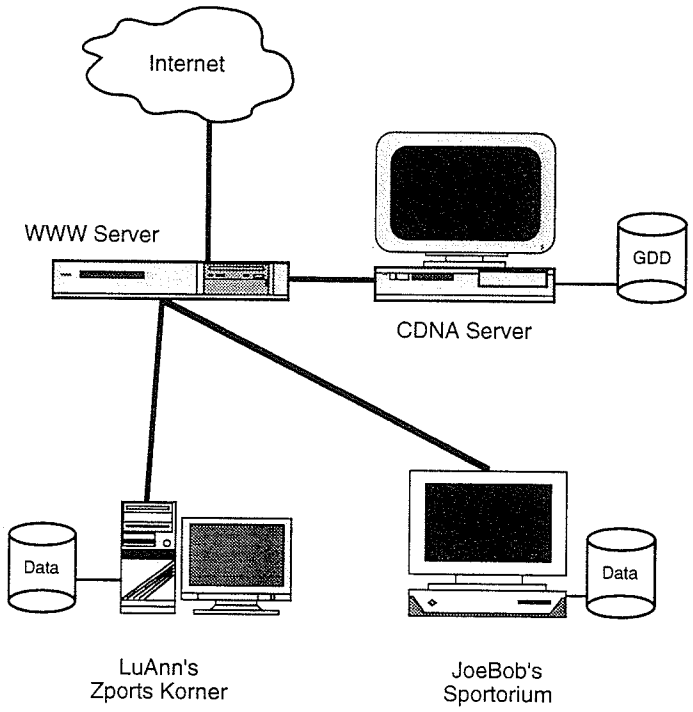

Figure 4.2: An overview of the realized architecture of the CDNA system

Figure 4.2 shows the architecture of the CDNA system as it is implemented. It is implemented on the local area network at TR*Labs*. The system consists of a Sun SPARC Station 5, which is used as a database server for one store (i.e. Joe Bob's Sportorium), while a 486 based PC running Linux is used as a database server for the second store (i.e. LuAnn's ZportsKorner). The CDNA system consists of a database

server that contains the tables needed for running CDNA, as well as an HTTP server. The database server is implemented on a DEC Alpha 3000/300. The HTTP server is running on the TR*Labs* WWW Server. The HTTP server and the database server can be run on the same machine but to reduce the load on the machine which runs the HTTP server, the tasks were distributed.

Through the HTTP server it is possible to make any machine on the Internet, which has a WWW Browser, into a terminal for the CDNA system.

4.1.2 The Database

Each of the participating stores must have a database, that is able to answer queries sent to it over a network. In the current state of the system, these databases are modeled by mSQL[Hugh96], a relational database.

mSQL was chosen to model the store databases because it has several desirable features:

- It is a relational database. Relational databases are not as powerful as object oriented databases, but their modeling abilities are sufficient for most business applications and they are in widespread use.

- It can be queried remotely. It can act as a server in a client/server scenario. This models the fact that the stores in the system keep autonomy over their data and database.

- It is multi-user capable. Up to 25 users can use the database engine concur-

rently, which means that up to 25 requests from the HTML terminals can be processed at the same time.

- It is free for academic use !

For each of the store databases a view should be created. This view would restrict access to data that is proprietary to the store, as for example, the wholesale price of their merchandise. Secondly the view would integrate the data needed by the CDNA system into one table. With mSQL it is unfortunately not possible to create views, so the views are simulated by tables that contain the same information that the views would have contained. In schema terminology, the conceptual schema of the databases is equal to the external schema. In a "real life" implementation, this would not be the case.

4.1.3 The Common Data Model

The common data model employed in this implementation is modeled in an object-oriented fashion so changing parts of the model is greatly simplified. Adding different capabilities to the data model is aided by this approach as well. So far a relational table based model is implemented. The reasons for this are that the underlying databases in a real world scenario are most likely relational as well. It is highly unlikely that business data would yet be stored in an object-oriented database as the relational model is adequate to support the needs that business has for databases. Business data is usually not very complex and can be categorized very well by re-

lational tables. Another reason for choosing a relational data model is that the databases that are used in the implementation are strictly relational and therefore an object-oriented data model cannot really be tested or utilized.

This object-oriented implementation of the relational data model still benefits the common data model. The benefits that can be derived from this are the usual benefits of using an object-oriented approach to programming. The code that is produced can be reused and it is easier to maintain and modify. This is achieved through the use of inheritance, polymorphism, and encapsulation.

Each of these points is important for the implementation of the CDNA project. Inheritance provides the reusability of code, as many of the methods that are being used to access the store databases are being used in communicating with the CDNA database server as well. Also the code that exists can be incorporated into further enhancements of the project, when access to different types of databases is required.

To incorporate different databases into the system the polymorphism property is important because the new access methods needed can be based on the existing methods. In conjunction with the common data model developed these methods will provide a consistent way to access the new data sources.

Encapsulation is important for consistent access because the data is requested from the object instead of being read from a variable. This request can remain the same regardless of what data model the database containing the information uses.

The data model, as it is implemented for the project, supports the insertion of the

results that are obtained from a query and the subsequent access to these results. The method of accessing the results is independent of the database the result is obtained from, so they can be accessed consistently regardless of how the database access is implemented.

4.1.4 Extensibility of the CDNA project

The CDNA project is designed so it is possible to extend the current project. One such extension is a global transaction manager. In a centralized database environment, a transaction manager is responsible for ensuring the consistency of the database and controls concurrent accesses to the database. Both of these issues are very important if the database is not a read-only database, but it also permits updates.

Each operation on the database must leave the database in a consistent state. This means that whatever failures occur, the operation must be completed. An example might be that while trying to update many records a system crash occurs. The transaction manager is responsible for ensuring that these updates are continued or undone when the system is restarted. To achieve this, logs must be kept of all operations and how far they proceeded. When the system is restarted after a crash these logs are consulted and the consistent database state is re-established. The recovery is achieved either by *redoing* certain operations that have not yet been written to a disk or by *undoing* those that have been pre-maturally written to a disk. An example of an undo operation is when a customer account has been debited, but the

item to be purchased is found not to be in stock.

In a multi-user environment, concurrency control becomes important because, multiple queries must not interfere with one another. For example, when part of one query writes to a data item that is being read by another query, care must be taken. The system must ensure that writing and reading occurs in a way that is equivalent to the serial execution of the two queries. There are many different protocols for making certain that the accesses occur in the correct order. One such protocol is the 2-phase locking protocol, based on *locking* records that are to be subsequently written or read. A write lock means that only the query, which has been granted the lock, is able to write to the specified record. A read lock means that the record cannot be changed, while a query has a read lock on it. With the 2-phase locking protocol, the first phase is for acquiring the locks, while the second phase is for freeing up the locked records. In other words, as soon as locks are being freed up no new locks can be acquired by this query. This protocol guarantees that the execution of the queries is equivalent to some serial execution.

These problems also exist in federated database systems. Here each of the participating databases will have its own transaction manager, but it is also necessary to ensure consistency between the different databases, so a global transaction manager is needed. For example, when a business transaction is made, where an item has been sold and the customer's bank account must be debited, while the vendor's bank account must be credited and a balance owing must be reset. Here it is very important

that this operation leaves the federation in a globally consistent state, where either both accounts are adjusted, or neither account is changed and the balance owing remains.

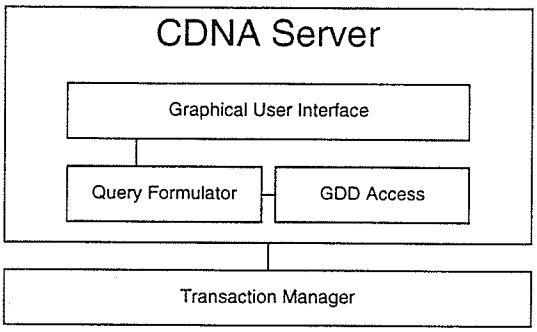

Figure 4.3: Conceptual addition of a transaction manager to the CDNA system

The CDNA project is designed so that a transaction manager can be easily integrated into the system. This would enable the system to go from a kiosk system, to an online shopping system, which is also capable of updating databases, while maintaining global consistency. Figure 4.3 shows how this would conceptually be attached to the system. Instead of accessing the databases directly, as is currently done, the queries would be sent to the transaction manager, which would in turn query the databases and return the results to the current CDNA system. In practice, this is achieved by calling a transaction management method or even a separate program, with the query and the final query destination (i.e. the store databases that the query is intended for) as arguments. After the transaction manager finishes processing the results can be returned in one of two ways: Either the results are returned as objects, if the CDNA system and the transaction manager are integrated into one

program, or the results can be stored in a temporary table, if the CDNA system and the transaction manager are two separate programs.

4.1.5 The CDNA Schema

In contrast to the general federated schema described by Sheth and Larson [ShLa90] the CDNA project does not need a five level schema. In the CDNA case a three level schema is sufficient to describe the system. Figure 4.4 shows the schemas that are needed. These schemas are the *local external schema*, the *component schema* and the *federated schema.*

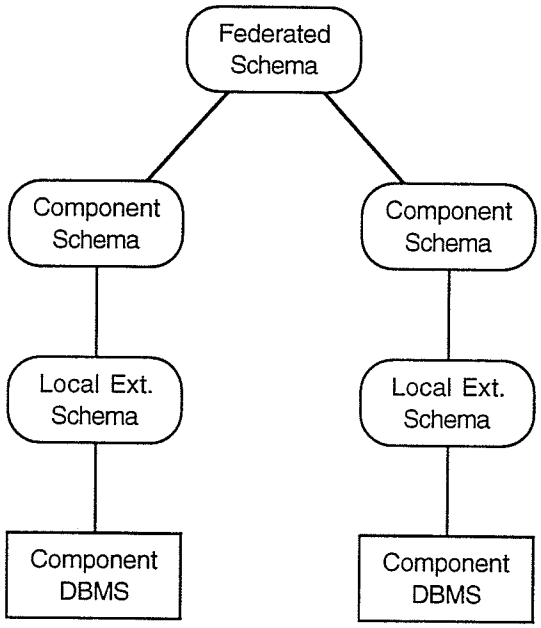

Figure 4.4: Schema of the CDNA system

The schemas are similar to the schemas described by Sheth and Larson. The local external schema is a part of the complete conceptual schema of the underlying

database, which only grants access to those parts of the database, that should be visible to the public. The component schema is the same as Sheth and Larson's component schema, it is the local external schema in a common data model description. Finally, the federated schema is the integration of the different component schemas. Both the export schema and the external schema described by Sheth and Larson are not necessary for the CDNA project. The export schema in their model is responsible for restricting access to the local schema of the underlying databases. It is not necessary because the access restriction to the data residing in the local databases is done at the database level by only exporting pertinent data to external schema. Similarly the external schema, which is responsible for restricting access to parts of the federated database system is not necessary in the CDNA project in its current form, because the access restriction is being done at the local schema level. None of the data that is being exported by the local databases needs any further access restriction, quite to the contrary, the stores will want the customers to look at them, so that a sale can be made.

4.2 Ease of Use

Since the system is targeted at the average shopper, no prior knowledge about computers or databases may be assumed. The querying of the system should therefore be as simple as possible. This is provided in the system by the use of HTML as a language to construct the user interface. With HTML it is possible to have a quasi-iconic

query language for the system.

The set of database systems that represent the stores can be viewed as a federation of databases because they have to remain separate, while in the context of CDNA it is necessary that they can be queried together.

Both of these points will be addressed in the following sections.

4.2.1 User Interface

The main goal of the user interface is to create a system that is as easy as possible to use. An information kiosk will only be accepted by the average shopper if it is extremely easy to use. To facilitate this ease of use a graphical user interface was chosen. This GUI has many of the capabilities of iconic languages. Recall that iconic languages have the benefit that the user clicks on the iconic representation of a concept and is presented with more information about this concept.

With HTML, which was chosen as a language to create the GUI for the system, it is possible to create such a visual query interface. It is also possible to mix the visual representation of concepts, with a textual representation. Textual and iconic representations are functionally equivalent as both can be *linked* to another page. Linking entails that a mouse click by the user leads to the display of a new page that is associated with the text or image. Since these two forms of display are equivalent, mainly the textual display is used in this system. The reason for this is that it is rather difficult to create meaningful icons as discussed in the section on iconic languages.

There are two ways of producing HTML pages, one is to create a page manually, the other is to have the HTML code created by a program. The manual writing of code is appropriate for pages that remain static, because their content does not have to be updated frequently. An example for this is the introductory page of the CDNA system. This page is rarely changed. It is only used as an introductory page to the system. The program based generation of HTML on the other hand is more suited to pages that change continually. An examples of this are pages that display the number of items a store has in stock, or in fact all pages that display the output of an external program. If these outputs are translated into HTML, they can be displayed by an HTML browser.

Very few parts of the CDNA system use static HTML pages. The entry page is static, as are the documentation pages. Where changes occur only very infrequently. Most of the HTML is created by a program that translates the user's mouse clicks into the query language of the underlying databases. The results from the queries are then translated back into HTML so that they can be displayed in an informative and visually pleasing fashion.

Each of the underlined words and phrases that the user can click on is a link to a CGI program. This CGI program is executed on the machine running the HTTP server when the user clicks on one of the underlined items. The CGI program then queries the CDNA server for more information on the subject that was represented by the phrase the user clicked on. New pages are created, based on the information

that is retrieved from the CDNA server. For information on the actual merchandise, the store databases are consulted.

4.2.2 Database Entry

One of the main benefits of the CDNA system over currently existing shopping mall systems is that the participating stores retain autonomy over their data because they do not ship the data to a single central site. Keeping the data at the stores ensures that the data remains as accurate as possible; it also does not allow any competitor to tamper with the data. Furthermore, since the data remains in the databases of the stores, all updates made to the data, like price changes or changes of the number of items in stock are immediately reflected in the system.

Since there are different stores, each with its own database, participating in the system, the CDNA system can be regarded as a federation of databases. Here the component databases clearly have to remain autonomous, since they are owned by the stores, and the stores will not give up the autonomy over their data. However, the data has to be accessible to the shopper in a simple and consistent fashion. The federation needs the integration of the component databases. Both schema level integration and instance level integration are necessary. The CDNA system provides support for both of these activities. Since instance level integration is always based on the semantics of the entities in the relations, some user input is required for it. The system tries to help the user with the input by providing hints for how the instances

could be integrated. The integration process of a new database into the system is also based on WWW pages, so that there is a consistent interface for both the shopper and the database administrator of the store's system. This means that the integration of a new database can be done from anywhere on the CDNA network.

4.2.3 Database Integration

There are separate tables in the system that contain the information necessary for the integration process. These tables are shown in Figure 4.5. First their functionality will be described and then a description of how they are built will be given.

The *Decision Tree* table contains a specialization hierarchy that is presented to the shopper. The shopper is given a choice between different high level categories of goods. After the initial choice the descendent nodes in the tree will be presented as the next choices that the shopper can make. This process continues until a leaf node in the decision tree has been reached. In the example this leaf node is *football.*

A hierarchical representation was chosen because it makes a goal directed search for merchandise very easy. It also corresponds to how a shopper shops for products in a regular store[BaLe93].

The *Product Index* table, the next table in Figure 4.5, is queried, when a leaf node has been reached. The query will select all instances that have the leaf node in the decision tree as their parent node. For each product in the store databases there is one instance in the product index table. These instances contain information

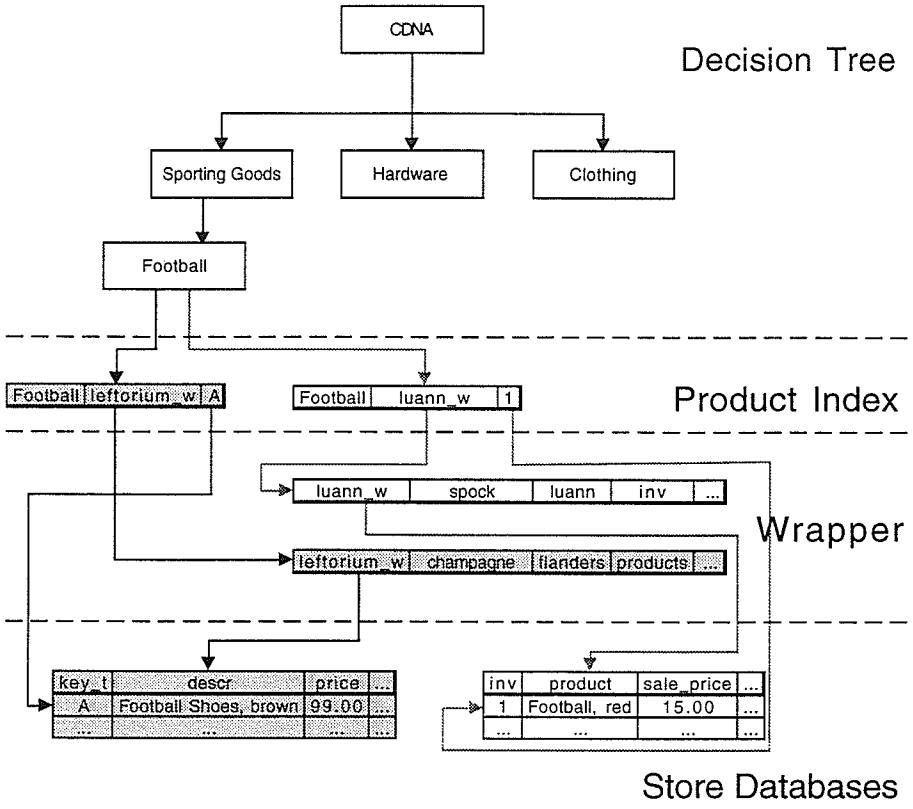

Figure 4.5: An overview of the interrelationships between different tables in the CDNA system

about which products the CDNA system knows. This is achieved by storing the key attribute of the actual store database. Should the key in the store database be a composite key then these columns can be amalgamated into one column through the view provided on the store database by the store. The next item of information stored in the product index table is which stores sell the particular item, and a reference to the stores that carry the item. This reference is a foreign key into the wrapper table, described in the next paragraph. Finally the product index also has an attribute that

specifies which leaf node of the decision tree is its predecessor. In the example given there are two products that are football related and they are sold by two different stores.

The *Wrapper* table holds meta data about the store databases. It contains the data that is entered through the web page shown in Figure 4.6. This table is responsible for schema level integration. Only certain columns in the view or table accessible to CDNA will be used. These columns are the minimal set of columns necessary to produce a meaningful overview of the merchandise that the stores offer. Furthermore, this is information that can reasonably be expected to be provided by the stores. The columns are a descriptive column, a price column, and an availability column. The descriptive column will contain the name of the item and some useful comments about the item. The idea for a wrapper table is taken from Tomasic, *et al.* [ToRV96] who present a similar structure.

Figure 4.6 shows the CDNA WWW page in which the necessary information has to be entered. In addition to the columns described above, other information is needed as well. The *Server name* and *Database name* information is needed specifically for mSQL. Since mSQL is capable of running on different machines and having several databases on each machine, this information has to be specified so the system can access the correct tables. With the knowledge about the naming scheme for the different tables, it becomes possible to formulate queries in a consistent and generic fashion, where the appropriate columns of the wrapper table are used in a fashion

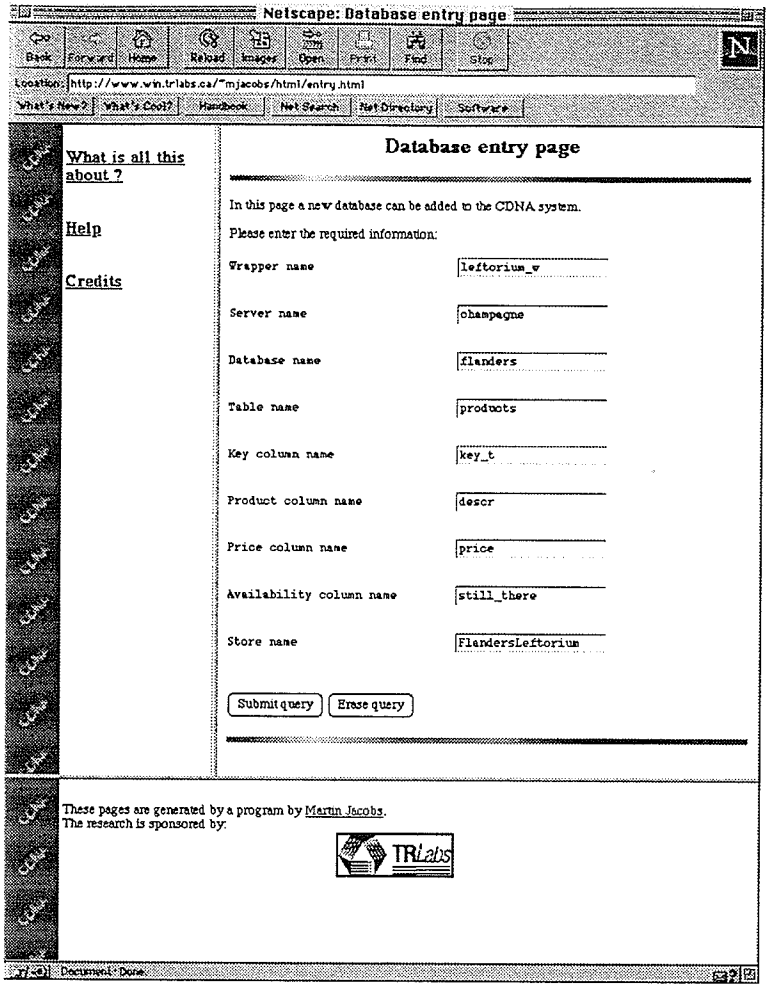

Figure 4.6: Necessary inputs to integrate a new database into the system

similar to pointers.

The *Store databases* refer to the actual store databases which will provide the final answers to the queries. In the example provided, the result of querying for football related items, if we follow the decision tree to the *football* leaf node, we would find brown football shoes sold by one store and a red football sold by another.

After the functionality of the tables shown in Figure 4.5 has been described, the

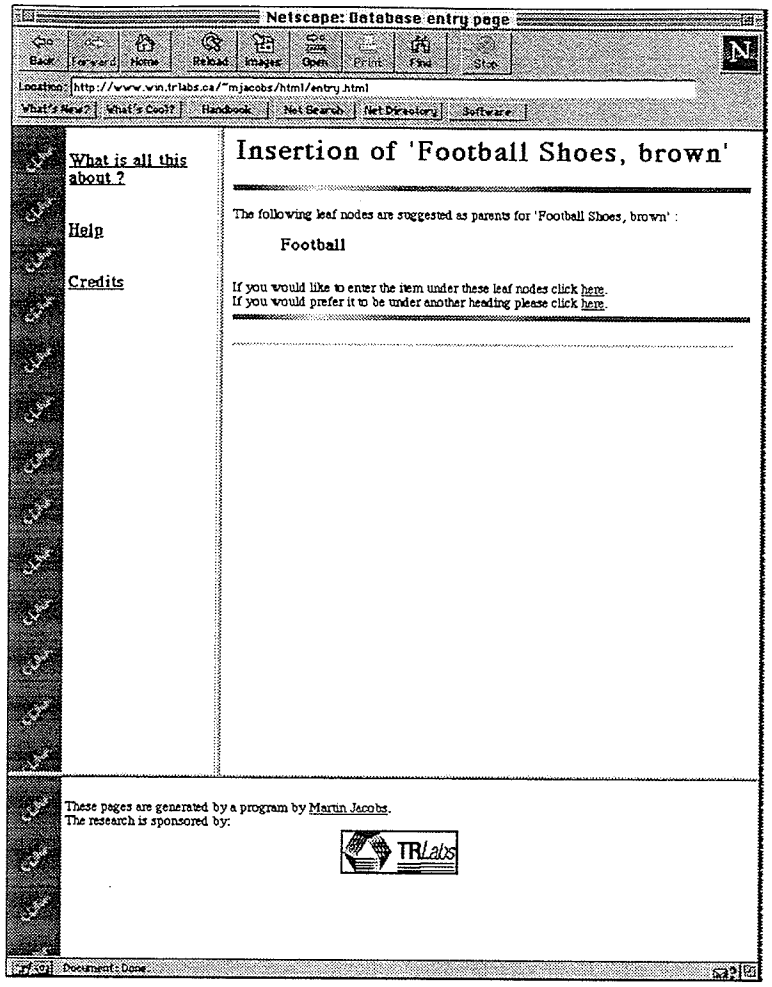

Figure 4.7: A suggestion for the placement of a product generated by the system

process in which these tables are created will be shown.

The database administrator for a store database that is newly entered into the system must first fill out the form shown in Figure 4.6. This information, as mentioned above, is stored in the wrapper table. With this, all the schema integration information that is necessary for the CDNA system has been acquired. Schema level integration is simplified in many respects in the CDNA system in comparison to schema level integration as described earlier. The pre-integration step for exam-

ple is not necessary because the sequence of integration is predetermined. The new databases are integrated one by one into the existing system. The comparison of schemas and conforming of schemas steps that are usually necessary for schema integration are also not needed here because it is known before integration which columns are provided by the database.

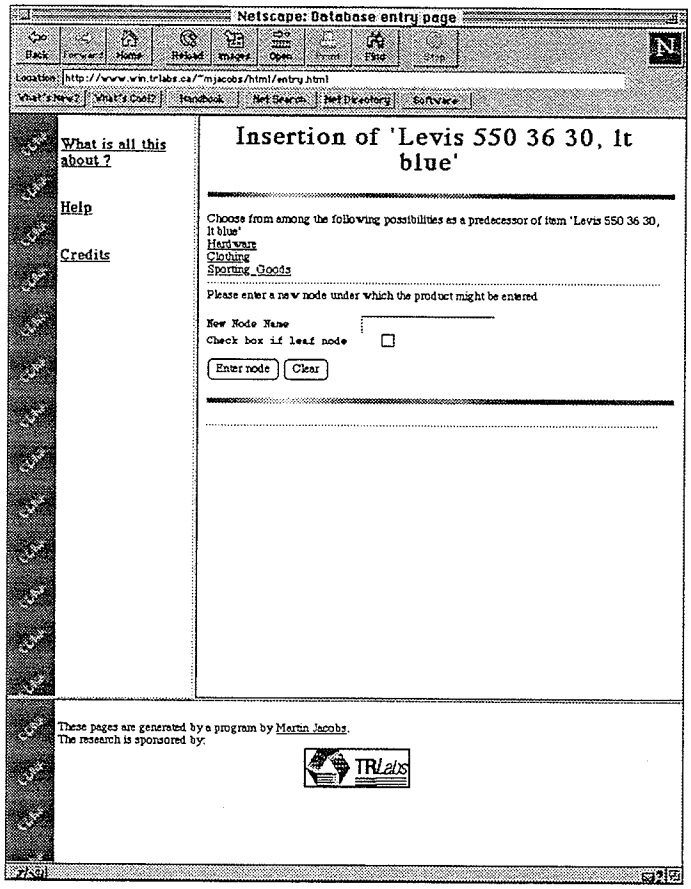

Figure 4.8: Selection of a parent for a new node

The next step is to provide instance level integration into the system. This means that for each item in the database a suitable leaf node in the decision tree has to be found. The system provides some help for the database administrator to simplify

this part of the integration process. After the initial information about the database itself has been provided, the system will query the store database for each item that is contained in the specified table. The contents of the column that describes the item is then compared against all leaf nodes in the decision tree. If the name of a leaf node is contained in the description of the product, this leaf node is suggested to the database administrator as parent to the particular item. This is shown in Figure 4.7. It is also possible to place the product under another leaf node. If this option is selected by the database administrator, a new node in the decision tree can be created. This is similar to the situation where the product name does not contain any of the names of the leaf nodes in the tree. To create new nodes, the database administrator first navigates through the decision tree, to find a place where a new node is needed. Then a new node can be created and sub-nodes of this new node can be made. This continues until a level of specialization has been reached where the product can be entered. Figure 4.8 shows the start of the navigation through the decision tree, while Figure 4.9 shows the creation of a new node. After the new node has been created, the product can be entered underneath it.

The product is entered into the system by storing the key value and the wrapper name of the store database in the product index table. Additionally, the decision tree leaf node that acts as the predecessor is stored in the row.

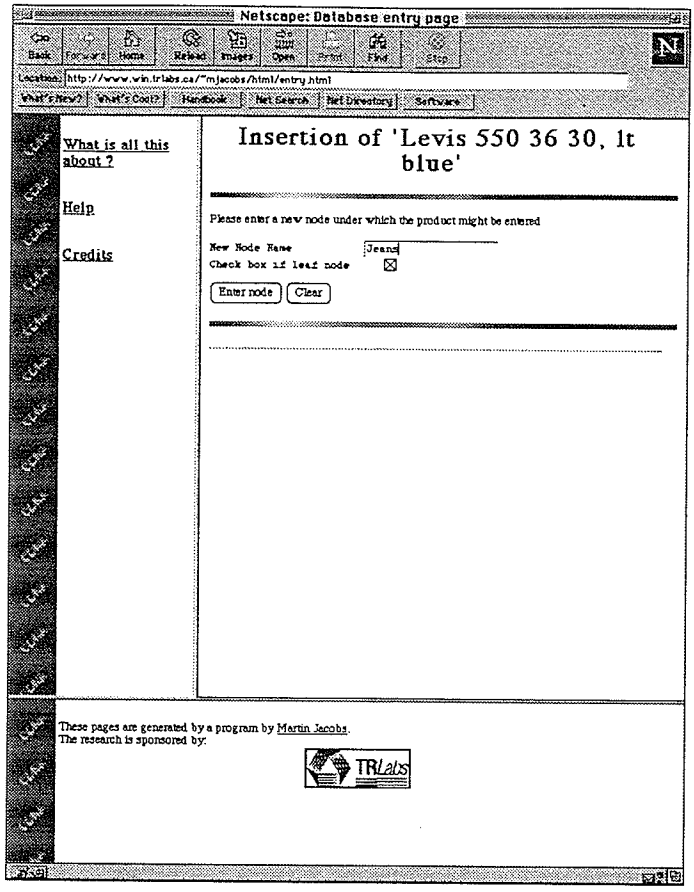

Figure 4.9: Creation of a new node

4.2.4 User Queries

The second part of the CDNA system, after the addition of databases to the system, is the querying of the system by shoppers. This querying of the system will be now be described.

The shopper is first presented with a high level choice of products that are on offer by the different participating stores. This presentation corresponds to the first level of nodes in the decision tree of Figure 4.5. At system startup, when the initial

HTML page is first called up, the decision tree is queried for the descendents of the root node. This is done via a CGI program that takes the root node, in this case "*Cdna*", as an argument and queries the decision tree table on the CDNA server. (In Figure 4.5 and also in the actual table, the *Cdna* node is the root node.) The root node does not carry any information but has to be present, as the system always queries for the descendents of a node. The result of the query for the descendents of the root node will be the top level hierarchy of products that are on offer. This top level hierarchy can consist of such items as *Sporting Goods*, *Clothing*, etc. These results are then translated into HTML. This is achieved by showing the result and attaching the invocation of the CGI program with a corresponding argument as an HTML link to the results shown.

Figure 4.10 is an example of the results of the initial query. This means that in this case the shopper can choose between three categories of products. All the products that the implemented system knows about fit under these categories. Each of these categories represents a node in the decision tree.

From this initial list the shopper can then select a category of interest. The selection of a category is accomplished simply by clicking on the desired underlined word. A query is issued on the decision tree which finds the descendent nodes of the node representing the product category that the shopper is interested in. The issuing of the query is again done via a CGI program that runs on the WWW machine, querying the decision tree table on the CDNA server. The results of the query are

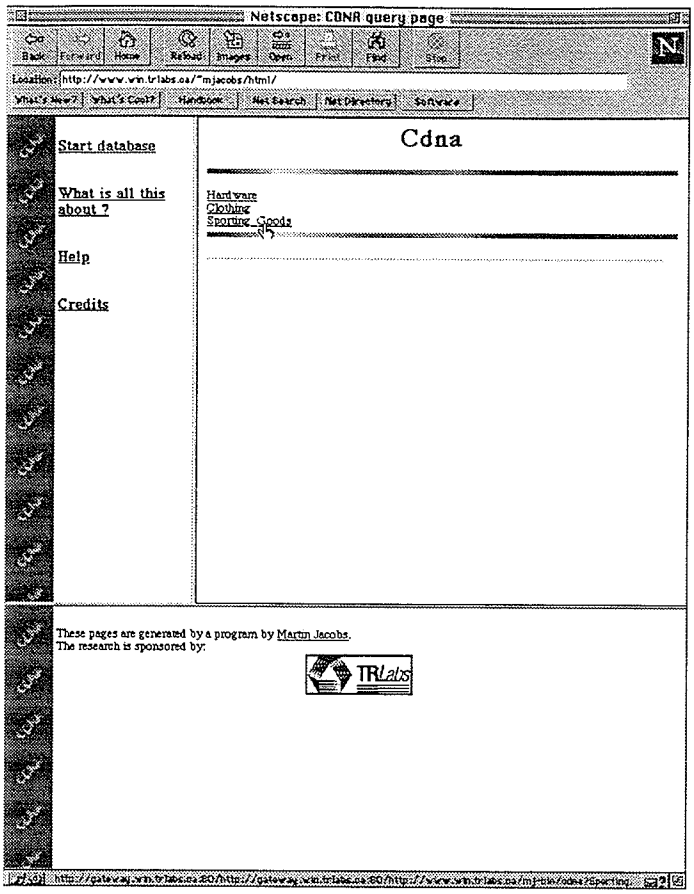

Figure 4.10: The first level of choices presented to a user

again shown on the HTML page and a CGI program with this node as argument is linked to it. As a result, the shopper is presented with the next layer of nodes in the decision tree. Further clicking on the category of interest will present a more specialized selection of categories. This process of the selection continues until the shopper has reached a leaf node in the decision tree.

If a leaf node in the decision tree has been encountered, the product index table of Figure 4.5 will be queried for all instances existing in this table, that have a leaf

node in the decision tree table associated with them. The results of this query on the product index table are all the products that the system knows about and that are classified as fitting in the category represented by the leaf node. The results are then ordered by the contents of the wrapper column. The wrapper column again contains a foreign key into the wrapper table, that contains the information about the different stores represented in the system. The results from the query to the product index table have to be ordered by the wrapper column so subsequent queries to the store databases can be issued together. This is important because the products that each store carries can be presented together in a single HTML table for each store.

After a leaf node in the decision tree has been reached and the product index table has been queried, the store databases have to be queried for the actual information of interest to the customer. These queries are formed based on the results from the product index table. The results contain the key values of the products desired. To form a correct query, information about the database is necessary which is contained in the wrapper table. Thus, for each store the wrapper table is queried for the names of the columns and table that contain the product information. The wrapper table only has to be queried once per store because the information obtained from the product index table is ordered by wrapper name. Based on the key values of the products of interest, a store database is queried. The result of that query is then transformed into a row of an HTML table. Such a table is shown in Figure 4.11. There is a separate table for each store. The information about the store, displayed

in the table (e.g. the store name) is contained in the wrapper table as well.

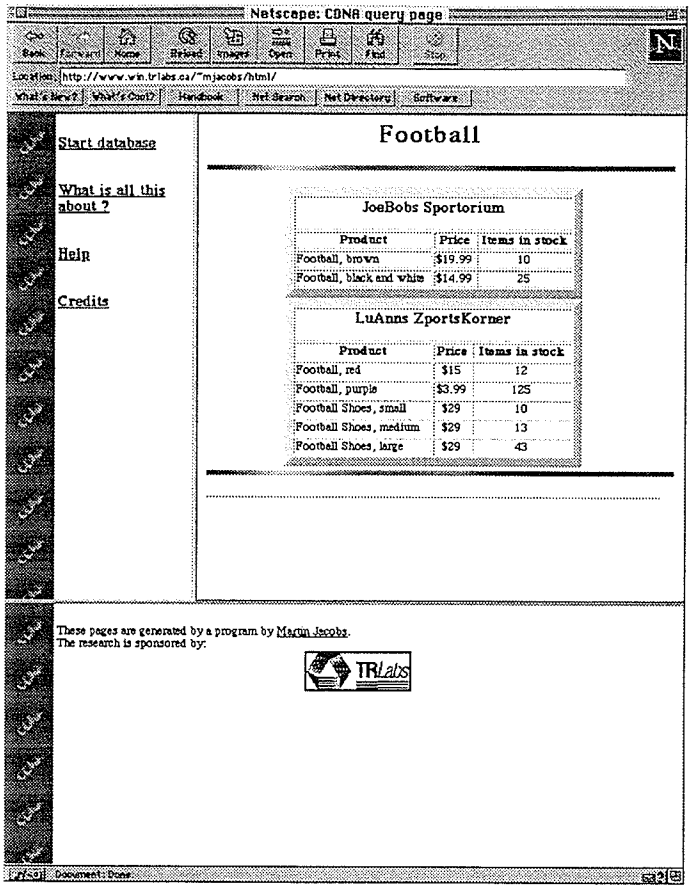

Figure 4.11: The final result of a query

Figure 4.11 shows the final result of a query posed to the CDNA system. It shows that there are two stores carrying items of interest to the user. Each of these two stores has a separate table on the HTML page, describing the products that they have for sale by a descriptive column, a column that contains the price, and a column that contains the number of items available.

4.3 Scalability

Another main feature of the CDNA system is its scalability. Through the choice of HTML as representation language, the system is not restricted to a physical mall setting. Since the HTTP protocol is associated with HTML, it is possible to connect the CDNA server to the Internet, and thereby have a world wide audience. Also, based on the size of the Internet mall and on the number of accesses that the system must handle, it is possible to have the CDNA server run on a different computer than the one that runs the HTTP server. This will reduce the load on the HTTP machine and enable it to handle more accesses. Another advantage of HTML is that the programming of the GUI has been greatly simplified. It is much easier to produce HTML, which will then be interpreted and displayed by the browser, than to create a GUI on Microsoft Windows, Macintosh OS, or the X-Window system directly. The HTML browser acts as an interface between these windowing systems and the programmer. Also the browser does not have to be programmed or maintained because there are many commercial HTML browsers available, that are maintained and constantly enhanced by their respective companies. A further benefit of using HTML is that HTML browsers are available for many different operating systems.

Chapter 5

Conclusions and Future Work

Formulating a query for a database management system is a non-trivial task for a person who has not been trained in the use of computers. Therefore, traditional query languages like SQL are not appropriate in an environment where the general public is supposed to gain knowledge from a DBMS.

This thesis presents an overview of currently existing techniques in the field of visual query languages, that try to simplify the use of databases. Both the diagrammatic query language approach and the iconic query language approach are described. Diagrammatic query languages are more suitable for users that have at least a little training in the use of databases because these languages usually require knowledge about the meaning of a database schema. Iconic languages are suitable for users who are not familiar with any kind of database, because they represent all entities contained in the data repository in a pictorial format. The actions that can be performed on the entities are also symbolized by icons. The user interaction is no more difficult

than simple pointing and clicking on the icons displayed. The thesis also presents arguments that an interface generated in HTML can be regarded as equivalent to a simple iconic query language because it is possible to use icons in HTML to represent entities on which certain actions can be performed.

To integrate multiple databases that must remain autonomous into a system that can be queried in a consistent manner requires a federation of databases to be established. The access to such a federation should appear to the user as if only a single DBMS is being accessed. User transparency is provided by a mediator stage [ToRV96] conceptually situated between the user access stage and the component databases. The user queries the federation by formulating queries that the mediator stage can process. The mediator stage in turn then formulates queries that are appropriate for the component database that contains the information necessary to answer the user's query. The component database returns the result to the mediator stage which transforms it into a common data model. The answer to the query is returned in a format understandable by the user. An overview of techniques for building federated database systems is presented.

The thesis then describes an implementation of the CDNA system which offers iconic access to a federation of databases. In its current implementation the CDNA system is an HTML based front-end to multiple relational databases that guides a shopper in a physical or virtual shopping mall towards products of interest. The information about the merchandise is retained at the store's site. An extensible data

model is developed based on the relational model but that can easily be modified to reflect different data models.

5.1 Recommendations for Future Work

There are several ways in which the CDNA project can be improved. One of these is the addition of a Transaction Manager, as briefly explained in Chapter 4. The transaction manager allows the system to be transformed from a read-only information system to a fully interactive electronic "mail order" system because a transaction manager allows on-line updates to take place. A possible scenario would be that an order is being taken, causing the amount of available units at the vendor's database to be decreased while the vendor's bank account is increased by the appropriate amount. At the same time the customer's bank account has to be decreased by the amount payable.

In the current system updates can only take place in an off-line fashion. This means that updates to the system can only occur when there is no user interaction, otherwise the system might not be consistent and the user might not get the correct results from the queries. Also it is not possible for the shopper to update the databases for reasons of possible inconsistencies. Adding a transaction manager to the system should be fairly easy because it is designed to be open to such an addition.

Another area where the system can be improved is by the use of Java [Java96, Lind96] as an enhancement of the current HTML based system. Java is an object-

oriented language that can be used to create applications which are called from an HTML page. The difference between Java and CGIs is that Java applications are not executed on the machine running the HTTP server but locally on the machine that requested the application. This gives the application much more flexibility in what it can do. All Java applications are executable on any platform that support a Java capable WWW browser. The application does not have to be modified according to which machine it is executed on. It is compiled into a byte-code that can be understood by the WWW browser which in turn interprets the byte-code for the appropriate architecture. The Java language itself is very similar to C++ so it is easy to learn for someone familiar with that programming language. In contrast to C++ it does not use pointers or explicit memory allocation but rather a garbage collection process. This avoids many of the commonly made mistakes when using C++.

The advantage of Java over HTML is that it is much more interactive. With Java it should be possible to implement a completely iconic language where one icon can affect another icon. As an example the user should be able to select the item of interest and drag this icon onto a cash register icon to specify that the item is being bought.

A further improvement to the system would be to introduce support for more databases than just mSQL. For relational databases this involves additions to the database access classes of the project. Academically more interesting is adding sup-

port for databases with entirely different data models. For example an object-oriented database is of interest, because this will require a common data model capable of translating a common query language into the appropriate database specific query language. Also a translation from the specific data model of the added database into the common data model will be necessary.

Bibliography

[AADD+92] A. Auddino, E. Amiel, Y. Dennebouy, Y. Dupont, E. Fontana, S. Spaccapietra, Z. Tari, Database Visual Environments based on Advanced Data Models, In *Advanced Visual Interfaces*, T.Catarci, M.F.Costabile, S.Levialdi (Eds.), World Scientific, pp. 156-170, 1992.

[AnCs91] M. Angelaccio, T. Catarci, G. Santucci, QBD*: A Fully Visual Query System, In *Journal on Visual Languages and Computing*, Vol. 1, No. 2 , pp. 255-273, 1991.

[AnEn95] M. Andries, G.Engels, A Hybrid Query Language for an Extended Entity-Relationship Model, Technical Report, University of Leiden, TR 95-03, 1995.

[BaLN86] C. Batini, M. Lenzerini, S. B. Navathe, A Comparative Analysis of Methodologies for Database Schema Integration, In *ACM Computing Surveys* Vol 18, No.4, pp. 323-364, December 1986.

[BaLe93] J. Baty, R. Lee, Electronic Shopping Infrastructures, a Design Representation, Technical Report, University of Rotterdam, RM-1993-01-03, 1993.

[BCCL91] C. Batini, T. Catarci, M. Costabile, S. Levaldi, Visual Query Systems, Technical Report, University of Rome, TR-04.91, 1991.

[BeCo95] T. Berners-Lee, D. Conolly, Hypertext Markup Language - 2.0, Work in Progress, W3, 1995. Available Online:*http://www.w3.org/hypertext/WWW/MarkUp/html-spec/html-spec.ps.*

[BeCa92] T. Berners-Lee, R. Cailliau, World-Wide Web, *Computing in High Energy Physics 92*, Annecy, 1992.

[BiOr94] A. Biliris, J. Orenstein, Object Storage Managment Architectures, In *Advances in Object-Oriented Database Systems*, A. Dogac, M.T. Özsu, A. Biliris, T. Sellis (Eds.), NATO ASI Series: Springer-Verlag, Vol. 130, pp. 185-200, 1994.

[CaSa95] T. Catarci, G. Santucci, Diagrammatic Vs Textual Query Languages: A Comparative Experiment, In *Proceedings of IFIP W.G. 2.6 Working Conference on Visual Databases*, Lausanne, 27-29 March, 1995.

[CGI96] The Common Gateway Interface, Available Online: *http://hoohoo.ncsa.uiuc.edu/cgi/overview.html*, 1996.

[CKTL93] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, F. Lambay, A Federated Multi-media DBMS for Medical Research: Architecture and Functionality, Technical Report, University of Florida, TR93-006, 1993.

[Cruz92] I. Cruz, DOODLE: A Visual Language for Object-Oriented Databases, In *ACM SIGMOD*, June 1992.

[Desa90] B. Desai, *An Introduction to Database Systems*, West Publishing Company, 1990.

[Gold90] C.Goldfarb, *The SGML Handbook*, Ed. Yuri Rubinsky. Oxford, New York: Oxford University Press Inc, 1990.

[HaMS94] J. Hammer, D. McLeod, A. Si, Object Discovery and Unification in Federated Database System, Technical Report University of Southern California, USC-CS-94-574, 1994.

[Hugh96] D. Hughes, mSQL, Available Online: *http://www.hughes.com.au*, 1996.

[KiLo89] W. Kim, F. Lochovsky (Eds.),*Object-Oriented Concepts, Databases, and Applications*, ACM Press, 1989.

[Java96] Java - Programming for the Internet, Available Online: *http://java.sun.com*, 1996.

[Lind96] P. van der Linden *Just Java*, The SunSoft Press, 1996.

[LSPR93] E.-P. Lim, J. Srivastava, S. Prabhakar, J. Richardson, Entity Identification in Database Integration, In *Proc. IEEE Int'l. Conf. on Data Eng.* pp. 294-301, Vienna, April, 1993.

[MaHe93] F. Manola, S. Heiler, A "RISC" Object Model for Object System Interoperation: Concepts and Applications, Technical Report GTE Laboratories Inc. TR-0231-08-93-165, 1993.

[McHo91] F. McFadden, J. Hoffer, *Database Management*, Benjamin Cummings Publishing Company, 1991.

[Meda95] G. El-Medani, A Visual Query Facility for Multimedia Databases, Technical Report, University of Alberta, TR 95-18, 1995.

[NBEF+93] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman,D. Petkovic, P. Yanker, C. Faloutsos, G. Taubin, The QBIC Project: Querying Images By Content Using Color, Texture, and Shape, In *Proceedings of 1993 SPIE/IS&T Conference on Storage and Retrieval for Image and Video Databases*, 1993.

[O2+90] O. Deux et al., The Story of O_2, In *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[OzVa91] M. Özsu, P. Valduriez, *Principles of Distributed Database Systems*, Prentice Hall, 1991.

[SaSo93] G.Santucci, P. A. Sottile, Query By Diagram: a Visual Environment for Querying Databases, In *Software Practice and Experience*, Vol. 23, No. 3, 1993.

[ShLa90] A. Sheth, J. Larson, Federated Database Systems for Managing Distributed Heterogeneous, and Autonomous Databases, In *ACM Computing Surveys* Vol 22, No 3, pp. 183-236, September 1990.

[Sinh92] A. Sinha, Client-Server Computing, In *Communications of the ACM* Vol 35, No 7, pp. 77-98, July 1992.

[ToRV96] A. Tomasic, L. Raschid, P. Valduriez, Scaling Heterogeneous Databases and the Design of DISCO , To appear in *International Conference on Distributed Computer Systems 1996*, 1996.

[Vois94] A. Voisard, Designing and Integrating User Interfaces of Geographic Database Applications, International Computer Science Institute, Berkeley, CA, TR-94-015, 1994.