GRADI: A Graphical Database Interface for a Multimedia DBMS[1]

Daniel A. Keim
Institut für Informatik, Universität München
Leopoldstr. 11B, 8000 München 40
e-mail: daniel@dbs.informatik.uni-muenchen.de

Vincent Lum
Systems Engineering, Chinese University
Shatin, Hongkong
e-mail: vlum@cuse11.se.cuhk.hk

Abstract

In this paper we describe the GRAphical Database Interface (GRADI) of a Multimedia Database Management System. As generally true, the user interface is an important part of the system which strongly determines the effectiveness of using it. In order to find an easy and natural way of interacting with the MDBMS system we examined the query specification process used by humans. We found that incremental query specification, predefined joins and an additional 'all'-operator are important to improve the process of query specification making it considerably easier to use compared to query languages like SQL. The Graphical Database Interface described in this paper incorporates the mentioned capabilities as part of the MDBMS system. We think that the principles are of general use not only for multimedia systems but for any database query interface.

Keywords: Graphical User Interface, Multimedia Database System, Natural-Language Interface, Information Retrieval

1. Introduction

The GRAphical Database Interface (GRADI) is designed as an integral part of the Multimedia Database Management System (MDBMS). It is designed to support a natural retrieval process of conventional and media data. The MDBMS system controls the management of multimedia data, which include image and sound among others, in addition to supporting conventional databases. The manipulation of multimedia data is not as straightforward as in conventional databases. One main problem is the retrieval of multimedia data from the database with the need to match the contents of multimedia data to a user query. In order to achieve a content based retrieval in our approach, we use *natural lan-*

1. This research was mainly done while the authors were at the Computer Science Department, Naval Postgraduate School, Monterey, California, USA and was supported in part by NOSC, Direct Funding and the German Scholarship Foundation.

guage captions allowing the user to describe the contents of multimedia data. In a similar manner, users will specify their queries on multimedia data contents in natural language form. One major problem with this approach is, that it is generally the case that the description of a multimedia data does not exactly match the description of a user query. The reason is that it is difficult for different users, or even the same user at different times, to describe the same thing identically because they can use synonyms or generalize/specialize categories belonging to the domain of interest and so on. Hence, in an earlier paper we proposed an intelligent approach to approximate matching by integrating object-oriented and natural language understanding techniques [KeKL91]. In the algorithms used for the intelligent matching only the retrieval of multimedia data by queries on the media part are considered. However, as is generally true, a query involves not only the media part but also the related formatted data. Queries may be composed of arbitrary combinations of conditions on the media and formatted data.

It is important to achieve an easy specification of complex queries composed of many conditions. The procedure for the specification strongly determines the effectiveness of using the system. Our goal in developing GRADI was to provide an easy and natural way of interacting with the MDBMS system. We examined the query specification process used by humans and found that in order to formulate complex queries a user partitions it into smaller pieces and puts them together in a later stage. This behavior is reflected in the principle of *incremental query specification* which is supported by GRADI. In addition, we observed that, for a given database, the joins necessary to specify most of the queries correspond directly to natural language expressions. This leads to the notion of *predefined joins*, also supported by GRADI. Furthermore, we introduce an 'all'-operator making the query specification in many cases considerably easier.

Over the last two decades several research projects have been conducted in the area of user interfaces for (multimedia) database systems. The first approach for a graphical database interface is the well-known QBE interface [Zloo77]. Most recent research in the area of user interfaces focuses on the entity-relationship [WoKu82, Fogg84, RoCa88] or the more complex semantic and object-oriented data model [KiMe84, GGKZ85, BrHu86, AgGS90], allowing queries to be directly specified within the schema. In contrast, we use an extension of the relational model to handle and manipulate the media data. In order to allow an easy query specification, we provide a graphical user interface which incorporates the capabilities as mentioned above, differing in many ways from QBE.

Another approach to achieve an easy query specification was chosen by researchers in the artificial intelligence area. Much effort went in building natural language understanding systems capable of understanding queries expressed in natural language. An overview over the research in this area can be found in [Alle87] and a good example for the current state of the art is the TEAM system [Gros87]. Because of the problems related to natural language understanding hardly any of the systems are actually used for retrieval in databases. In GRADI we are not trying to understand natural language. We argue that in order to express a query on formatted data in most cases natural language understanding is not necessary. The user can easily choose the necessary tables and attributes from lists, type the desired values and combine simple conditions into more complex ones. To our knowledge, none of the natural language interfaces to database systems can handle complex combinations of conditions because of the semantic problems related to multiple sentences. Furthermore, a graphical user interface like GRADI is generally applicable and less complicated and time-consuming than natural language understanding.

The paper is organized as follows: Section 2 gives a short overview of the Multimedia Database Management System (MDBMS). Section 3 outlines important features of GRADI and gives examples for query specifications. Section 4 describes the other database operations (schema definition, insert, update and delete) and Section 5 gives concluding remarks and a summary.

2. Overview of the Multimedia DBMS Prototype

As mentioned before, multimedia data, in the broadest sense, consists of unformatted data such as text, image, voice, signals, etc. in addition to alphanumeric data. A multimedia database management system is a system that manages all multimedia data and provides mechanisms to handle concurrency, consistency, and recovery in addition to providing a query language and query processing. In the following, we outline object model, architecture and main components of our Multimedia Database Management System (MDBMS).

2.1 Object Model

Despite differences in data model and implementation, all research projects on MDBMS have decided to organize multimedia data using the abstract data type (ADT) concept. This is generally accepted as an adequate approach. However, none of the projects have addressed the problem of content retrieval of multimedia data.

The fundamental difficulty in handling multimedia data is intrinsically tied to its very rich semantics. To answer queries posed on media objects a person must draw from a very rich experience encountered in life to derive a good answer. One must have a sophisticated technique for analyzing the contents of the images to get the semantics of different things in the images. Technology today is not advanced enough to expect systems to have this kind of capability to answer multimedia queries. However, we can abstract the contents of multimedia data into text and use the text description equivalent of the original multimedia data to match the user request or query. This is the principle we used in designing a MDBMS to handle multimedia data for different applications. Figure 1 shows the format of multimedia data which consists of the registration, raw and description data.

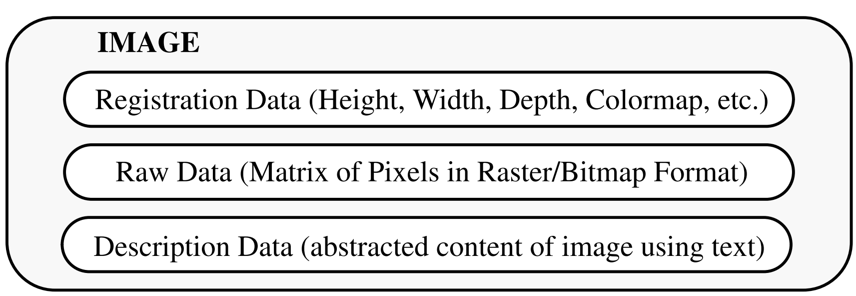

Figure 1: Example for the Multimedia Data Format

Raw data is the bit string representation of the image, sound, signal, etc. obtained from scanning or digitizing the original multimedia data. Registration data generally enhances the information about raw data and is not redundant. The contents of multimedia data are described by description data. We assume that users will supply the description data for multimedia data in natural language form. In future systems, however, the description could be automatically derived by the computer.

2.2 Architecture

The overall architecture of the MDBMS system is shown in figure 2. The components break down into GRAphical Database Interface (GRADI), query processor, data access and intelligent retrieval subsystem. The query processor accepts queries from GRADI and executes them by calling the other components. When a new description for a multimedia data is entered for example, the query processor calls the parser. The parser uses the dictionary to produce first-order predicates and returns them to the query processor. The query processor then hands the predicates over to the description manager which links the description to its multimedia data.

When the query processor receives a query the first task is to decompose the query into subqueries each affecting only the conventional or the media part. The conventional subquery is passed directly to the conventional data manager without modifications. For the text description, the query processor calls the natural language parser to obtain the equivalent query predicates. The predicates are then handed to the matcher. The matcher tries to match the query with the qualified multimedia data by comparing the predicates of the query with that of the stored multimedia data. The matcher does this by calling the description manager and using domain knowledge. As the solution to the natural language part of a query, the query processor receives links to the qualified multimedia data. After combining them with the results of the conventional subquery the final results are retrieved by the Data Access Subsystem.

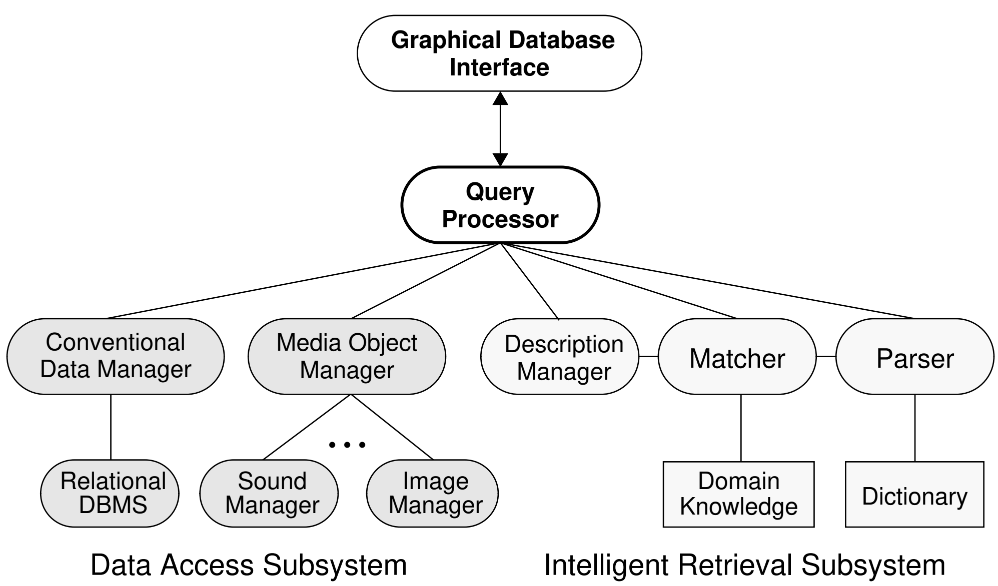

Figure 2: Architecture of the MDBMS System

2.3 Parser

In order to accomplish the goal of content retrieval of multimedia data, full understanding of natural language is not necessary. However, a restricted interpretation is necessary which is done by the parser component using the application dependent dictionary as a semantic basis. The dictionary or lexicon is necessary for parsing and gives each possible natural language word its part of speech, its grammatical form and the form of literals needed to represent it.

The parser automatically partitions the text description into subject, verb and object components and translates them into a set of predicates called the *meaning list*. The imprecision and ambiguity of the natural language descriptions are reduced considerably by transforming them into a set of predicates in first-order predicate calculus. These predicates state facts about the real world entities involved with multimedia data like their properties and relationships. Important features of the parser are *supercaptions*, a generalization of captions, and *frames for stereotypical actions*, allowing a set of predicates to be derived from terms in the description.

Our current implementation of the parser uses augmented-transition network parsing and interpretation routines. It is implemented in Quintus Prolog and running on a SUN SPARC workstation. The details of the parser are beyond the scope of this paper and are given in [Dull90, Gugl92].

2.4 Matcher

The major problem with content retrieval by natural language descriptions is that generally the description of a multimedia data does not exactly match the description of a user query because the same media object may be described differently. To solve the problem the matcher provides an approximate matching algorithm using domain knowledge organized as object hierarchies. The matcher searches in the noun and verb generalization hierarchies of the object classes and assigns weights depending on the distance in the object hierarchy. Then the weights for single component groups (subject noun, verb and object noun phrases) are combined using appropriate weighting factors as received from the user. Finally, the multimedia data with combined weight exceeding a threshold value set by the user will be retrieved. A prototype of MDBMS has been implemented at the Naval Postgraduate School [HoLR90, Pei90, LuMe91, LuKK92]. In this paper, we focus on the user interface called GRADI.

3. Query Specification in GRADI

The goal of a graphical database interface is to support the query specification process allowing the user to efficiently use the database system. It should allow inexperienced users to retrieve data from the database without having to know a specific query language. In today’s database management systems the user is forced to think in terms of data model and query language which differs greatly from the user’s way of thinking. Often a user can express a query easily in natural language, but has difficulties to express it in some given query language.

Most queries involve both media and formatted data. For the media part of the query we use our intelligent matching algorithm which is directly processing natural language captions. For conditions on formatted data, natural language expressions are mostly too imprecise to be directly processed. We try to overcome this problem in GRADI by allowing the user to directly select the desired tables and attributes.

The data model adopted in our system is an extended relational model. Despite some drawbacks the relational model has great advantages: It is well known, widely used and has a firm theoretical basis. For our purpose, we extend the relational model to capture media datatypes and, as shown below, we also extend the query language to allow the manipulation of media data and facilitate the query specification process.

Before describing GRADI in more detail, we first outline ways to achieve a natural query specification process. We are aware that what is natural to us may not be natural to others but in our context natural means also easy, convenient, useful and so on.

3.1 Towards a Natural Query Specification

Usually every user can describe a query (or at least the desired result) easily in natural language. Unfortunately, natural language expressions representing a query are imprecise and difficult to translate automatically into a formal query language to be understood by a database management system. We argue that the gap between the user's way of expressing a query in natural language and database manipulation languages like SQL can be improved considerably.

When comparing the user's natural language (NL) expression for a query with corresponding SQL statements the first difficulty is that the table and attribute names do not exactly match. In a graphical user interface this problem is easy to overcome. All table and attribute names can be presented to the user who simply selects the desired ones using a pointing device (e.g. mouse).

Another difficulty is related to joins between tables. In examining a large number of queries expressed in natural language as well as SQL we found that in most cases the join condition directly corresponds to some specific NL expressions. However, the tables connected by a join condition had to be deducted from the context because in most cases they were implicit. Additionally, the number of joins used in most of the queries was small compared to the number of possible joins. This can be explained by two facts. First, the number of semantically meaningful joins is restricted and second, some of the most frequently used joins are already intended at the design time of the database. In order to provide an easy way of expressing and using joins, in our system we allow the database designer and user to define and name joins prior to their actual use. A predefined join can involve more than two tables (e.g. two tables joined by means of a third table) thereby providing a simple way of expressing m:n relationships. Once defined and named, all predefined joins can be used to specify a query. Predefined joins differ from views: First, the result of a predefined join is not a table as in the case of a view, but a specific connection between tables. Second, predefined joins allow connections between different levels in nested queries and even recursive joins can be expressed. Examples are given in the next section.

Another thing we learned in examining the process of query specification is the handling of complex queries. Given a complex data retrieval task the user partitions it into smaller subtasks which are easier to handle. Starting with the clear parts of the query the user deals with all parts and combines the results into the final solution. In our system we support this way of handling complex queries by an incremental query specification which is described in the next section.

Finally, we observed that a special category of queries is easy to express in NL but rather complicated in a formal query language. Additional operators, closely related to corresponding NL expressions, allow an easier and clearer query specification. Considering for example a query like *"Select the name of ships which carry all weapons of the category surface-to-air!", w*e found that a special *'all'* operator would greatly enhance the readability and understandability of the SQL-like query making it similar to the user's NL expression. For the example we presume to have the tables *plane*, *weapon*, *ship_weapon* and a predefined join named *carries* expressing the m:n relationship between planes and weapons.

Example 1:

```
select s_name from ship
where ship carries weapon
      and w_nr = all (select w_nr from weapon
                      where category = 'surface-to-air')
```

A SQL statement expressing the same query without the 'all'-operator is rather complicated. Two possibilities are:

```
1. select s_name from ship
   where ((select w_nr from ship_weapon A
           where ship.w_nr = A.w_nr)
         contains
         (select w_nr from weapon
          where category = 'surface-to-air') )

2. select s_name from ship
   where not exists
         (select * from ship_weapon B
          where B.w_nr in (select w_nr from weapon
                           where category = 'surface-to-air')
                and not exists
                    (select * from ship_weapon C
                     where C. s_nr = B.s_nr
                           and C.w_nr = B.w_nr) )
```

3.2 Description of the Query Specification

In this section, we will describe GRADI in more detail by presenting several examples. We will show the main features of GRADI, especially those which are different from other graphical database interfaces. We start with a general description of the steps used in the retrieval process.

When starting the MDBMS system the user will be automatically connected to GRADI and the first step is to select the desired database. Then the user gets the system menu providing the main database manipulation functions: insert, delete, update or retrieve. When selecting retrieval, the user gets the query specification window and the first step is to select the tables to be used in the query. For each selected table a list with all attributes will be displayed in a separate window and all predefined connections involving at least one of the selected tables will appear in the *Connections* window. To specify the result list (projection) the user has to move the desired attributes to the *Result List*. Now, only the conditions need to be specified. Using connections, attributes of the selected tables and operators provided by the *Tool Box*, the query can easily be built using the mouse. In the *Query Representation* window the query is displayed graphically. Each part of the query is represented by a small box, simple conditions by a single box, subqueries by a double box, and the connection lines are labeled with the kind of connection used. An advantage is that every part of the query can be addressed for edit or delete at any time during the query specification process.

Predefined Joins

A special feature of GRADI are predefined joins. Predefined joins can be defined at design time of the database by the database designer. Having the necessary connections between tables in mind, the database designer tunes the database so that joins can be executed efficiently. All semantically meaningful joins can already be defined at design time. However, if other joins are needed later, the user can define them at any time.

Let us consider a sample database with the following tables:

mission (m_id, m_name, direction, goal, task)
navy_base (base_id, location, size)
officer (o_id, o_name, address, salary, commander_id, ship_nr, o_image)
ship (s_nr, s_name, class, yr_built, cap_id, mission_id, base_id, s_image)
ship_weapon (s_nr, w_nr, quantity)
weapon (w_nr, w_name, category, type, range, w_image)

Only few of the possible joins between these tables are semantically meaningful; e.g. the only meaningful equi-join between ship and officer is *ship.cap_id = officer.o_id*. Most other equi-joins like *ship.s_name = officer.address or ship.s_nr = officer.o_id* are senseless and will never be used.

Predefined joins allow an easier specification of complex queries. The user does not need to think about the attributes and conditions necessary to join tables, one simply chooses the desired predefined joins in the Connections window. Predefined joins can involve more than two tables, e.g. the following SQL statement expresses a three way join between ship and weapon:

Example 2:

```
select s_name, w_name from ship, ship_weapon, weapon
where ship.s_nr = ship_weapon.s_nr
      and ship_weapon.w_nr = weapon.w_nr
```

To express the join conditions of the same query in GRADI, only one step is necessary. After selecting tables and attributes the predefined join *ship **carries** weapon* has to be selected. The result as displayed in the Query Representation window is shown in figure 3.

Predefined joins can even be used to express joins of relations with themselves, e.g. the query *"Select the name of each officer together with the name of his immediate commander!"* can be easily specified using a predefined join. The user could specify the query as follows: First the officer table has to be selected. Since we deal with two instances of this

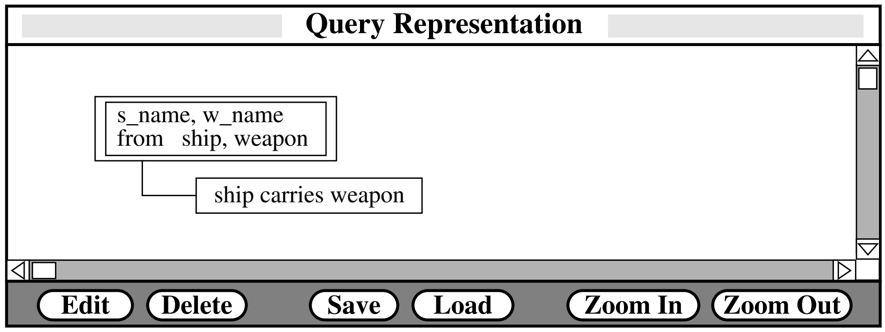

Figure 3: Example for a predefined join

table it has to be selected twice resulting in the officer1 and officer2 window. The last step is to select the predefined join *officer* ***is_commander_of*** *officer.*

Two more things about predefined joins need to be mentioned: First, any kind of join (not only equijoins) may be predefined and second, it is allowed to predefine identical joins with different names. This is useful to allow an easy identification of the required predefined join since the same query can be expressed differently. A simple example is *ship* ***carries*** *weapon* and *weapon* ***is_carried_by*** *ship.*

'All'-Operator

As mentioned before we introduced an additional 'all'-operator to make the specification of a special class of queries easier. The use of the 'all'-operator in GRADI is similar to other relational operators, e.g *'exists'* or '*in'*. The user specifies it by selecting an attribute, an operator (=, >, <, ≥, ≤) and a double box representing a (sub-)query or a table. The semantics of the 'all'-operator will be given in section 3.5.

Incremental Query Specification

To support incremental query specification we allow the user to start with any part of the query; e.g. to specify the query example 1 (see above) the user can start with the subquery *weapons of category 'surface-to-air'* and then continue with the main part of the query without specifying the connection between these two parts. At a later stage, the user may combine the separate parts.

As an additional feature, we provide an option to save and reload any part of a query for later use. If the user needs part of the query later for other queries the desired part may be saved by selecting the corresponding items in the Query Representation window and assigning a name to them. Later, when working on a different query, desired parts can be reloaded and integrated in the new query. Furthermore, to enhance the clarity of display, parts of a query can be grouped together and displayed as one box (zoom in). If the user wants to see the query in full detail at a later stage, the zoom out option can be used.

Tool Box

The Tool Box allows fast access to all functions supported by the system. The functions are divided into five groups: logical operators and basic elements *(AND, OR, Condition, Subquery)*, comparison operators *(=, >, <, ≥ , ≤)*, nesting operators *(Exists, not Exists, IN, not IN, ALL)*, set operators *(∪, ∩, −, ⊆, ⊇)* and aggregate operators *(AVG, SUM, MAX, MIN, COUNT)*. The semantics of most operators are the same as in SQL. The additional

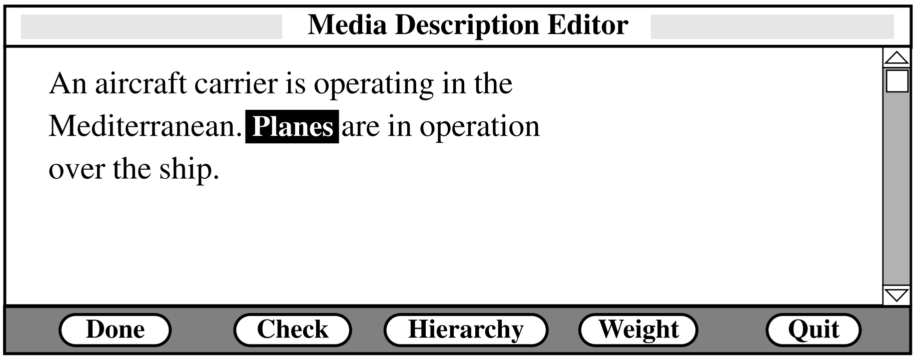

Figure 4: Media Description Editor

'all'-operator has already been introduced (see above). *Condition* and *Subquery* options are necessary for the incremental query specification process. Using these options, the user is able to continue the query specification with a different part of the query. When selecting *Condition* the user gets a new condition box and in the case of selecting the *Subquery* option a new double box for a new subquery is created.

Media Description Editor

Another important part of our system is the way of specifying the natural language description part of a query necessary when media data are involved. If the user selects a media attribute in the specification of the condition, automatically a special *Media Description Editor* will be displayed in a separate window where the media description can be specified. The description editor has special features to support the intelligent matching process mentioned above. When selecting the '*Check*' button the entered description is instantly sent to the parser. The parser tries to check and interpret it and, in case of an error, gives back the error message. The '*Hierarchy*' button supports the user in finding the right description. For a selected word or phrase (highlighted) it presents the corresponding part of the object-oriented domain knowledge base thereby providing hints for a better description (see figure 5 for an example). With the '*Weight*' button the user is able to assign weighting factors to the different component groups of a query. As mentioned before, the

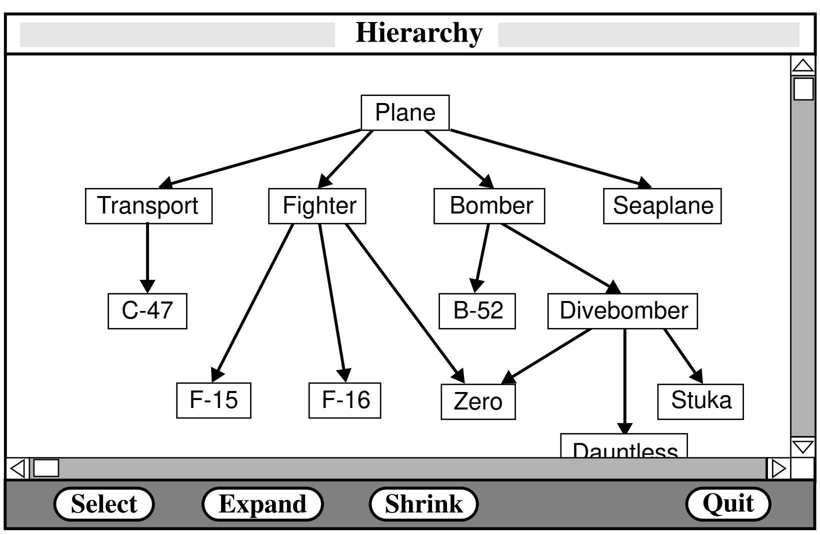

Figure 5: Example for a Noun Hierarchy

weighting factors are used to combine the weights of single groups. If the user does not provide weighting factors, an equal factor is assigned to all component groups. When selecting the *'Done'* button the description is automatically checked (like in case of the *'Check'* button) and the Media Description Editor disappears. If the user wants to edit the description at a later stage the corresponding box in the Query Representation window has to be selected and the *'Edit'* button has to be pushed. An example for the description *"An aircraft carrier is operating in the Mediterranean. Planes are in operation over the ship."* is shown in figure 4.

A Larger Example

To further explain the query specification process let us walk through a complex example:

> *"Select the name, base_id and image of ships which can carry all weapons of the category surface-to-air and where the image shows 'An aircraft carrier is operating in the Mediterranean. Planes are in operation over the ship.' ".*

When specifying the query the user might want to start with an easy part, e.g. "*weapons of the category surface-to-air"*. To specify this part the user first selects *Subquery* in the *Tool Box* providing a second double box for the subquery. Then *weapon* in the Tables window must be selected. As a result all attributes of the weapon's table are displayed in a separate window and by clicking to *w_nr*, the desired attribute is selected. The next step is to specify the condition. By clicking to *Cond* in the Tool Box the user gets an empty condition box in the Query Representation window and by clicking to the attribute *category* in the weapon's window, '=' in the Tool Box and typing in *surface-to-air* the box is filled with the actual condition.

As the next part the user might want to specify the image description condition *"image shows 'An aircraft carrier is operating in the Mediterranean. Planes are in operation over the ship.'* ". The specification process for this part is similar to the specification of the first part. The user selects the *ship* table and after getting a new condition box the attribute *s_image* from the ship window can be selected. Because *s_image* is a media attribute, the system automatically provides the special *Media Description Editor* window. In this window the user can type the natural language description for the image, in our example *"An aircraft carrier is operating in the Mediterranean. Planes are in operation over the ship."*. When selecting the *'Done'* button the description will directly be interpreted by the parser to get the equivalent predicates.

The last step is to specify the main part of the query and to compose the parts into the final result. Starting with the beginning of the query (*'Select name*, *base_id and image*') the user moves the attributes *s_name, base_id* and *s_image* to the Result List window. By selecting *Cond* from the Tool Box and *ship carries weapon* from the connections window the user specifies the join condition. Now, as the last part of the query, the user has to specify the 'all'-condition. This can be accomplished by getting a new condition box, clicking to *w_nr* in the weapons window, *'='* and *'all'* in the Tool Box and the double box representing the subquery *'weapons of the category surface-to-air'* in the Query Representation window. The last step is to combine the conditions into the final result. This is done by selecting the conditions and the logical operator AND from the Tool Box. In figure 6 the final result of the query specification process is shown.

3.3 Presentation of Results

An important aspect of a graphical user interface for multimedia database systems is the presentation of the results. The question is how to present a huge amount of multimedia objects. The problem is that, unlike conventional attributes, multimedia objects may have a time and space dimension.

To solve these problems we choose a combined form and list oriented approach. Generally, the results are presented as a list. In place of the media values only buttons are displayed which the user selects in order to see and/or hear the corresponding media object. Another way to see an object in more detail is to point to the line containing the desired tuple to get the tuple displayed in its form representation. Forms allow users to see more attributes (including media attributes) than available in a list; however, in contrast to lists, only one tuple at a time is displayed. By using the list representation the user can easily scan a huge amount of data but at any step the user has also the possibility to get the more detailed form version of a media object. When specifying a query automatically a new form is created including spaces for the values of all attributes involved. With the help of a graphical design tool the user can rearrange the form according to his needs and store it under a different name. In future queries the user can choose an already defined form when dealing with a similar query. In figure 7 the results of our sample query are shown using the customized form *withDescription*.

The combined list and form oriented approach only solves the space problem. It is highly desirable to have an influence on the time dimension of multimedia objects, too. Nobody wants to see a whole video in order to identify it as the desired one. Each time, a media object with time dimension (e.g. video or sound) is played, the user should have the possibility to stop, skip a part, go back, etc. In a special window all possible options should be presented as buttons so that the user can choose one of them using the mouse. A precondition for this kind of handling time dependant media objects is random access to their storage representation. In our MDBMS prototype system random access to media objects with time dimension is not supported yet and therefore we do not provide the features for time dependant objects. Other desirable options for time dependant media objects are the possibility to see a text version of a sound object (e.g. possible for speech or songs), the possibility to define index points which are directly accessible without linearly scanning the media object and the possibility to define synchronization points (for combined media objects).

3.4 Predefined Joins and Query Optimization

So far we have introduced predefined joins only from the user's point of view. In this section, however, we will explain the internal handling of predefined joins and related query optimization issues.

To process a predefined join the query is transformed substituting each occurrence of a predefined join by its definition. Additionally, missing tables are added to the 'from' part of the query. By this expansion some queries become more complex than necessary. The reason is that sometimes a two way join is sufficient although a three or four way join is substituted for predefined joins. In example 3 the predefined join *'ship **carries** weapon'* is used. In the expanded version the predefined join is substituted by the three way join *‘ship.s_nr = ship_weapon.s_nr* ***and*** *ship_weapon.w_nr=weapon.w_nr’*. However, for the evaluation of the query only a two way join is necessary (see optimized version).

Example 3:

select s_nr **from** ship
where ship *carries* weapon *(original query)*
and weapon.type = torpedo

select s_nr **from** ship, ship_weapon, weapon
where ship.s_nr = ship_weapon.s_nr *(expanded version)*
and ship_weapon.w_nr = weapon.w_nr
and weapon.type = torpedo

select s_nr **from** ship_weapon, weapon
where ship_weapon.w_nr = weapon.w_nr *(optimized version)*
and weapon.type = torpedo

In order to automatically generate a simplified version we developed an optimization algorithm to be applied recursively after substituting all predefined joins according to their definition. The first step of the optimization algorithm is to check whether a simplification is possible or not. A precondition for a simplification is that one of the join attributes must be the only attribute of that table used in the query. In this case table and join condition are omitted and each occurrence of the join attribute is substituted by the other attribute of the join condition. Formally, the optimization algorithm for a two way join can be described as follows:

$$\text{if } \left(a_{i_k} \in Attr[A] \right) \text{and} \left(\forall\, (k' \neq k) : \left(a_{i_{k'}} \notin Attr[A] \right) \right) \text{ and} \left(\forall n : \left(a_{j_n} \notin Attr[A] \right) \right)$$

$$\text{then substitute } \pi_{a_{i_1} \ldots a_{i_n}} \sigma_{\left(a_{j_1} \Theta\, c_1 \right) \ldots \left(a_{j_m} \Theta\, c_m \right)} \left(A \underset{a_{i_k} = a_l}{\bowtie} B \right)$$

$$\text{by } \pi_{a_{i_1} \ldots a_{i_{k-1}} a_l a_{i_{k+1}} \ldots a_{i_n}} \sigma_{\left(a_{j_1} \Theta\, c_1 \right) \ldots \left(a_{j_m} \Theta\, c_m \right)} (B) .$$

In more complex queries the optimization algorithm may be applied several times to achieve an even larger simplification. In example 4 a query is shown which can be simplified considerably (reduction from a four way to a two way join) using the optimization algorithm recursively.

Example 4:

select s_name, o_id, m_id **from** ship, officer, mission
where officer *is_captain_of* ship (original query)
and ship *is_on* mission
and ship.class = aircraft_carrier

select s_name, o_id, m_id **from** ship, officer, mission
where officer.o_id = ship.cap_id (expanded version)
and ship.mission_id = mission.m_id
and ship.class = aircraft_carrier

select s_name, cap_id, m_id **from** ship, mission
where ship.mission_id = mission.m_id (first optimization)
and ship.class = aircraft_carrier

select s_name, cap_id, mission_id **from** ship
where ship.class = aircraft_carrier (final version)

3.5 Semantic of the 'all'-operator

To make the query specification process easier we have introduced an 'all'-operator. The semantics of the 'all'-operator corresponds to the division-operator in the relational algebra. In this section we will explain formally how the 'all'-operator can be translated into the relational algebra.

Let us consider a simple one table query [$\pi_{a_1 \ldots a_m} \sigma_{c_1 \ldots c_n}(A)$] with the 'all'-operator being used once. To explain the semantic of the 'all'-operator we use an extension of the relational algebra and show how it is translated into pure relational algebra. For our extension we allow a condition c_i to be an 'all'-operator $c_i = all(B)$ with *B* being either another table or a query $B \equiv \pi_{a_1 \ldots a_{m'}} \sigma_{c_1 \ldots c_{n'}}(A')$. We define the semantic of the 'all'-operator by a transformation rule to be applied to all occurrences of the 'all'-operator:

$$\pi_{a_1 \ldots a_m} \sigma_{c_1 \ldots c_n}(A) \quad \rightarrow \quad \pi_{a_1 \ldots a_m} \sigma_{c_1 \ldots c_{i-1} c_{i+1} \ldots c_n}(A \div B)$$

In case of nesting of 'all'-operators the semantic of a complex query is defined recursively. Starting innermost, each occurrence of the 'all'-operator is substituted until all occurrences are transformed into the relational division-operator.

For a better understanding we apply the transformation rule to query example 1. The following query representing example query 1 in our extended relational algebra

$$\pi_{\text{s\_name}} \sigma_{\text{w\_nr} = all(\pi_{\text{w\_nr}} \sigma_{\text{category = 'surface-to-air'}}(\text{weapon}))}((\text{ship} \underset{\text{s\_nr = s\_nr}}{\bowtie} \text{ship\_weapon}) \underset{\text{w\_nr = w\_nr}}{\bowtie} \text{weapon})$$

may be transformed into a semantically equivalent query using only operators of the well defined relational algebra:

$$\pi_{\text{s\_name}}(((\text{ship} \underset{\text{s\_nr = s\_nr}}{\bowtie} \text{ship\_weapon}) \underset{\text{w\_nr = w\_nr}}{\bowtie} \text{weapon}) \div (\pi_{\text{w\_nr}} \sigma_{\text{category = 'surface-to-air'}}(\text{weapon}))) \quad .$$

4. Other Database Operations: Schema Definition, Insert, Update and Delete

In this section we will give an overview of the other database operations that are supported by GRADI. The operations to be described are Schema Definition, Insert, Update and Delete. For the schema definition we choose a rather simple table-oriented approach. The system designer defines a new relation by identifying name, type, width and key of all the attributes. The possible datatypes including media datatypes are presented in a menu. More important at this stage, however, is the possibility to predefine joins allowing an easier query specification by the end user.

Insert, Delete and *Update* are performed using a form-based approach. When creating a new table automatically a new form is created. The spaces for the attribute values reflect the possible length of values to be inserted or updated. As mentioned before, the user is able to rearrange the form according to his needs.

The **insert** operation is performed by filling a form with data. After specifying the attribute values for a tuple the user selects the *'Insert'* button to trigger the actual insert. During the insertion process also an *'Erase All'* button to erase all fields is available. After inserting a tuple the user remains in the form to insert other tuples. To quit the insertion mode the user has to use the *'Quit'* button. An example for the insertion window is given in figure 8.

The first step of the **update** operation is similar to the retrieval operation because it is necessary to identify the tuples desired for update by specifying a selection condition. The condition, a simple or complex one, is specified using the query specification window as described in section 3.2. However, only attributes from one table may be in the *Result List.* The result for the specified query is presented as a list and by clicking to one row of the list a tuple is caused to be displayed in a form. To change the tuple the user simply edits the values in the form and uses the *'Update'* button. Other buttons available in the form are the *'Next Tuple'*, *'Previous Tuple', 'Update All'* and, of course, the *'Quit'* button. By using the *'Next Tuple'* or *'Previous Tuple'* button the user gets the next or previous tuple found by the user given selection condition letting the displayed tuple unchanged. When using the *'Update All'* button an empty form is provided which the user fills to change all tuples found using the user given selection condition. Figure 9 shows an example for the update process.

Like update, **delete** is a two step operation. First, tuples must be retrieved according to a specified selection condition. In contrast to update, no *Result List* is necessary because tuples cannot be deleted partially. The second step, the actual deletion, is performed using the resulting list or a form. Both list and form provide buttons for deleting the tuples one-by-one or all tuples at once.

Another important issue is, how the media datatypes are integrated into forms because e.g. a sound cannot be displayed in a form and other difficulties arise for images. For the sound type two fields are necessary, one for the path of the sound file and one for the description. Furthermore a *'Play Sound'* button is available for each attribute of type sound to play the sound after inserting the path. For the attributes of type image a frame, two text fields for the path and the description and a *'Display Image'* button to display the image after inserting or updating the path are provided (see figure 8). The frame for the image can be of an arbitrary size making it necessary to zoom the image in or out.

5. Concluding Remarks

A major problem faced in today's database systems is the lack of an easy way to specify complex queries. It is caused by the gap between the user's way of thinking and the query languages used in most systems. Basically these systems are still linear in syntax. Such languages have not made use of the visual aspect of human senses nor the natural process of the human minds to process information. Although a lot of work has been done in the area of user interfaces for database systems no query language comes close to the natural query specification process used by humans.

Our contribution exploited in this paper is a graphical database interface supporting a natural query specification process. We combine an extended relational DBMS with an easy-to-use graphical interface allowing direct access to standard as well as media data. It narrows the gap between the user's way of thinking and formal query languages by using graphical user interaction. In our system, we support an incremental query specification, predefined joins and the special 'all'-operator to make the query specification process user friendly. The user is guided as much as possible allowing a quick, convenient and useful query specification. Further research is necessary to come even closer to the user's way of query specification e.g. by allowing the user to directly communicate with the system in natural language when appropriate.

We believe that our system provides a simple and elegant approach to the retrieval of multimedia data. The simplicity of our user interface lies in the easy way of query specification being directly obtained from queries expressed in natural language. We also believe that our approach is a general one that can be readily applied to most database query interfaces (e.g. relational systems and extensions hereof).

References

[AgGS90] Agrawal R., Gehani N. H., Srinivasan J.: *OdeView: The Graphical Interface to Ode*. Proc. ACM-SIGMOD Int. Conf. on Management of Data, Atlantic City, 1990.

[Alle87] Allen J.: *Natural Language Understanding*. Benjamin/Cummings Publishing, 1987.

[BrHu86] Bryce D., Hull R.: *SNAP: A Graphics Based Schema Manager*. Proc. IEEE Int. Conf. on Data Engineering, Los Angeles, 1986.

[Dull90] Dulle J.: *The Scope of Descriptive Captions for Use in a Multimedia Database System*. M.S. Thesis, Computer Science Department, Naval Postgraduate School, Monterey, CA, 1990.

[Fogg84] Fogg D.: *Leesons from a 'Living in a Database' Graphical User Interface*. Proc. ACM-SIGMOD Int. Conf. on Management of Data, 1984.

[GGKZ85] Goldman K. J., Goldman S. A., Kanellakis P. C., Zdonik S.B.: *ISIS: Interface for a Semantic Information System*. Proc. ACM-SIGMOD Int. Conf. on Management of Data, Austin, TX., 1985.

[Gros87] Grosz B.J. et. al.: *TEAM: An experiment in the Design of Transportable Natural Language Interfaces*. Artificial Intelligence, No. 32, 1987.

[Gugl92] Guglielmo E.: *Natural Language Processing of Captions for Retrieving Multimedia Data*. Proc. Conf. on Applied Natural Language Processing, Trento, Italy, 1992.

[HoLR90] Holtkamp B., Lum V. Y., Rowe N. C.: *Demon - A media object model incorporating natural language descriptions for retrieval support.* Tech. Report NPS52-90-019, Computer Science Department, Naval Postgraduate School, Monterey, CA,1990.

[KeKL91] Keim D. A., Kim K.-C., Lum V.: *A Friendly and Intelligent Approach to Data Retrieval in a Multimedia DBMS*. Proc. Int. Conf. on Database and Expert Systems (DEXA), Berlin, Germany, 1991.

[KeLu92] Keim D. A., Lum V.: Visual Query Specification in a Multimedia Database System, Proc. Conf. Visualization, CS Press, Los Alamitos, CA., 1992.

[KiMe84] King R., Melville S.: *Ski: A Semantics-Knowledgeable Interface*. Proc. Int. Conf. on Very Large Data Bases, Singapore, 1984.

[LuMe91] Lum V., Meyer-Wegener K.: *An Architecture for a Multimedia Database Management System Supporting Content Search*. Advances in Computing and Information - ICCI '90, Lecture Notes, Springer, 1991.

[LuKK92] Lum V., Keim D. A., Kim K.-C.: *Intelligent Natural Language Processing for Media Data Query*. Proc. Int. Golden West Conf. on Intelligent Systems, Reno, Nev., 1992.

[Pei90] Pei S.-C.: *Design and Implementation of a Multimedia DBMS: Catalog Management, Table Creation and Data Insertion*. M.S. Thesis, Computer Science Department, Naval Postgraduate School, Monterey, CA, 1990.

[RoCa88] Rogers T. R., Cattell R. G. G.: *Entity-Relationship Database User Interfaces* in Readings in Database Systems, ed. by M. Stonebraker, 1988.

[WoKu82] Wong H. K. T., Kuo I.: *GUIDE: Graphic User Interface for Database Exploration*. Proc. Int. Conf. on Very Large Data Bases, Mexico City, 1982.

[Zloo77] Zloof M.: *Query-By-Example: A Data Base Language*. IBM Systems Journal, No. 4, 1977.