Conceptual Modeling and Querying in Multi-media Databases

Chitta Baral, Graciela Gonzalez and Tran Son
Department of Computer Science
University of Texas at El Paso
El Paso, Texas 79968
U.S.A.
{*chitta,chelis,tson*}@*cs.utep.edu*

Abstract

In this paper we discuss some design principles that will aid in the design and querying of multi-media databases. We use an object-relational data model and argue that multi-media objects should normally have a special attribute called 'core' which stores the real object itself in contrast to the abstraction which is reflected in the rest of the attributes. We present an extension to the ER Diagram that takes advantage of the 'core' notion to facilitate design of multi-media databases. We discuss some desirable features in a query language for multi-media databases: simplifications like the use of path expressions and implicit use of functions (methods) as attributes, and explicit specification of the display layout and format either at the data definition level or query specification level. To materialize this last feature, we propose a display specification extension to SQL (SQL+D) that we have implemented.

1 Introduction

A Multi-media database (MMDB)[1] differs from a conventional database in that its content may consist of types that are not commonly found in conventional databases. A list of such data includes: pictures (in gif, in jpeg, etc.), sound clips (in au, etc.), movies (in mpeg, in quicktime, etc.), documents (even multi-media documents), applets, and text (in postscript, in dvi, in pdf, etc.).

With the availability of tremendous amount of digitized multi-media content (from the WWW to the CD-ROMs and video disks) and the increasing affordability of disk space in computers, people and organizations can now consider having multi-media databases. Such databases will make it easier to answer queries or to locate information about multi-media data, without involving physical effort of a person. For example, a multi-media database containing movies and information about them in a virtual video store (i.e., a huge video server) will not only integrate the steps of going to a video store, looking at the (limited) database about movies in the video store, and walking on the aisles looking for a particular movie (which may not be there because of misplacement or because it has not been re-stacked after it was returned), but also allow us to ask queries about the movies where the query itself has multi-media content. An example would be to give a clip of Joe Pesci in a particular movie and query about movies with 'similar' clips or similar 'dialogue delivery' by

[1] We are also concerned about representing and querying semi-structured and unstructured data. A list of such data includes: Emails, News postings (semistructured ascii); Programs in a computer language such as, C, C++, Java, etc. (semistructured ascii); Newspaper articles, and free text (almost unstructured).

Joe Pesci. Many such examples involving images, videos, maps, documents, audio clips, etc. can easily be developed.

In last couple of years the importance of multi-media databases has been recognized. The US Army and University of Maryland Institute of Advanced Computer Studies (UMIACS) organized workshops on multi-media information systems in 1995 (in Washington DC) and 1996 (West point, NY), and plan to organize another one in 1997 in Italy. A special issue of IEEE Computer (September 95, vol 28, number 9) has been devoted to multi-media databases involving images [Fea95, OS95, Sri95]. But to our knowledge most research in multi-media database design have focussed on a particular kind of multi-media data, and most of them are concerned with what relations are needed to adequately store *the content* of a particular multi-media database (with particular kind of multi-media content). For example, [OS95] is concerned with images, while [SYH94] is concerned with representing the spatial relationship among objects in pictures, and [MMM96] is concerned with querying the web. Another direction of research on multi-media databases is focussed on data structures and algorithms for storing and processing multimedia content [Fal96].

Our goal in this paper is to explore common design principles for multi-media databases in general – not for a particular multi-media database, or for multi-media databases with a particular kind of content. It seems to us that the time has come to extrapolate, extract and learn the common design principles from particular examples.

1.1 Designing multi-media databases

A multi-media database is different from a multi-media repository [2]; but it is obvious that it is usually built around multi-media repositories.

The question we now need to answer is how to build a multi-media database around a multi-media repository. A partial answer comes from the observation that conventional databases are also built around repositories. For example, the repository behind a supermarket database [Ull88] includes sales receipts, purchase orders, invoices, employment applications, and many other pieces of information.

The observation that both conventional and multi-media databases are built around repositories suggests that many of the techniques and methodologies used in designing conventional databases can be used (with appropriate enhancements and/or modifications) in designing multi-media databases. For example, one of the basis in the conceptual design [BCN92] of a conventional database is the kind of queries that the database is supposed to answer. We should use that same concept when designing a multi-media database around a multi-media repository. Similarly, we would need to consider the relationships among the data elements. An excellent tool that might also be ported over to multi-media database design is the use of an entity-relationship diagram, possibly with extensions to allow for the correct representation of multi-media features. *In a later section we describe such extensions in what we call* **core entity relationship model** *and use it in designing the schema for a multi-media database.*

1.2 Data models for multi-media databases

Considering that a multi-media database will either interact with conventional databases or will have a component which is of the conventional kind, we would like to use a data model that is upward compatible with the relational data model - the most widely used data model for conventional databases.

[2] A multi-media repository is basically a collection of multi-media artifacts, with very limited querying ability.

In many content-based approaches to query multi-media data, the content of the multi-media artifacts are abstracted as a relational database and hence normally do not need any new capability. But, let us consider the following quote:

> *A picture is worth thousand words.*

The implication of the above quote is that, even after we have extracted (manually, semi-automatically and/or automatically) the various features of a picture[3] and stored it in our database, we can never be sure that we have extracted *all*[4] the useful information. There is always a possibility that in the future we may need some additional information (expressed by the picture) which was not extracted from it earlier. Moreover, for some features it may be preferable to extract them during query answering time rather than extracting them before and storing them in a table.

This leads us to the conclusion that we can not just abstract information from a multi-media artifact into a relational database, but rather should have the provision to store the artifacts themselves, and should also be able to specify operations on the artifacts that can be executed at run-time. This requires types that allow multi-media data and the ability to define methods. For example, we should be able to define a relation of the form:

$EMPL(name, ssn, position, salary, dept, picture)$

where the attribute 'picture' could be of the type "gif-blob", and we should be able to define operations on pictures and possibly other attributes.

SQL3[5] [Bee93, MS] and many of the recent models in object-relational database products such as Illustra [Sto96], Odapter, UniSQL, DB2, Omniscience have this and additional features useful for multi-media databases. Moreover, with the current effort in object-oriented database standards such as ODMG-93 [Cat96] and the ongoing effort on SQL3/OQL merger [Wad96, BC95], we can also use object oriented data models that are upward compatible with the relational model or SQL-89.

2 Modeling MMDB using Object Relational Data Models

In this section we discuss through examples the design and representation of multi-media and semistructured data using object relational data models. We then identify some salient features common to modeling different multi-media data types. We first start with data description language and then move to the query language.

2.1 Data Description language

Example 1 Consider designing a relational database for movies, where we not only store information about movies but also store the movies themselves. Let us assume that for each movie we have information about its title, year, producer, director, length, filmtype, and prod_studio. This can be easily stored by having a relation called $MOVIE$ with attributes as given below.

[3] This can be generalized to any artifact, including multi-media artifact and artifacts abstracted in conventional databases.

[4] This is also true for conventional databases. For example, when the IRS investigates a corporation, it looks beyond the database of the company (like at other computer files, invoices, account statements, reports, and other documents) to find information not in the database.

[5] Most observations about SQL3 in this paper are based on articles about SQL3, i.e., based on secondary sources. We have been unable to obtain the authentic SQL3 document, although we did consult the SQL-92 book [MS92].

$MOVIE(Title, Year, Producer, Director, Length, Movie\_type, Prod\_studio)$

But what about the movie itself? Also, what about operators and functions that can be applied to the movie itself? (Such operations might include extracting the opening sequence of the movie, removing all occurances of the MacDonald arch in the movie, finding similarities between movies, overlaying text or sound in certain sequences of the movie, etc.)

The current ORDB models offer two approaches.

- *First Approach*: We can define a complex type called movie_t, as shown below in the notation of Illustra:

  ```
  create type movie_t(
  Title varchar(30),
  Year int,
  Producer varchar(30),
  Director varchar(30),
  Length int,
  Movie_type varchar(30),
  Prod_studio varchar(30),
  MV blob);

  create table movie of type movie_t;
  ```

  Now, we can define and register various user defined operators and functions for the type *movie_t*. The definition of these functions and operators may use the value of the attribute 'MV'.

  One possible[6] drawback of this approach is that we are completely moving away from the relational model and are using object-oriented model. In the process we are not taking advantage of storing the abstracted attributes (all attributes other than 'MV') of movie in relations.

- *Second Approach*:
  We can add an attribute of an extended type 'movie_blob' to the *Movie* relation. Various user defined operators and functions will be defined and registered for this extended type.

  Note that one of the main difference between this approach and the first approach is that in the first approach functions and operators are defined for the type movie_t, while in this approach they are defined for the type movie_blob, whose values does not include the abstracted attributes, such as Title, Producer, etc.

  There are two concerns:

  - An intuitive name for this attribute: The most intuitive name perhaps would be to call this attribute 'movie', but that would create confusion with the relation name 'Movie'. Suppose we find another intuitive name for this attribute. Say we call it 'MV'.
  - There will be always some confusion between the relation 'Movie' and the object 'Movie.MV'. Since Movie.MV refers to a movie, a user may incorrectly refer to the title of Movie.MV by using the expression Movie.MV.title instead of the correct expression Movie.title.

[6] With merger of object-oriented and relational models [Wad96] this may not remain a drawback.

> The confusion arises because of two ways to look at a movie object; the first is the digitized movie itself, and the second is the abstracted information about the movie.

Actually, the confusion between names will also arise in the first approach. For example, for a movie object with object id $m123$, there could be confusion between $m123$ and $m123.MV$.

Based on the above observations, we would like to propose that objects in general and multi-media objects in particular, whether they are represented in the object-oriented style (as in the first approach) or in a relational style (as in the second approach, where the object is a value ADT in the syntax of SQL3), should be allowed to have a special attribute[7]*, say called,* **'core'**. *We propose that this word be a reserved word in the data description language, and used across all objects.* By using it across all objects the users would become familiar with it and this would eliminate the confusion discussed earlier. Note that we are not requiring that this should be part of the type definition of all objects. We are saying that if the type definition of an object would like to use an attribute to represent the digitized object itself then it should use a particular name, perhaps 'core', which other objects may also use. Of course the type definition of an 'employee' object may not have this attribute, as we can not[8] digitize the employee and store it in our database. We hope that the standards committee of SQL3 and ODMG will consider our suggestion.

Example 2 Following our own suggestion we add the special attribute 'Core' to the 'Movie' relation. In this particular example, the attribute 'Core' has the type as 'movie_blob'. After this enhancement, the 'MOVIE' relation becomes

$MOVIE(Title, Year, Producer, Director, Length, Movie\_type, Prod\_studio, Core)$

This is the view that the data entry person has about the data. But we allow users who may query the database to have a object-oriented view of the database. For example, the query: "Show the opening sequence of all movies produced by Spielberg in 1993" *should be permitted* to be expressed in a SQL-like language as follows:

```
SELECT Movie.opening_seq
FROM Movie
WHERE Year = 1993 AND Director = "Spielberg"
```

Note that the relation, Movie does not have an attribute called opening_seq. It is a function defined on values of type 'movieblob' (which is the type of the attribute Core) and returns values of the same type. This can be described in SQL-3 as follows:

```
DECLARE EXTERNAL opening_seq(movie_blob)
    RETURNS movie_blob
LANGUAGE C;
```

The meaning of the above is that, the function opening_seq is defined on objects of the type movie_blob and the value of this function is also of the type movie_blob. Also, to obtain the value of the function opening_seq on any object of the type movie_blob, we need to apply the program opening_seq.c to that object.

In summary, since users will be uniformly using the word 'Core' in the data definition language, they are permitted to leave it out in the queries, and the query interpreter is programmed to look at the attribute 'Core' if necessary. In the above example, the query interpreter after failing to find opening_seq defined on the value ADT Movie, knows to look at Movie.Core.

[7] Another possible name for this attribute could be 'self'.

[8] Unless we are talking about science fiction.

The Movie relation above and other relations that have the special attribute 'Core' will be referred to as core-relations, and we will call the corresponding schema as a *Core-ORDB schema.*

Example 3 Another example of a core-relation is:

$EMAIL(From, To, Date, Size, Core, Subject)$

In this relation all attributes except 'Core' are of standard types. The type of the attribute 'Core' is an extended type and is called 'ascii_email'.

The query, "List the emails of John that contain the word "tenure" can be expressed in a SQL like language as follows:

```
SELECT Email
FROM Email
WHERE Email.Keyword["tenure"] = TRUE
```

The declaration about the function Keyword can be expressed in SQL-3 in the following way:

```
DECLARE EXTERNAL Keyword(email_ascii,charstring)
    RETURNS boolean
LANGUAGE C;
```

At present we prefer core-relations (because of its relational nature) for modeling multi-media data and in the rest of the paper we will be following the style of the second approach described in Example 1 instead of the first approach. Unlike the data model in Illustra, our core-relations can also be viewed as object classes. This is similar to row-types in SQL-3.

2.2 Query Language

In this section we discuss two aspects that we would like to see in a query language for multi-media databases. One aspect is related to our discussion in the previous section, and the other aspect is related to the SQL3 data model we use. *Both aspects are concerned with making it easier for a user to frame a query.* Since we discussed in the previous section how to simplify queries using 'core', we now discuss the second aspect.

2.2.1 Query simplification using subtable information

In SQL3 tables are allowed to have subtables. For example, we can specify in SQL3:

CREATE TABLE manager UNDER employee

In Examples 1 and 3 we saw how our use of core-relations makes it easier to express queries. We now give some examples that show how we can take advantage of UNDER relation between tables and take it one step further than SQL3 to simplify queries.

Example 4 Consider the following schema:

$EMPS(ename, salary, dname)$
$DEPTS(dname, d\#, manager)$

and the additional information:

$MANAGER$ is a subtable of the table $EMPS$
$DEPTS.manager$ is of type $MANAGER$.

Suppose we would like to ask the query: "Find the salary of the manager of department number 23". An intuitive way to express this query in a SQL-like syntax will be the following query which we refer to as Q_A:

SELECT DEPTS.manager.salary
FROM DEPTS
WHERE DEPTS.D# = 23.

In the absence of the subtable information this query is not meaningful, and we would have to think of the attribute 'manager' of the $DEPTS$ relation as a simple attribute to store the name of the manager of the department, which would then be used as a way to get the salary in the $EMPS$ tuples. The 'regular' version of the above query would be the following query Q_B:

SELECT EMPS.salary
FROM DEPTS, EMPS
WHERE DEPTS.D# = 23 AND EMPS.ename=DEPTS.manager

We will use our data model, particularly the subtable information to present an algorithm that will translate Q_A to Q_B.

To reinforce our point, let us enhance our schema with the additional relation PERS(ssn, name) and subclass information:

$EMPS$ is a subtable of $PERS$.

The query asking to list the social security number of the manager of department number 23 can be expressed in an intuitive manner by the following query Q_C:

SELECT DEPTS.manager.ssn
FROM DEPTS
WHERE DEPTS.D# = 23

In the absence of the subclass information this query is not meaningful, and again, we would have to make the 'manager' attribute store the manager's name. The meaningful version of it would be the following query Q_D:

SELECT PERS.ssn
FROM DEPTS, PERS
WHERE DEPTS.D# = 23 AND PERS.name=DEPTS.manager

2.2.2 Query Simplification: Summary

The two suggestions regarding query simplification in the previous sections: the use of path expressions and the implicit use of functions as attributes, do not exactly follow SQL[9] in two major respects.

- In Example 1 and 3 we have the queries

  SELECT Movie.opening_seq()
  FROM Movie
  WHERE Year = 1993 AND Director = "Spielberg"

  SELECT Email
  FROM Email
  WHERE Email.Keyword("tenure") = TRUE.

  where, opening_seq is neither an attribute of the relation Movie, nor it is a function (strictly) defined on elements of Movie. Rather it is a function defined on Movie.Core. Similarly the function Keyword is a function defined on Email.Core and character strings.

[9] Here we are referring to SQL-92 and before. SQL3 does not have the first simplification. It is possible that it has the second.

- In Example 4, the queries Q_A and Q_C, needs to be translated to Q_B and Q_D to be meaningful in terms of SQL.

The reason we allow the above two shortcuts is to make it easier for the user to ask a query. For example, in Q_A, the user wants to know the salary of the manager of a particular department. An intuitive path expression that the user can construct without completely knowing the Database schema corresponding to it (the user's needs) is department.manager.salary

One of the reasons object-oriented database are attractive is because the data is supposed to be arranged in an intuitive manner[10] which allows the user to ask questions without completely knowing the schema. (This is impossible in a relational database.) We would like to retain and take advantage of as much of this useful feature of object-oriented databases as possible.

We now explain the process that converts queries with shortcuts to SQL queries.

Given a path expression a.p, if a neither has an attribute called p, nor has a function called p whose input is of the type a, then there are two choices[11]

- The 'a' in a.p has an attribute called 'Core' on which p is defined. In that case we replace a.p by a.Core.p in the whole query.

  It is easy to see that we use this step in Example 1 and 3.

- We have the clauses a is a subtable of a_1, a_1 is a subtable of a_2, ..., a_{n-1} is a subtable of a_n in our schema such that p is either an attribute of a_n or is defined on a_n, and p is neither an attribute and nor it is defined on $a_1, \ldots, a_{n-1}$. In that case we replace $a.p$ by $a_n.p$ in the query, add $a.k = a_n.k$ (where, k is the key of the relations $a_1, \ldots, a_n$) to the WHERE clause of the query and add a_n to the FROM clause of the query.

  It is easy to see that we use this step (with slight generalization) in translating Q_A to Q_B and Q_C to Q_D.

In the above we assume that a table is not a subtable w.r.t. two different tables.

2.3 Semantics Issues: Can relations be object classes?

It has been pointed out, particularly by Date and Darwen [DD95] that semantical problems arise when relations are treated as object classes. For example, Consider a relation P with attributes W and V, and a relation Q with attribute X, Y and Z. Let us also assume that the type of W is integer and the type of V is the class corresponding to relation Q. Now suppose the tuples of relation Q are $\{(a, b, c), (d, e, f)\}$. The question is what does a tuple of the relation P look like. Is (1, a, b, c) a possible tuple of P? The answer is no, since P only has two attributes.

We can avoid the above problem by associating with each relation R, a class C_R, and a constructor f_R. We say $(a_1, \ldots, a_n)$ is a tuple of R iff $f_R(a_1, \ldots, a_n)$ is a member of the class C_R.

With respect to relations P and Q, $(1, a, b, c)$ is not a possible tuple of P, while $(1, f_Q(a, b, c))$ is.

[10] For example, Kifer, Lausen and Wu in [KLW93] say the following to contrast relational databases with object-oriented database:
"The central feature of the relational data model is that data is conceptually grouped by properties. For instance, information regarding a given person may be scattered among different relations, such as employee, manager, and project. In contrast, object-oriented representation seeks to group data around objects... and the user can access, directly or by inheritance, all public information about any object once a 'handle' for the object is obtained."

[11] We assume that our language is designed such that both choices are not applicable.

When we just say that relations can be thought of as classes we are taking liberty with the syntax. This is fine as long as the semantics is made clear. That is what we have tried to do here. We would like to point out that similar concerns have been expressed during the discussion [BC95, Wad96] regarding merging of OQL and SQL3.

3 A design methodology for MMDB

In this section we describe a methodology for designing multi-media and semi-structured databases, based on the approach of using ER diagrams as a first step in designing relational databases.

First, we will introduce an extension of the ER diagram, called *Core-ER diagram*, for the conceptual design of multi-media and semi-structured databases. Then we describe how to to obtain a Core-ORDB schema from the Core-ER (CER) diagram.

3.1 Core-ER diagram for Multi-media and Semistructured databases

We extend the ER diagram to account for multi-media and semi-structured databases. We call it a CER diagram. The CER diagram is an extension of ER diagrams, and hence has has all the features[12] of ER diagrams. The main addition is to the notion of entities. For example, an entity **Movie** has attributes such as title, producer, director, etc., that correspond to the abstract concept of a movie. But at the same time it also has a blob which is the movie itself. In our extension entities may have an *abstract* part and a *real* part. We are now ready to describe a CER diagram.

1. *Entities:*

   Entities are represented by either a rectangle or by a rectangle containing an oval. We call the first "abstract entities", and correspond to entities in traditional ER diagrams, and the second "core entities". In this case, the outer enclosure (rectangle) corresponds to the original meaning of an entity in an ER diagram, while the inner enclosure (oval) represents the *real or the core entity.* In Figure 1, **Movie** is a **core entity**. The entity **Employee** in Figure 2 is an **abstract entity**. We would say simply "entity" or "entities" when something applies to either a "core entity" or an "abstract entity", or when the actual nature of the entity is clear by its context.

   The intuition behind this distinction is simple: for abstract entities, such as **Employee** in Figure 2, we are not storing the real employee in our database, while for core entities (which might also be called "real entities"), we do store the actual (digitized) entity. Intuitively, any object (entity) that is in digital format may be represented as a core entity.

2. *Attributes*

   - *Conventional attributes* are represented as in ER diagrams. In other words, they are represented by an oval associated to the outer closure of an entity. All attributes of the entity **Movie** in Figure 1 are of this type.
   - *Unconventional attributes* of an entity are represented by an oval inside a dotted rectangle. The attribute **Picture** of the entity **Employee** in Figure 2 is of this type. The dotted rectangle suggest that the picture has no abstractions. In presence of abstractions it is like another entity.

[12] For brevity we do not discuss weak entities. We intend to treat them as done in ER diagrams with minor changes.

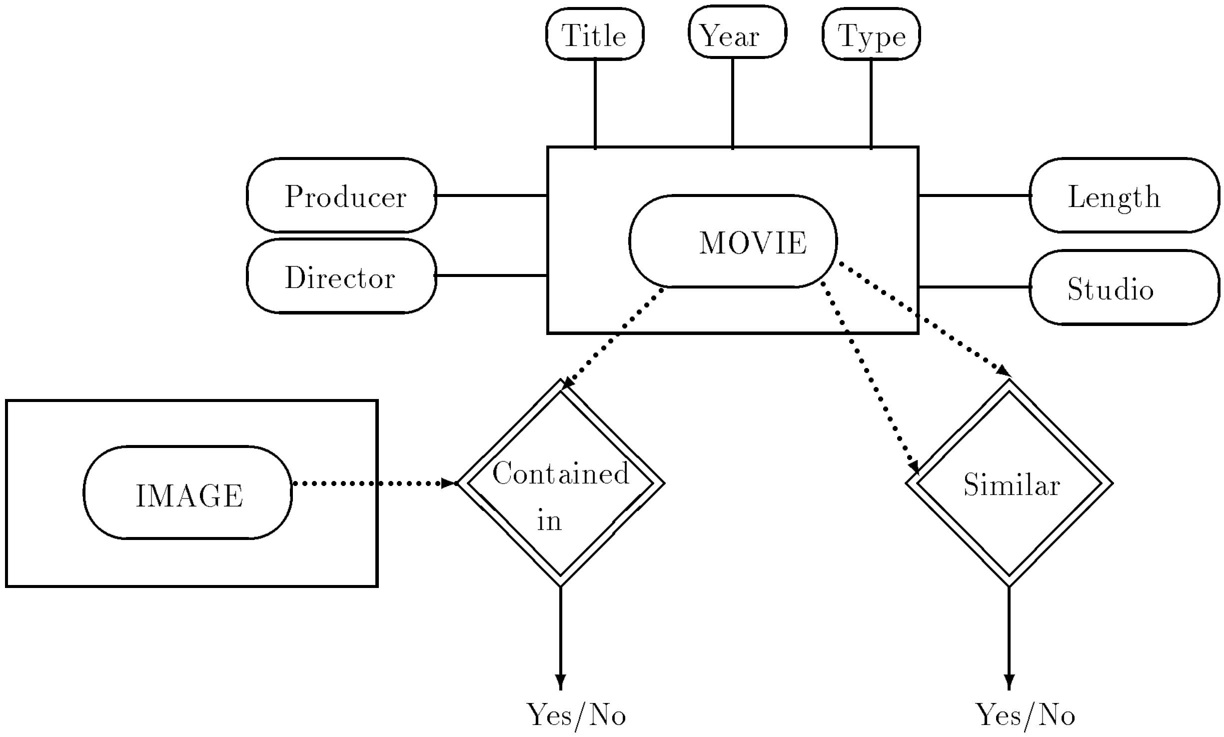

Figure 1: The **Movie** core entity

3. *Methods* (also referred to as functions or operators) are represented by diamonds (with a double enclosure, as opposed to diamonds with single enclosure for relationships) which have dotted arrows coming in and one solid arrow going out from it. The arrows represent the inputs and output of a method, respectively. The incoming arrows may either come from the inner enclosure (oval) of an entity or may come from the outer enclosure (rectangle) of an entity. When they come from the inner enclosure it means, that method uses the real entity as an input. For example in case of Figure 1, the method Contained_in uses the movie itself as one of the inputs. On the other hand, we may have a method *similar_credential* (not shown in Figure 1) that takes input as the abstract attributes of two members of the entity movie and outputs a boolean value. A definition of this method could be: "Two movies have *similar credentials* if they have the same director, they were made in the same decade, either their producer or their studio is the same, and they are of the same Type." For this method the input arrows will come from the outer enclosure of the entity **Movie**.

4. *Keys and Relationships:* Keys and relationships are represented exactly as in ER diagrams. (Note that to easily distinguish between relationships and methods, we use diamonds with one enclosure for relationships, and two enclosures for methods. Also, incoming arrows to methods are dotted while all connections to relations are solid.)

Notice the basic difference between methods and attributes: values of attributes are stored and the values returned by methods are not stored; rather they are evaluated at runtime.

Example 5 The CER diagram consisting of the **Employee** and **Email** tables and a 1-to-n relationship between Employee and Email is given in Figure 3.

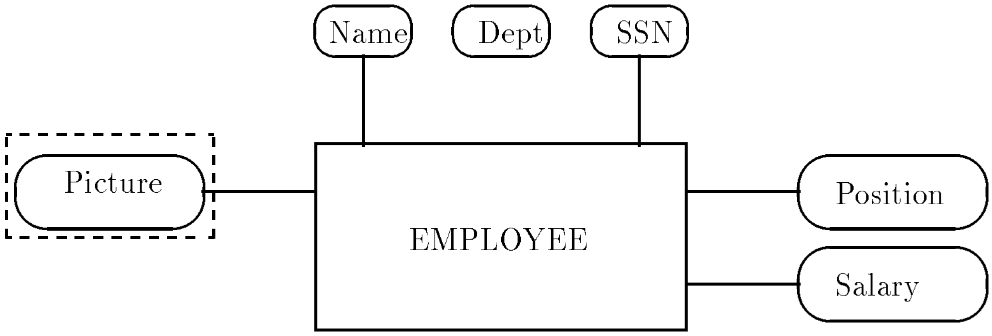

Figure 2: The **Employee** abstract entity

3.2 From a CER diagram to a Core-ORDB schema

It is obvious that if we remove the additional features (w.r.t. an ER diagram) from a CER diagram we obtain an ER-diagram. We refer to it as the *embedded ER diagram* of a CER diagram. We now present an algorithm to construct a Core-ORDB schema from an CER diagram.

1. For each abstract entity create a relation with its associated attributes as is done when translating an ER diagram. Unconventional attributes are treated the same way as conventional ones. For example, from Figure 3 we create the relation *EMPLOYEE(Name, Dept, SSN, Position, Salary, Picture)* corresponding to the entity EMPLOYEE.

2. For each entity represented as a rectangle containing an oval create a relation with its associated attributes together with one additional attribute named *Core*. For example, the relation *EMAIL(emailID, from, to, size, date, resp_to, date_resp, Core)* is created corresponding to the entity EMAIL.

3. For each double diamond create a method definition[13] using as input the starting point of the incoming dotted lines, be it a core entity or one or more attributes of the abstraction of a core or abstract entity [14]. For example, the following method definition is created for the entity EMAIL.

   FUNCTION Keyword(email,charstring)
   RETURNS boolean
   <function definition>;

4. Obtain the embedded ER diagram from the given CER diagram and create tables corresponding to the non-ISA relations in the standard way. For the CER diagram in Figure 3, there is only one relationship: HAS(SSN, emailID) where emailID is the key of the relation EMAIL.

[13]Note that by using CER diagram instead of ODL definitions, we not only make the schema visual but also we do not have to repeat methods with respect to each object type whose elements can be one of the inputs to the method.

[14]Here, the abstraction that can be depicted reflects only the distinction between having as input the core entity or one or more of its attributes. For simplicity in the diagram, exactly which attributes from the abstraction are passed is not specified. This would need to be defined when the diagram is translated to actual functions

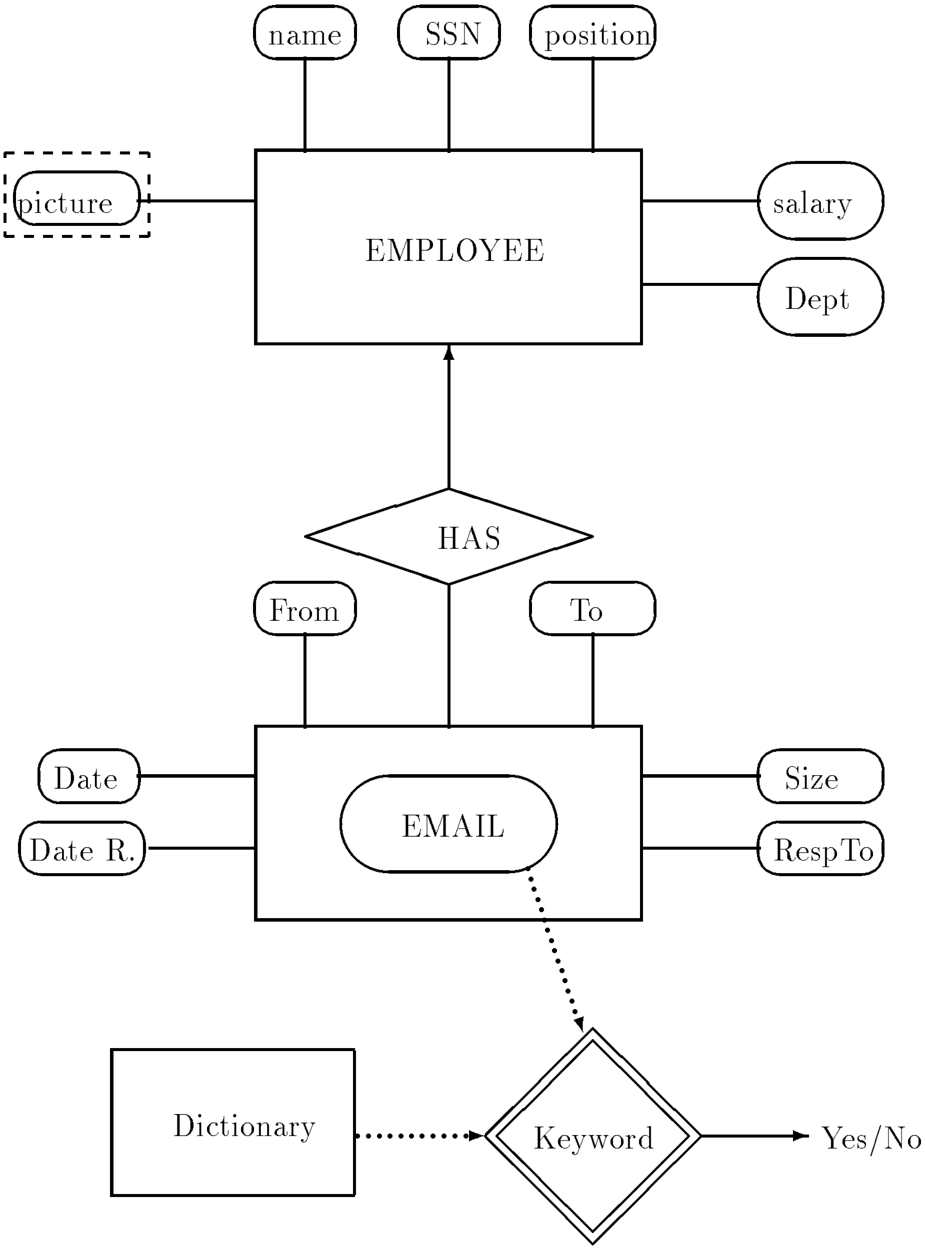

Figure 3: CER diagram for employees and their emails

5. For each IS_A relationship from an entity E_1 to an entity E_2, E_1 is a subtable of E_2.

6. Keys are defined as in the embedded ER diagram, except that for entities with two enclosures the attribute 'Core' (normally containing a pointer to the real entity) can be a key.

So, the complete Core-ORDB schema for the CER diagram in Figure 3 is as follows.

EMPLOYEE(Name, Dept, SSN, Position, Salary, Picture)

EMAIL(emailID, From, To, Date, Size, Resp_To, Date_Resp, Core)

HAS(SSN, emailID)

FUNCTION Keyword(email,charstring)
    RETURNS boolean
<function definition>;

Similarly, the CER diagram in Figure 1 can be translated into the following core-ORDB schema:

MOVIE(Title, Year, Type, Producer, Director, Studio, Length, Core)

IMAGE(Core)

FUNCTION similar(movie1:movieblob, movie2:movieblob)[15]
    RETURNS boolean
<function definition>;

FUNCTION contained_in(picture:gifblob, movie:movieblob)
    RETURNS boolean
<function definition>;

4 Modeling the WWW as a multi-media database

In this section we show how our conceptual modeling methodology can be used to model the WWW as a multi-media database. We start with a CER diagram that is very close to the model used in [MMM96].

The CER diagram in Figure 4 has two entities, one of which is a core-entity. There is one relation and four methods. Note the distinction between the methods *contains* and *mentions*. The input arrows of the method *contains* comes from the outer enclosure of the entity HTMLdoc and from 'String', while the input arrows of the method mentions comes from the inner enclosure (the core) of the entity HTMLdoc and from 'String'. The difference is that the method 'mentions' uses the 'core' while the method 'contains' uses one of the attributes of the abstraction: the title.

The Core-ORDB schema corresponding to the CER diagram in Figure 4 has the following relations:

HTMLdoc(h_url,title,type,length,lastModify,core)[16]

Links(l_url,label)

Include(h_url,l_url)

and the following methods:

- **contains(HTMLDoc.Title, string)**: This method takes a string and the title of an HTML-doc and finds out if the title contains the given string.

[15] We do not actually get the data types of movie1 and movie2 from the diagram, but from knowledge about the nature of the core entity *MOVIE* that the database designer has. We include the data types here for clarity.

[16] We are using h_url and later l_url to distinguish the attributes HTMLdoc.url and Links.utl, respectively.

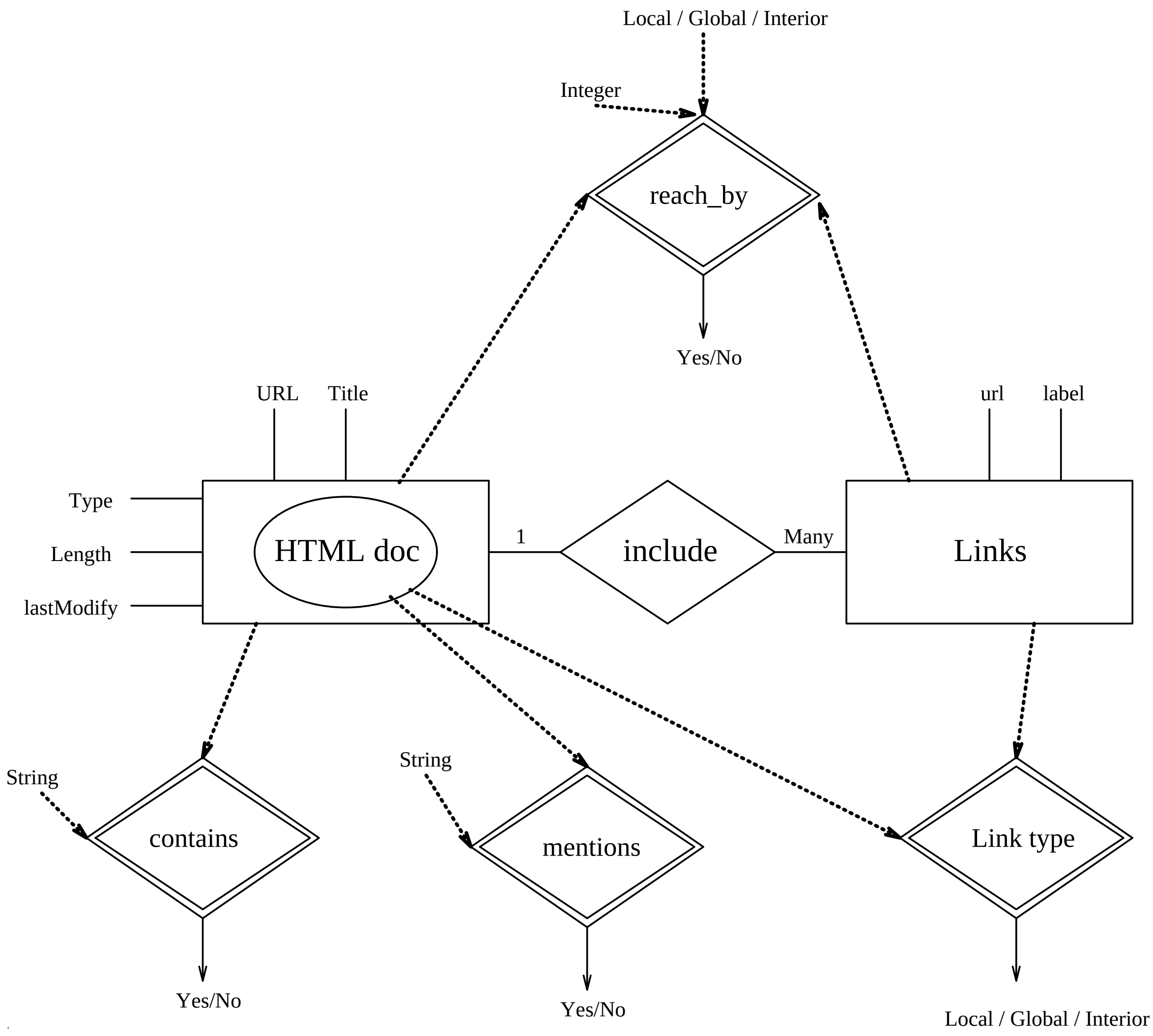

Figure 4: CER diagram modeling WebSQL

- **reach_by(HTMLDoc.url, url_to, by_n, l_type)**: Here, 'HTMLDoc.url' is the url corresponding to the HTML document and 'url_to' is a url; 'by_n' is an integer and 'l_type' can have the value, local, global or interior. This methods finds out if the url given by *url_to* can be reached from HTMLDoc.url given by after traversing at most *by_n* links of the type given by *l_type*.
- **mentions(HTMLdoc, string)**: This method takes a string and an HTMLdoc and finds out if the HTMLdoc (its core) contains the given string.
- **linktype(HTMLdoc, url)**: This method returns which type of link is the link specified by *url* w.r.t *HTMLdoc* (if any).

We now show how some of the queries from [MMM96] can be expressed in an object-relational query language with respect to the above schema. We first give the query in English and then it representation in an object-relational query language[17].

[17] Here we are following the notation of Illustra.

1. Find all HTML documents about "hypertext"

```
SELECT  HTMLdoc.h_url
FROM    HTMLdoc
WHERE   contains(HTMLdoc.title, "hypertext")
```

2. Find all links to applet from documents about "java"

```
SELECT  Links.l_url
FROM    HTMLdoc, Links, Include
WHERE   mentions(HTMLdoc, "java")
AND     HTMLdoc.h_url = Include.h_url
AND     Links.l_url = Include.l_url
AND     substring("applet", Links.label)
```

   Note that *substring(instring, string)* is a standard method that searches for the occurrence of *instring* in *string.*

3. Starting from the Computer Science home page, find all documents that are linked through paths of lengths two or less containing only local links. Keep only the documents containing the string "database" in their title.

```
SELECT  Links.l_url
FROM    HTMLdoc, Links, Include
WHERE   substring("database", HTMLdoc.title)
AND     HTMLdoc.h_url = Include.h_url
AND     Links.l_url = Include.l_url
AND     reach_by("http://cs.utep.edu", Links.l_url, 2, local)
```

4. Find all documents mentioning "Computer Science" and all documents that are linked to them through paths of lengths two or less containing only local links.

```
SELECT  HTMLdoc.h_url, Links.l_url
FROM    HTMLdoc, Links
WHERE   mentions(HTMLdoc, "Computer Science")
AND     reach_by(HTMLdoc.h_url, Links.l_url, 2, local)
AND     linktype(HTMLdoc.h_url, Links.l_url) = local
```

Note that we can use the 'recursion' feature of SQL3 to express queries that require the transitive closure of 'reach_by'.

In [MMM96] the authors describe the query language WebSQL to express the above queries. The difference between our approach and the approach in [MMM96] is that we use a general data model and a general query language. We must note that the WebSQL described in [MMM96] is very useful and is implemented and we are not immediately planning to implement an alternative querying mechanism. We are only showing how the conceptual modeling methodology of this paper can be used to model the WWW as a database.

As in [MMM96] the relations in our schema corresponding to Figure 4 are virtual. The methods are supposed to be defined in the client site from where the query is submitted. However, in the future we can envision a combination of virtual and real schema, where some real data about the web pages may be specified as meta-information by the web masters of each site (and perhaps collected by some index servers), being used by a standard database query language (some extension of SQL). This is what we are advocating.

To give a flavor of our approach, we now extend the CER diagram in Figure 4 to model additional features that will allow us to specify more sophisticated queries. The first extension is the CER diagram in Figure 5.

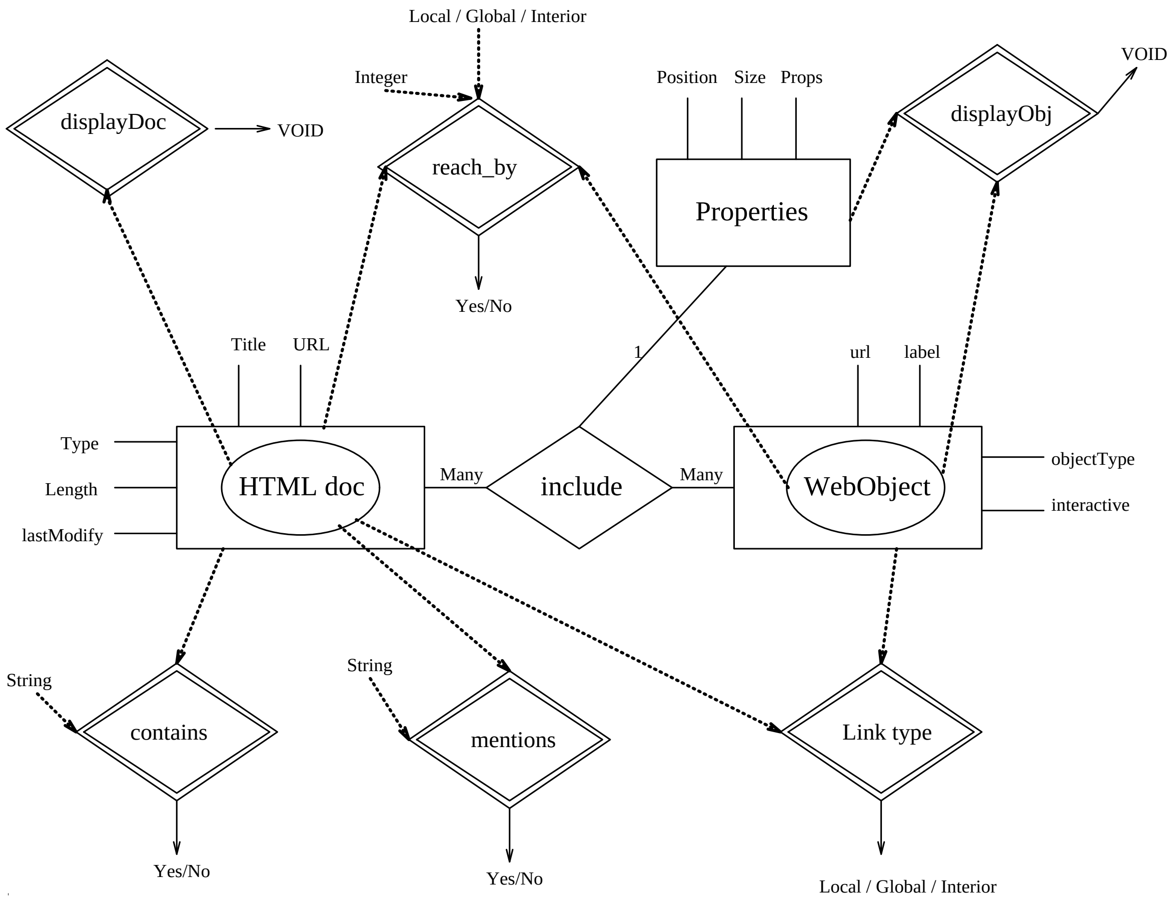

Figure 5: CER diagram with 'uniform' WebObjects

The relations in the schema corresponding to the CER in Figure 5 are:

HTMLdoc(h_url,title,type,length,lastModify,core)

WebObject(w_url,label,objectType,interactive,core)

Properties(position,size,props)

The properties relation is used to store the properties of a WebObject in an HTML document such as if it is an icon, a picture, a video or audio clip, a picture with a link etc.; and the size and position of the WebObject w.r.t. the document. The relation *include* now has additional attributes - the properties of an object in an HTML document, and it is now a many-many-1 relation.

include(h_url,w_url,position,size,props)

We now have two additional methods:

- **displayDoc(HTMLdoc):** This methods displays an HTMLdoc.
- **displayObj(WebObject, Properties.position, Properties.size, Properties.props):** This method displays an web object in the position given by *position*, with size given by *size* and with properties given by *props*.

Using the above schema (corresponding to the CER diagram in Figure 5) we can specify many additional queries such as:

- List all documents that have interactive web objects in them.
- List all documents that use the web object in *url* as an icon and also as a hot link.
- List all documents that have audio and video clips.
- List all documents that have a video clip or a picture labeled 'Clinton'.

The last query can be expressed as follow:

```
SELECT   HTMLdoc.h_url
FROM     HTMLdoc, WebObject, Include
WHERE    HTMLdoc.h_url = Include.h_url
AND      WebObject.w_url = Include.w_url
AND      (WebObject.objectType = "IMAGE"
  OR       WebObject.objectType = "VIDEO")
AND      WebObject.label = "Clinton"
```

The CER diagram in Figure 5 does not completely distinguish between the different kinds of web objects. It uniformly[18] abstracts all the different type of web objects. This is similar to the approach in [MS96] and the detailed abstraction in it ([MS96]) can be easily added to Figure 5.

To distinguish between different kinds of web objects we can introduce various subclasses to the class of web-objects. This is modeled as a CER diagram in Figure 6. These subclasses may have abstractions and methods specifically applicable to them. In Figure 6 we have only shown some of the attributes of the entity IMAGE (a subclass of WebObject).

We could have methods such as the method 'Contained in' in Figure 1 with inputs from IMAGE and VIDEO. We could further introduce classes such as 'aeroplane_image' as a subclass of 'IMAGE' and so on. The possibilities of enhancing the CER diagram in Figure 6 is endless and our approach provides a conceptual tool to do that.

[18] V. S. Subrahmanian introduces the principles of autonomy, uniformity and hybrid organization for organizing multi-media data in his forthcoming book on multi-media databases. While the CER diagram in Figure 5 corresponds to his principle of uniformity, the CER diagram in Figure 6 corresponds to hybrid organization.

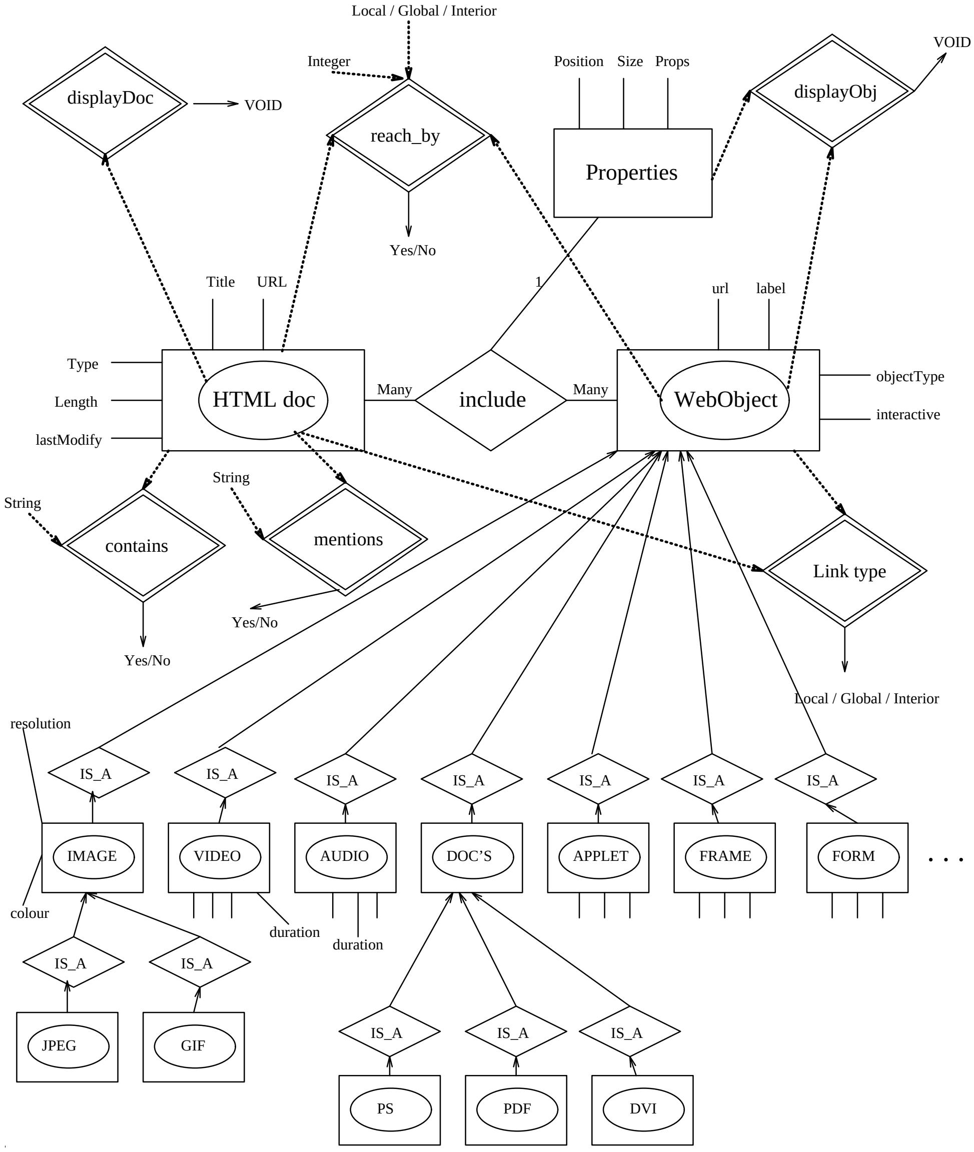

Figure 6: CER diagram with subclasses of WebObjects

5 Displaying answers

One question that we have not addressed so far is how the answer to a query submitted to a multi-media database is presented to the user. In conventional databases, either the answer is presented as a table or the user writes display specifications in the embedding language (Visual Basic, C++, etc) or in a report writing language. We believe that since multi-media databases may have web like front ends we should provide a simpler way for the user to specify display specifications.

We consider two different approaches: (i) Objects in general and multi-media objects in particular have a 'display' specification as part of the data definition. (ii) The display specification of the answer to a query is included with the query. The two approaches can work together if we assume that, if a display specification is given in the query, then it is followed, and if no display specification is given in the query, then the 'display' specification in the type definition of the object is used by default.

5.1 Display specification as part of the data definition

We suggest that 'display' be a reserved word in the data definition language which will be used in type or table definitions as a method that displays the corresponding object (or row object). Like other methods a type may inherit 'display' from its super types. In response to a query, even if 'display' is not mentioned in the query (as described in the next subsection) the final answer is passed on to the appropriate 'display' method to be displayed properly. As to exactly how this is implemented, it depends on the actual DDL implementation, and our idea is that it will follow the standard method definition of the DDL in question.

5.2 Display specification bundled with the query

We have designed and implemented a display specification language extension to SQL (SQL+D) [BGS97] that will allow us to specify how the answer of a query posed to a multi-media database should be displayed. The whole language specification and architecture, and a comparative view of it in light of other efforts is too large to be part of this paper. We present here a few examples of display specifications using SQL+D to give a flavor of our approach.

Example 6 *Consider a database for a video rental store containing movie titles and other general information of the movies, plus a movie clip and a picture of the promotional poster. Also available is a list of the actors in a movie, and other information about the actors, including their picture. The schema looks as follows:*

MOVIE	*(available, title, director, producer, date, classification, rating, Core, poster)*
MOVIE_ACTORS	*(title, name, role)*
ACTORS	*(name, dob, biography, picture)*

where all the attributes of **MOVIE_ACTORS** *and all the attributes of* **MOVIE** *are character strings, with the exception of Core, which is a video (mpeg or avi), and poster, which is an image (in tif, gif, jpeg or any other standard image format). The attributes name, dob and biography in* **ACTORS** *are also character strings, biography being a long (memo) field, and picture is an image.*

We would like to pose the following queries:

1. List all actors in "Gone With the Wind" with their pictures and biographies.
2. Display all action movies and their posters, allowing to play the movie if desired.

The first query would look as follows after it is translated to **SQL+D** :

```
SELECT    MOVIE_ACTORS.name,
          ACTORS.biography,
          ACTORS.picture
FROM      MOVIE_ACTORS, ACTORS
WHERE     MOVIE_ACTORS.title = "Gone With the Wind" AND
          ACTORS.name = MOVIE_ACTORS.name
DISPLAY   PANEL main, PANEL info ON main(east)
WITH      MOVIE_ACTORS.name AS list ON main(west),
          ACTORS.picture AS image ON info(north),
          ACTORS.biography AS text ON info(south).
```

Query 1. Querying the actors in "Gone With the Wind"

This is a standard SQL query up to the **WHERE** clause, thereafter the **DISPLAY** clause is used to specify where data is to be placed on the screen. Since the DISPLAY clause operates on the **data extracted** from the query, only the attribute names included in the **SELECT** clause can be used inside the **DISPLAY** clause.

The list of **PANEL** declarations following the **DISPLAY** keyword constitute the layout statement, and determine the overall layout of the screen: PANEL main is always the first and represents the whole screen or window where the query answer is being displayed, and the panels that follow (like "info", in the above example) are positioned within "main" or within other panels as specified by the **ON** subclause (in the example, the panel "info" is positioned on the west side of "main"). The panels listed here will contain different data display elements, specified in the **WITH** statement that follows.

The **WITH** statement contains a list of display specifications, indicating the kind of display element (like image, text, button, and others) desired for each attribute in the SELECT clause and their relative location within the panels previously defined in the layout statement. By default,if the user does not specify how to display all the attributes, the values of the attributes that have no display specification are grouped and displayed as a table in the south-most and east-most panel (lower right).

This query will display, as shown in Figure 7, a scrollable list of the actors on the west half of the screen, and a panel on the east half containing a picture at the top and the actor's biography at the bottom. The picture and biography displayed correspond to the currently selected tuple from the answer set.

The "currently selected tuple", is in a sense a pointer, which at first points at the first tuple in the answer set. Thereafter, the user can move through the answer set by using one of the four master control buttons displayed on the main panel: one each to move the pointer to the previous tuple and the next tuple, plus two that allow jumping to the first or last tuple in the answer set. The tuples are sorted by the attribute indicated by the user in the (regular) SQL "ORDER BY" and/or "GROUP BY" clauses, if any.

In addition, there are ways to control navigation via the displayed data directly, as is the case of a list or through the use of the keywords **ALL** or **ALL DISTINCT**, explained later. In a list, the value of all the tuples for a particular attribute are listed (the names of the actors, in the above example), and the user can select any tuple simply by clicking on a particular item on the list. If this is done, the pointer in the answer set is moved to the tuple that contains the selected attribute.

The user has visual cues in all the displayed elements that indicate which is the selected tuple. For example, in a list, the current tuple is highlighted in reverse video.

Figure 7: Display generated by Query 1

It is a good practice to display the key attributes of the relation as a list, whenever possible, to make navigation easier. That way, the user can see some portion of the data in advance and go directly to the tuple of interest, instead of blindly going tuple by tuple until the correct one is found.

The second query, to display all action movies and their posters, allowing to play the movie **if desired**, will be written as follows in **SQL+D** :

SELECT title, poster, core
FROM MOVIE
WHERE classification = "Action"
DISPLAY PANEL main, PANEL b ON main(*east*)
WITH poster AS *image* ON main(*west*),
title AS *list* ON b(*north*),
"Play" AS *button* ON b(south)
TRIGGER 'mpegplay('+ core +')'

Query 2. Query to see the poster of action movies and play a selected movie at will.

The display corresponding to this query is shown in Figure 8. The north, south, east, and west locations are sized automatically to fit the data displayed on them.

Since the button labeled "Play" has a **TRIGGER** associated to it, double clicking on the button causes an associated command to be executed. In **SQL+D** , the associated command is indicated in the **TRIGGER** portion of the display specification.

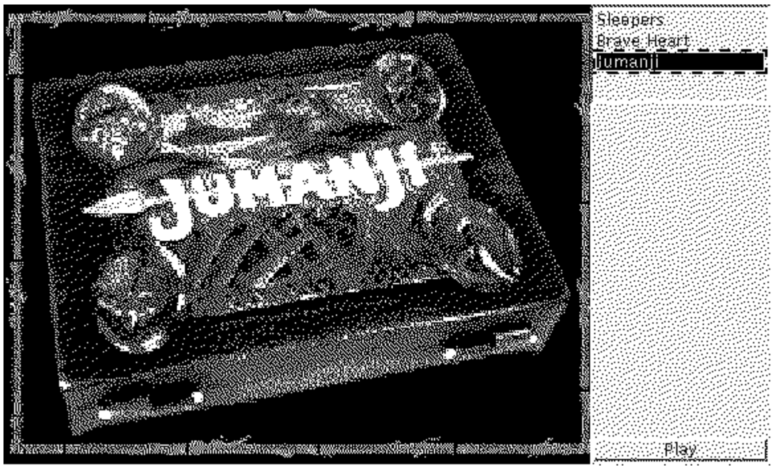

Figure 8: Display generated by Query 2

A **TRIGGER** associates either a query or an executable program to a particular attribute or display element. The associated query or application is invoked when the element that contains the **TRIGGER** is double clicked. The **TRIGGER** keyword follows an attribute or fixed label specification and is followed by the name of a stored query or an executable ("mpegplay", in the above example) and a list of parameters to be passed to it (the value of the attribute video_clip, above).

For example, if the user were to double click on the "Play" button while "Jumanji" was highlighted on the list, the following command will be triggered:

mpegplay(jumanji.mpeg)

assuming the value of the video_clip attribute of the tuple that contains "Jumanji" is "jumanji.mpeg".

When database attributes are listed as parameters to queries or executables, the value of the attribute(s) when the TRIGGER is invoked is(are) passed to the associated query or executable. The value of the attribute is determined by the current tuple selected in the display.

The video clip could be played directly upon selection of the tuple, by substituting the last attribute specification in the **WITH** statement by the following: **video_clip AS video ON b(south)**

In this case, the user would be presented with a video display element in the bottom half of panel **b**, with the standard command buttons "Stop", "Play", "Pause", "FF", "RW" found in any video player.

However, since the query intended to let the user decide when to load the video rather than load it automatically for display, we must use the button as a display element and trigger the video player only when the user clicks on the button. This feature allows fast retrieval of tuples for navigation, loading "heavy" [19] attributes only when the user wants to see them. The same technique could be used for attributes that contain images, audio, postscript documents, or any other element which takes longer to load than text.

We will now introduce a more extended example application, which we have implemented.

[19]By "heavy" we mean attribute values that are of a significantly larger size than the rest of the attributes of the tuple

Example 7 *The database contains the campus map, plus maps of each floor in the different buildings on the UTEP campus. For each building, some pictures are available, together with a list of rooms, their descriptions, and their corresponding coordinates in the floor map. There is also some additional information and/or the http address of an associated home page (if available) for some of the data items in the database. This information is in free-text format, and includes historic highlights or detailed directions to reach the buildings, and the home pages correspond to the people, organizations, offices or departments that occupy the rooms listed.*
The database schema is as follows:

CAMPUS	*(campus_area, campus_area_map)*
BUILDING	*(building_name, building_picture, campus_area, coordinates)*
FLOOR	*(building_name, floor, floor_map)*
ROOM	*(building_name, floor, room_name, description, home_page, detail_info, coordinates)*

To this schema, a user can pose queries that will let them find a particular place on campus, but also there are many (common) queries pre-defined and available.

For example, the following pre-defined query allows to view the main campus with a list of all of the buildings, as shown in Figure 9:

```
SELECT    CAMPUS.campus_area_map,
          BUILDING.building_name
FROM      CAMPUS, BUILDING
WHERE     CAMPUS.campus_area = "Main Campus"
DISPLAY   PANEL main
WITH      BUILDING.building_name AS list ON main(east),
          CAMPUS.campus_area_map AS image ON main(west)
```

Query 3. Query to display area map and list of buildings in the Main Campus.

In addition, if we want the buildings to be outlined on the map (polygons drawn over the building in green), we can use the coordinates attribute (adding it to the **SELECT** clause). It contains a series of x,y coordinates for each vertex of the polygon that outlines the building (or room, in the ROOM relation), and substitute the **DISPLAY** clause in the previous query by the next clause:

```
DISPLAY   PANEL main
WITH      building_name AS list ON main(east),
          CAMPUS.campus_area_map AS image ON main(west)
          ALL BUILDING.coordinates AS polygon ON main(west)
```

The keyword **ALL** indicates that all the attribute values of "BUILDING.coordinates" in the answer set are to be shown at the same time.

As the user navigates through the answer set, the polygon corresponding to the selected tuple (building), will be filled in red (or reverse video). The rest of the polygons (non-selected) will remain unchanged, to give a visual cue as to which tuple (building) is currently selected. If the user clicks on one of the polygons, the corresponding building will be highlighted on the list. Thus, the **ALL** keyword turns any display element into a virtual "list" in the sense the user can navigate through the answer set via the displayed elements, just as if they were displayed in a list.

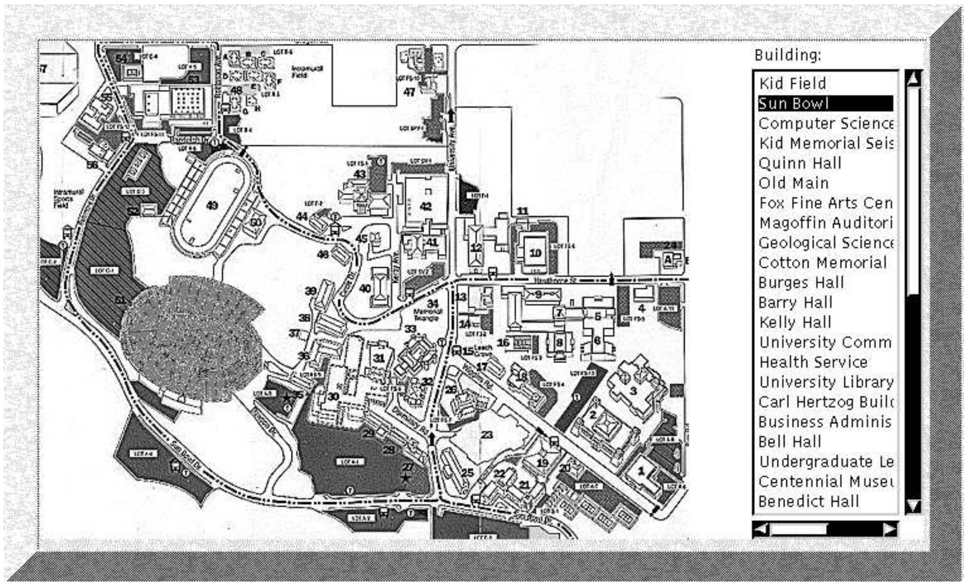

Figure 9: Display generated by Query 3

Specifying the same location for both the coordinates and the area_map indicates that the elements should be overlaid. Overlaying order follows the order in which the elements are specified in the **DISPLAY** clause.

The idea of overlaying can be extended and applied for such things as overlaying subtitles in English over a movie with audio in Spanish. This is more complex, however, as it would require matching subtitles to the corresponding frames in the movie. But it might be practical for short video clips in a language course, for example.

Now, if we want to trigger a separate query to explore a building, we change the display specification for building_name to the following:

building_name **AS** list **ON** main(east) **TRIGGER** 'building_maps(' + building_name + ')'

where the symbol '+' indicates concatenation with the actual value of the attribute building_name at run time, when the user clicks on the corresponding element on the list.

The query to start exploration of a building is stored as *building_maps*, receives as input parameter the name of a building, and is written in **SQL+D** as follows:

```
SELECT    BUILDING.building_name,
          BUILDING.building_picture,
          FLOOR.floor
FROM      BUILDING, FLOOR
WHERE     BUILDING.building_name = + input + AND
          FLOOR.building_name = BUILDING.building_name
DISPLAY   PANEL main, PANEL b ON main(south),
WITH      building_picture AS image ON main(north),
          building_name AS title ON main(north),
          ALL floor AS button on b(east) TRIGGER
            'floor_buildings(' + building_name + ',' + floor + ')'
          "Select Floor to Visit" on b(west)
```

Query 4. Query to display the building picture and buttons to go to different floors.

Here, "input" is the name of an input variable that the query receives when it is executed. In this case, the building name. It results in the screen shown in Figure 10.

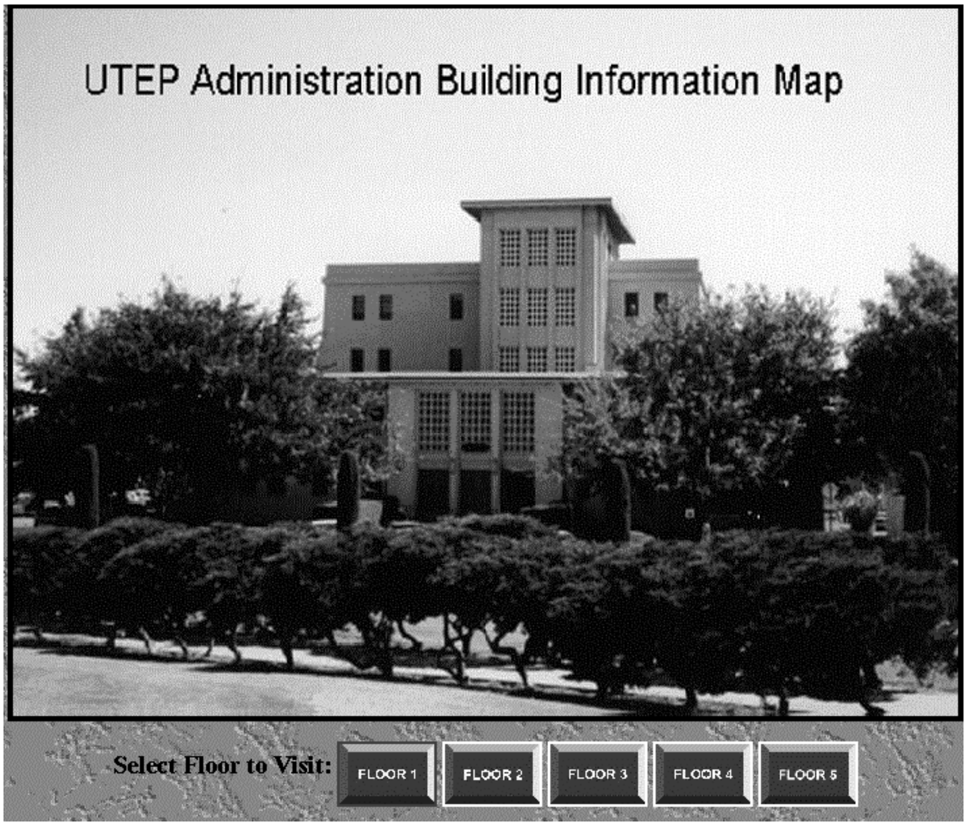

Figure 10: Display generated by Query 4

The triggered query, **floor_buildings**, looks as follows:

```
SELECT    FLOOR.floor_map, ROOM.floor,
          ROOM.room_name, ROOM.detail_info,
          ROOM.home_page, ROOM.coordinates
FROM      FLOOR, ROOM
WHERE     FLOOR.building_name = +input_1+ AND
          FLOOR.floor = +input_2+ AND
          ROOM.floor = FLOOR.floor
DISPLAY   PANEL main,
          PANEL b on main(west),
          PANEL c on main(east),
          PANEL d on b(south)
WITH      floor_map AS image ON c(north),
          ALL coordinates AS poly ON c(north),
          room_name AS list ON b(north),
          detail_info AS text on b(center),
          ALL DISTINCT floor AS button on c(south)
            TRIGGER 'floor_buildings(' + building_name + ',' + floor + ')'
          home_page AS button on b(south)
```

Query 5. Query to display the floor map and a list of rooms, plus extra information.

It receives two input parameters: the building name and the floor. This is a more elaborated query, since now the trigger makes a recursive call. In **SQL+D** , recursively calling a query is possible since now the same query is invoked, but with new parameters, and the display changes to show the changing data. The generated display is shown in Figure 11.

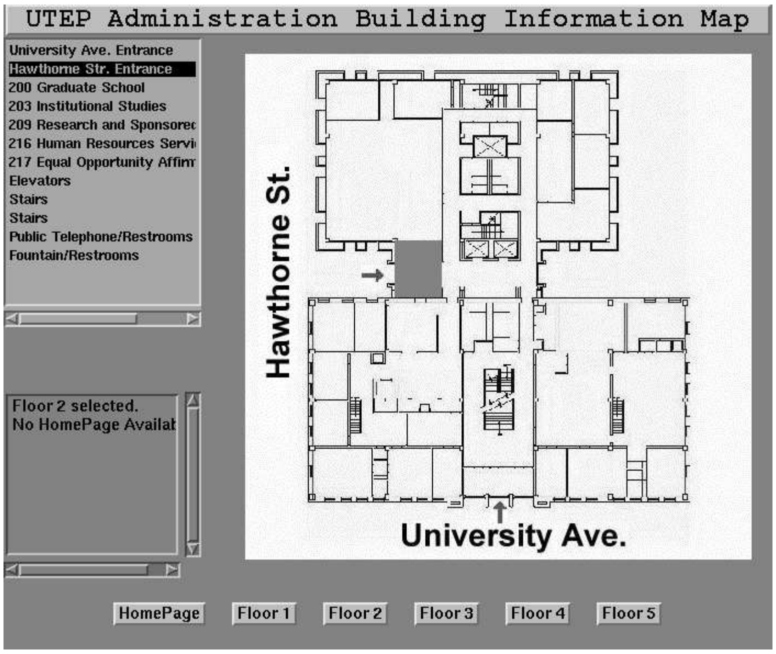

Figure 11: Display generated by Query 5

6 Related Work and Future Directions

Our work is a first step towards general and systematic design and modeling of multi-media and semi-structured databases. In the past several specific designs have been suggested [Fea95, OS95, Sri95, MMM96]. For example in [MMM96] WebSQL a query language with respect to a virtual web database is presented. We believe it is important to focus on general approaches. In [MS96] a general approach to design multi-media databases is presented, but it proposes a completely new model. Our goal has been to stay close to standard models (ORDB, OODB), particularly to relational model.

Recently, there has been a trend towards object-relational databases. Very few papers have been published on this topic though. As evident from our approach we believe it (with certain modifications) to be suitable for multi-media databases. In the text of this paper we have used features from SQL3 and the data model of Illustra. Most ORDB models focus more on complex types than on multi-media databases. For example, Stonebraker in his book [Sto96] has significantly more examples dealing with non-multi-media complex data than on multi-media data. *In this paper our main concern is multi-media data and thus we have consciously overlooked non-multi-media complex data.*

In the future we would like continue further on our proposed approach. In particular, we would like to immediately pursue, query optimization, combining relations, and making it easier for the user to ask a query. With respect to query optimization, we need to prioritize different selections. In general, less expensive selections should be done before more expensive selections. For example, equality and inequality based selections may need to be done before selections involving similarity between pictures.

In conventional database design, after ER-diagrams are converted to relations, certain relations can be combined. We need to do the same with respect to the CER diagrams. But we need to be careful about the meta-data (particularly, the subclass informations). For example, when the ER diagram (in [Ull88]) about Employees, departments, and managers is converted to relations we have the relations EMPS(ename, salary), MANAGER(ename), DEPTS(dname, d#), MANAGES(ename, dname). The last three are later merged to a single relation DEPTS(dname, d#, manager). If this was a CER diagram, we will also have $MANAGER$ is a subtable of $EMPS$, which is affected by the merging and we need to change it to $DEPTS.manager$ is a subtable of $EMPS$. This is used in Example 4.

One of our goal is to make it easier for a user to ask a query. Particularly, we would like the user to be able to ask a query even if he/she is not completely familiar with the schema. In this paper we have taken a few steps on this direction. We can further improve our approached by allowing incomplete path expressions [VdBV93, Gon94, IL94].

References

[BC95] F. Bancilhon and A. Carlson. Providing rich query functionality - SQL3 discussion paper, Dec 95. ISO/IEC JTC1/SC 21/WG3 DBL LHR, ASC X3H2-95-462.

[BCN92] C. Batini, S. Ceri, and S. Navathe. *Conceptual database design: an ER approach.* Benjamin/Cummings, 1992.

[Bee93] D. Beech. Collections of Objects in SQL3. In *VLDB 93*, pages 244–255, 1993.

[BGS97] C. Baral, G. Gonzalez, and T. Son. A multimedia extension to SQL. Technical report, Dept of Computer Sc, University of Texas at El Paso, 1997.

[Cat96] R Cattel, editor. *The Object Database Standard: ODMG - 93, release 1.2.* Morgan Kaufman, 1996.

[DD95] C.J. Date and H. Darwen. The third manifesto. *ACM SIGMOD Record*, 24(1), 1995.

[Fal96] C. Faloutsos. *Searching multi-media databases by content.* Kluwer academic publishers, 1996.

[Fea95] M. Flickner et al. Query by image and video content: The QBIC system. *IEEE Computer*, 28(9):23–32, 1995.

[Gon94] G. Gonzalez. Intuitive querying in object oriented databases, 1994. M.S Thesis, Department of Computer Science, University of Texas at El Paso.

[IL94] Y. Ioannidis and Y. Lashkari. Incomplete path expressions and their disambiguation. In *SIGMOD*, pages 138–149, 94.

[KLW93] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of ACM– to appear*, 1993.

[MMM96] A. Mendelzon, G. Mihaila, and T. Milo. Querying the world wide web. Tech Report. Univ of Toronto. 1996

[MS] F. Manola and J. Sutherland. SQL3. In ANSI X3H7 Object Model Features Matrix, http://info.gte.com/ftp/doc/activities/x3h7/by_model/SQL3.html.

[MS92] J. Melton and A. Simon. *Understanding the new SQL: a complete guide.* Morgan Kaufman, 1992.

[MS96] S. Marcus and V. S. Subrahmanian. Foundations of multimedia information systems. JACM, May 1996.

[OS95] V. Ogle and M. Stonebraker. Chabot: Retrieval from a relational database of images. *IEEE Computer*, 28(9):40–48, 1995.

[Sri95] R. Srihari. Automatic indexing and content-based retrieval of captioned images. *IEEE Computer*, 28(9):49–56, 1995.

[Sto96] M. Stonebraker. *Object Relational DBMSs.* Morgan Kaufmann, 1996.

[SYH94] P. Sistla, C. Yu, and R. Haddad. Reasoning about spatial relationship in picture retrieval systems. In *VLDB 94*, Santiago, Chile, 1994.

[Ull88] J. Ullman. *Principles of database and knowledge-base systems*, volume 1. Computer Science Press, 1988.

[VdBV93] J. Van den Bussche and G. Vossen. An extension of path expressions to simplify navigation in object-oriented queries. In *Proc. Deductive and Object Oriented Databases*, pages 267–283, 1993.

[Wad96] Drew Wade. SQL3/OQL Merger, May, 1996. http://www.jcc.com/sql_odmg_convergence.html.