Object Relational and Extended Relational Databases

Database Design for an ORDMS

The object-relational database systems are an attempt to merge the two different kind of system. It is an object database enhancement of a relational model, a hybrid in design. Perhaps, the most visible aspect that we might observe is in the addition of object database features in the SQL revision for this hybrid model. One of the pitfalls of relation model was in describing complex objects. The vibe of object-oriented mechanisms was brought into the play with the introduction of type constructors to describe row type that corresponds to the tuple constructor, array type for specifying collections, sets, list, mechanism for specifying object identity, encapsulation, inheritance, and more.

Note that the core technology used in ORDBMS is based upon the relational model. The commercial implementations simply added a layer of some of the object-oriented principle on top of the relational database management system. The simplest example is Microsoft SQL Server. Since this system is based on relational model, there is one more problem added to it: translating object-oriented concept to relational mechanism. However, this problem is extenuated by an object-oriented application that helps in the communication between the object-oriented applications with the underlying relational databases.

Understand that relational and object-oriented principle do not go well together, because they work on different principle. Therefore, it may seem that this model somehow tries to coerce them into a truce for the sake of developer's convenience. The real reason is to permit storage and retrieval of objects in an RDBMS way by providing extension to the query language to work on the object-oriented principle.

Some common implementation includes Oracle Database, PostgreSQL, and Microsoft SQL Server.

Advantages and Disadvantages of ORDBMS

ORDBMSs can provide appropriate solutions for many types of advanced database applications.

Advantages of ORDBMSs

There are following advantages of ORDBMSs:

Reuse and Sharing: The main advantages of extending the Relational data model come from reuse and sharing. Reuse comes from the ability to extend the DBMS server to perform standard functionality centrally, rather than have it coded in each application.

Increased Productivity: ORDBMS provides increased productivity both for the developer and for the, end user

Use of experience in developing RDBMS: Another obvious advantage is that .the extended relational approach preserves the significant body of knowledge and experience that has gone into developing relational applications. This is a significant advantage, as many organizations would find it prohibitively expensive to change. If the new functionality is designed appropriately, this approach should allow organizations to take advantage of the new extensions in an evolutionary way without losing the benefits of current database features and functions.

However, there are also disadvantages.

Disadvantages of ORDBMSs

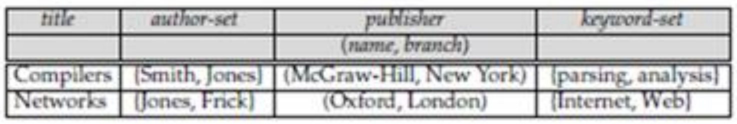
The ORDBMS approach has the obvious disadvantages of complexity and associated increased costs. Further, there are the proponents of the relational approach that believe the· essential simplicity' and purity of the .relational model are lost with these types of extension.

ORDBMS vendors are attempting to portray object models as extensions to the relational model with some additional complexities. This potentially misses the point of object orientation, highlighting the large semantic gap between these two technologies. Object applications are simply not as data-centric as relational-based ones.

Nested Relations and Collections

We defined *first normal form* (1NF), which requires that all attributes have *atomic domains*. Recall that a domain is *atomic* if elements of the domain are considered to be indivisible units.

The assumption of 1NF is a natural one in the bank examples we have considered. However, not all applications are best modeled by 1NF relations. For example, rather than view a database as a set of records, users of certain applications view it as a set of objects (or entities). These objects may require several records for their representation. We shall see that a simple, easy-to-use interface requires a one-to-one correspondence between the user's intuitive notion of an object and the database system's notion of a data item.



title	*author-set*	*publisher*	*keyword-set*
		(name, branch)	
Compilers	{Smith, Jones}	(McGraw-Hill, New York)	{parsing, analysis}
Networks	{Jones, Frick}	(Oxford, London)	{Internet, Web}

The **nested relational model** is an extension of the relational model in which do- mains may be either atomic or relation valued. Thus, the value of a tuple on an at- tribute may be a relation, and relations may be contained within relations. A complex object thus can be represented by a single tuple of a nested relation. If we view a tu- ple of a nested relation as a data item, we have a one-to-one correspondence between data items and objects in the user's view of the database.

We illustrate nested relations by an example from a library. Suppose we store for

each book the following information:

• Book title

• Set of authors

• Publisher

• Set of keywords

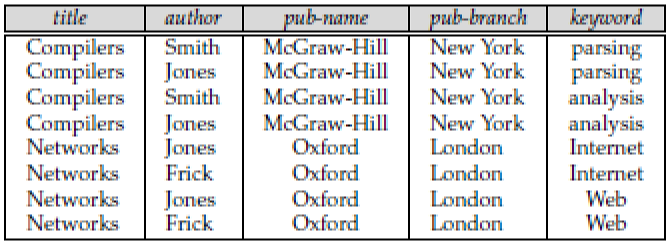
We can see that, if we define a relation for the preceding information, several domains will be nonatomic.

• **Authors**. A book may have a set of authors. Nevertheless, we may want to find all books of which Jones was one of the authors. Thus, we are interested in a subpart of the domain element "set of authors."

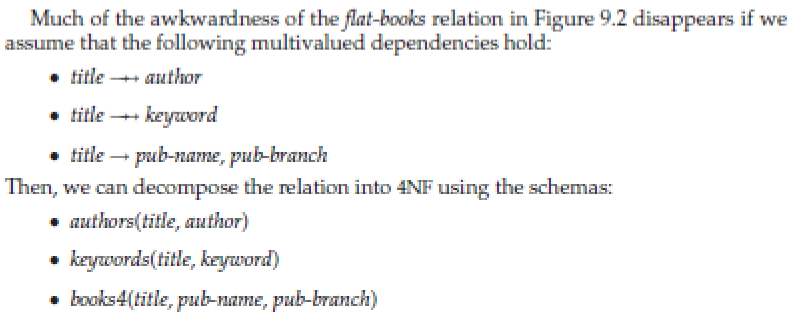
• **Keywords**. If we store a set of keywords for a book, we expect to be able to retrieve all books whose keywords include one or more keywords. Thus, we view the domain of the set of keywords as nonatomic.

• **Publisher**. Unlike *keywords* and *authors*, *publisher* does not have a set-valued domain. However, we may view *publisher* as consisting of the subfields *name* and *branch*. This view makes the domain of *publisher* nonatomic.

Figure 1 shows an example relation, *books*. The *books* relation can be represented in 1NF, as in Figure 2. Since we must have atomic domains in 1NF, yet want access to individual authors and to individual keywords, we need one tuple for each (keyword, author) pair. The *publisher* attribute is replaced in the 1NF version by two attributes: one for each subfield of *publisher*.



title	*author*	*pub-name*	*pub-branch*	*keyword*
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web



Much of the awkwardness of the *flat-books* relation in Figure 9.2 disappears if we assume that the following multivalued dependencies hold:

- *title* $\twoheadrightarrow$ *author*
- *title* $\twoheadrightarrow$ *keyword*
- *title* $\rightarrow$ *pub-name, pub-branch*

Then, we can decompose the relation into 4NF using the schemas:

- *authors(title, author)*
- *keywords(title, keyword)*
- *books4(title, pub-name, pub-branch)*

Figure shows the projection of the relation *flat-books* of Figure 2 onto the preceding decomposition.

Although our example book database can be adequately expressed without using nested relations, the use of nested relations leads to an easier-to-understand model: The typical user of an information-retrieval system thinks of the database in terms of books having sets of authors, as the non-1NF design models. The 4NF design would require users to include joins in their queries, thereby complicating interaction with the system.

We could define a non-nested relational view (whose contents are identical to *flat- books*) that eliminates the need for users to write joins in their query. In such a view, however, we lose the one-to-one correspondence between tuples and books.

Storage and Access Methods

Since object-relational databases store new types of data, ORDBMS implementors need to revisit some of the storage and indexing issues. In particular, the system must effi ciently store ADT objects and structured objects and provide effi cient indexed access to both.

Storing Large ADT and Structured Type Objects

Large ADT objects and structured objects complicate the layout of data on disk. This problem is well understood and has been solved in essentially all ORDBMSs and OODBMSs. We present some of the main issues here. User-de fined ADTs can be quite large. In particular, they can be bigger than a single disk page. Large ADTs, like BLOBs, require special storage, typically in a di fferent location on disk from the tuples that contain them. Disk-based pointers are maintained from the tuples to the objects they contain.

Structured objects can also be large, but unlike ADT objects they often vary in size during the lifetime of a database. As the years pass some of the bit actors in an old movie become famous

When an actor becomes famous , Dinky might want to advertise his or her presence in the earlier fi lms. This involves an insertion into the stars attribute of an individual tuple in fi lms. Because these bulk attributes can grow arbitrarily, flexible disk layout mechanisms are required.

An additional complication arises with array types. Traditionally, array elements are stored sequentially on disk in a row-by-row fashion; for example

However, queries may often request subarrays that are not stored contiguously on disk (e.g.,

A 1,A ,...,A).

Such requests can result in a very high I/O cost for retrieving the subarray. In order to reduce the number of I/Os required in general, arrays are often broken into contiguous chunks,

which are then stored in some order on disk. Although each chunk is some contiguous region of

the array, chunks need not be row-by-row or columnby- column.

For example, a chunk of size 4 might be A ,A ,A ,A , which is a square region if we think

of the array as being arranged row-by-row in two dimensions.

Indexing New Types

One important reason for users to place their data in a database is to allow for efficient access via

indexes. Unfortunately, the standard RDBMS index structures support only equality conditions (B trees

and hash indexes) and range conditions (B trees).

An important issue for ORDBMSs is to provide efficient indexes for ADT methods and operators on

structured objects.

Many specialized index structures have been proposed by researcher's for particular applications such as cartography, genome research, multimedia repositories, Web search and so on

An ORDBMS company cannot possibly implement every index that has been invented. Instead, the set of index structures in an ORDBMS should be user extensible. Extensibility would allow an expert in

Cartography.

For example, to not only register an ADT for points on a map (i.e., latitude/longitude

pairs), but also implement an index structure that supports natural map queries.

One way to make the set of index structures extensible is to publish an access method interface that lets users implement an index structure outside of the DBMS. The index and data can be stored in a le system, and the DBMS simply issues the open, next, and close iterator requests to the user's external index code. Such functionality makes it possible for a user to connect a DBMS to a Web search engine, for example. A main drawback of this approach is that data in an external index is not protected by the DBMS's support for concurrency and recovery. An alternative is for the ORDBMS to provide a generic `template' index structure that is sufficiently general to encompass most index structures that users might invent. Because such a structure is implemented within the DBMS, it can support high concurrency and recovery. The Generalized Search Tree (GiST) is such a structure. It is a template index structure based on B trees, which allows most of the tree index structures invented so far to be implemented with only a few lines of user-defined ADT code.

Query Processing and Optimization

ADTs and structured types call for new functionality in processing queries in ORDBMSs. They also change a number of assumptions that affect the efficiency of queries.

User defined Aggregation Function

Since users are allowed to define new methods for their ADTs, it is not unreason-able to expect them to want to de ne new aggregation functions for their ADTs as well. For example, the usual SQL aggregates|COUNT, SUM, MIN, MAX, AVG|are not particularly appropriate for the image type in the Dinky schema.

Most ORDBMSs allow users to register new aggregation functions with the system. To register an aggregation function, a user must implement three methods, which we will call initialize, iterate,and terminate.The initialize method initializes the internal state for the aggregation. The iterate method updates that state for every tuple seen, while the terminate method computes the aggregation result based on the nal state and then cleans up. As an example, consider an aggregation function to compute the second-highest value in a eld. The initialize call would allocate storage for the top two values, the iterate call would compare the current tuple's value with the top two and update the top two as necessary, and the terminate call would delete the storagefor the top two values, returning a copy of the second-highest value.

Method Security

ADTs give users the power to add code to the DBMS; this power can be abused. A buggy or malicious ADT method can bring down the database server or even corrupt the database. The DBMS must have mechanisms to prevent buggy or malicious user code from causing problems. It may make sense to override these mechanisms for efficiency in production environments with vendor supplied methods. However it is important for the mechanisms to exist, if only to support debugging ADT methods, otherwise method writers would have to write bug free code before registering their methods with the DBMS not a very forgiving programming environment.

One mechanism to prevent problems is to have the user methods be interpreted rather than compiled. The DBMS can check that the method is well behaved either by restricting the power of the interpreted language or by ensuring that each step taken by a method is safe before executing it. Typical interpreted languages for this purpose include Java and the procedural portions of SQL:1999.

An overview of SQL3

Retrieving data from a database consists of 3 main processes:

1. Formulation of an information/data request - the query,
2. Query execution by the dbms *query processor* and
3. Result presentation.

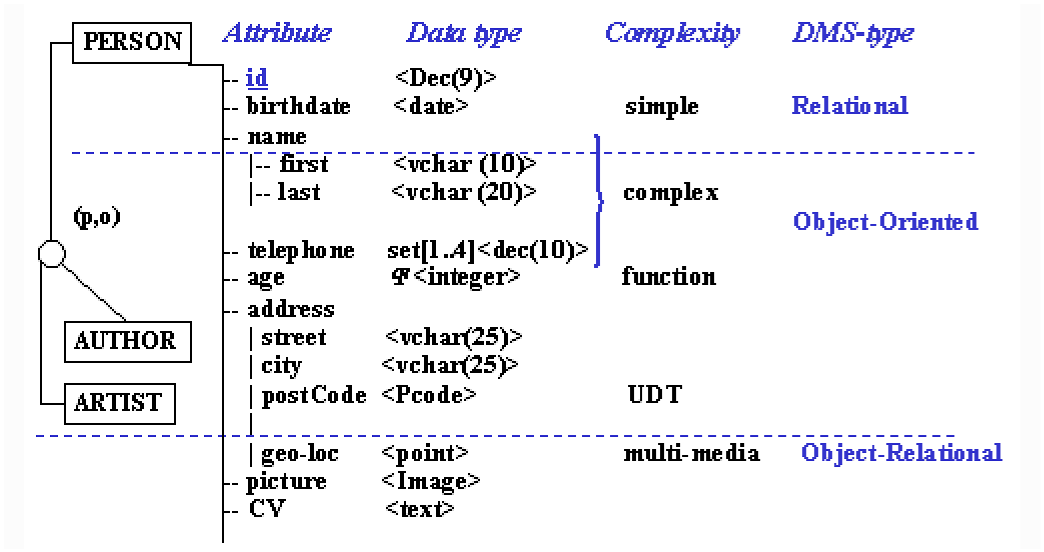
A DB query language will always provide specification of the selection criteria for the desired information and can also include information for the remaining processes.

Note here that the SQL query includes elements of the execution process (the joins) and the desired result, while the text IR query only gives the selection criteria, assuming that the system will provide the selection process as well as determine the result presentation.

For structured/relational databases, the query language utilizes the well-defined DB structure that gives names and dimensions to relations/tables and their attributes/columns, for query specification, execution and result presentation,

SQL was designed for the regularly structured data of relational databases and for *exact match*, Boolean queries, i.e. selection of data that exactly match the search values given in the query conditions. It is currently the most well-known of the structured DB query languages and has formed a framework for query languages for other (non-relational) database types such as object-oriented and spatial databases. However, SQL (and SQL2) do not support access to complex structures.

SQL3 was accepted as the new standard for SQL in 1999, after more than 7 years of debate. Basically, SQL3 includes data definition and management techniques from Object-Oriented dbms, OO-dbms, while maintaining the relational dbms platform. Based on this merger of concepts and techniques, DBMSs that support SQL3 are called *Object-Relational* or or-dbms'.



The most central data modelling notions included in SQL3 are illustrated in Image and support specification of:

- Classification hierarchies,
- Embedded structures that support composite attributes,
- Collection data-types (sets, lists/arrays, and multi-sets) that can be used for multi-valued attribute types,
- Large OBject types, LOBs, within the DB, as opposed to requiring external storage, and
- User defined data-types and functions (UDT/UDF) that can be used to define complex structures and derived attribute value calculations, among many other function extensions.

Query formulation in SQL3 remains based in the structured, relational model, though several functional additions have been made to support access to the new structures and data types. *Note: there are syntactic differences between or-dbms implementations of this new functionality.*

Implementation Issues of Extended Type

There are various implementation issues regarding the support of an extended type system with associated functions (operations). We briefly summarize them here

• The ORDBMS must dynamically link a user-defined function in its address space only when it is required. As we saw in the case of the two ORDBMSs, numerous functions are required to operate on two- or three-dimensional spatial data, images, text, and so on. With a static linking of all function libraries, the DBMS address space may increase by an order of magnitude. Dynamic linking is available in the two ORDBMSs that we studied.

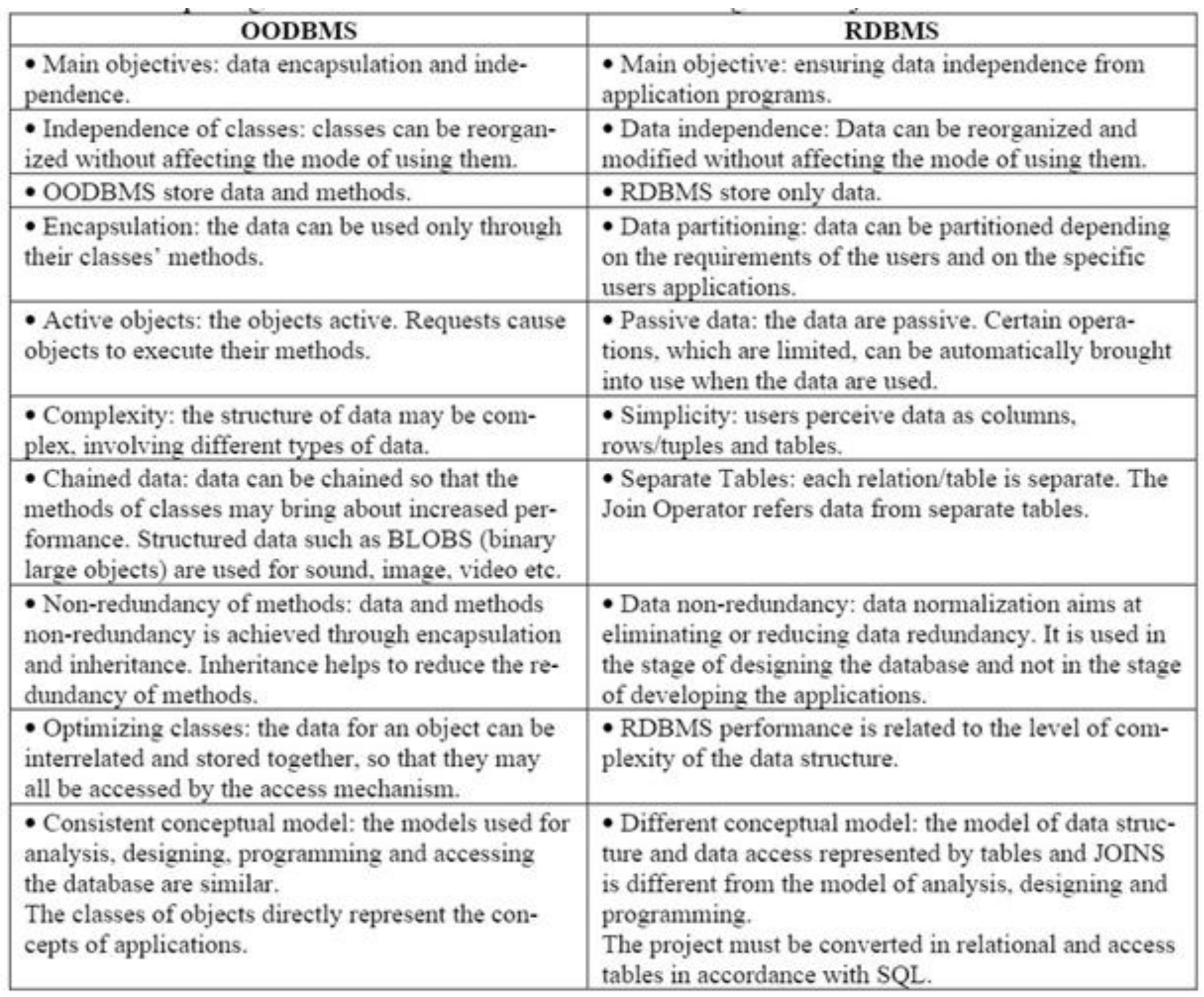
• Client-server issues deal with the placement and activation of functions. If the server needs to perform a function, it is best to do so in the DBMS address space rather than remotely, due to the large amount of overhead. If the function demands computation that is too intensive or if the server is attending to a very large number of clients, the server may ship the function to a separate client machine. For security reasons, it is better to run functions at the client using the user ID of the client. In the future functions are likely to be written in interpreted languages like JAVA.

• It should be possible to run queries inside functions. A function must operate the same way whether it is used from an application using the application program interface (API), or whether it is invoked by the DBMS as a part of executing SQL with the function embedded in an SQL statement. Systems should support a nesting of these "callbacks."

• Because of the variety in the data types in an ORDBMS and associated operators, efficient storage and access of the data is important. For spatial data or multidimensional data, new storage structures such as Rvtrees, quad trees, or Grid files may be used.

The ORDBMS must allow new types to be defined with new access structures. Dealing with large text strings or binary files also opens up a number of storage and search options. It should be possible to explore such new options by defining new data types within the ORDBMS.

System comparison of RDMS,OODBMS and ORDBMS



OODBMS	RDBMS
• Main objectives: data encapsulation and independence.	• Main objective: ensuring data independence from application programs.
• Independence of classes: classes can be reorganized without affecting the mode of using them.	• Data independence: Data can be reorganized and modified without affecting the mode of using them.
• OODBMS store data and methods.	• RDBMS store only data.
• Encapsulation: the data can be used only through their classes' methods.	• Data partitioning: data can be partitioned depending on the requirements of the users and on the specific users applications.
• Active objects: the objects active. Requests cause objects to execute their methods.	• Passive data: the data are passive. Certain operations, which are limited, can be automatically brought into use when the data are used.
• Complexity: the structure of data may be complex, involving different types of data.	• Simplicity: users perceive data as columns, rows/tuples and tables.
• Chained data: data can be chained so that the methods of classes may bring about increased performance. Structured data such as BLOBS (binary large objects) are used for sound, image, video etc.	• Separate Tables: each relation/table is separate. The Join Operator refers data from separate tables.
• Non-redundancy of methods: data and methods non-redundancy is achieved through encapsulation and inheritance. Inheritance helps to reduce the redundancy of methods.	• Data non-redundancy: data normalization aims at eliminating or reducing data redundancy. It is used in the stage of designing the database and not in the stage of developing the applications.
• Optimizing classes: the data for an object can be interrelated and stored together, so that they may all be accessed by the access mechanism.	• RDBMS performance is related to the level of complexity of the data structure.
• Consistent conceptual model: the models used for analysis, designing, programming and accessing the database are similar. The classes of objects directly represent the concepts of applications.	• Different conceptual model: the model of data structure and data access represented by tables and JOINS is different from the model of analysis, designing and programming. The project must be converted in relational and access tables in accordance with SQL.

Object oriented model	**Relational Model**	**Differences**
Object	Entity	The object specifies behavior too
Class of objects	Types of Entities	The class of objects includes the common behavior of objects in that class
Class hierarchy	The data base scheme	The class hierarchy includes inheritance, while the scheme includes external keys
Class instance	Entity, tuple or record	The instance may have a more restrictive character
Attribute	Attribute	There are no differences
Relations	Relations	There are no differences They have the meaning of descriptions but with the OODBMS the inheritance includes both the state and the behavior
Messages/Interface	There are none	
Encapsulation	There is none	
Object identifier (OID)	Primary key	In the relational model if the primary key is not identified the system generates an identifier automatically
Inheritance	There is none	