Introduction

The relational data model is the dominant paradigm in the commercial database market today, and it has been for several years. However, there have been challenges to the model over the years, and they have influenced its evolution and that of database technology. The object-oriented revolution that got started in programming languages arrived to the database area in the form of a brand new data model. The relational model managed not only to survive the newcomer but to continue becoming a dominant force, transformed into the object-relational model (also called extended relational, or universal) and relegating object-oriented databases to a niche product. Although this market has many nontechical aspects, there are certainly important technical differences among the mentioned data models. In this article I describe the basic components of the relational, object-oriented, and object-relational data models. I do not, however, discuss query language, implementation, or system issues. A basic comparison is given and then future trends are discussed.

Background

In order to facilitate the comparison among the models, I will use the same example throughout the article. The example involves a universe of people; some are professors and some are students. Each professor works at a department, and some professors chair departments. Students have professors as advisors. The exact situation is depicted as entity-relationship (ER) diagram (Chen, 1976) in Figure 1. The notation given denotes that Chairs is a one-to-one relationship, Faculty is a one-to-many relationship, and Teaches is a many-to-many relationship. Also, Phones is a multi-valued attribute and Student and Professor are subclasses of Person. The inverted triangle symbol is used to denote a class/subclass relationship; the reader is warned that different authors use different symbols for this purpose.

The relational data model (Date, 2004) is well known; one simply overviews its main concepts here as it serves as the baseline for the comparison. A domain is a nonempty set; intuitively, it provides a pool of values. Every domain is assumed to come with a name (an infinite number of names, technically). Given a schema or list of domain names \( R = \{A_1, \ldots, A_n\} \), a relation on \( R \) is a subset of the Cartesian product \( A_1 \times \ldots \times A_n \). The elements of the relation are called tuples; each tuple is made up of a list of values \( a_1, \ldots, a_n \), with \( a_i \) coming from domain \( A_i \). A key \( K \) for relation \( r \) in schema \( R \) is a subset of \( R \) (i.e., a set of attributes) such that, for any two tuples in \( r \), they are the same if they have the same value for \( K \). Intuitively, the key represents (stands for) the whole tuple, a fact that is exploited in relational database design.

Usually, the domains allowed in most implementations are data types that the computer can easily handle: different numerical types (integers, reals, etc.), characters, and strings. Most database systems also offer a “date” and a “time” domain, to facilitate expression of temporal information as well as large object types, frequently used to deal with multimedia data. However, no complex types are allowed. “Complex” here means, roughly, offering the ability to store more than one value. Because tuples will be used to represent entities, this means that attributes with multiple values (or many relationships among objects) will force an object to be represented over several tuples—perhaps even over several tables. This characteristic, called the first normal form, will become an issue later on in this article.

A relational-database schema is a set of relation schemas plus a set of integrity constraints. An integrity constraint is a condition specified over the database schema that restricts the data that can be stored in the relations. The most important constraints are the key...
constraint, which specifies that certain attribute(s) form a key in a relation, and the foreign key constraint, which specifies that certain attribute(s) form a foreign key in a relation. The foreign key attributes \( K_1 \) have all their values drawn from some primary key \( K_2 \); \( K_1 \) is said to refer to \( K_2 \). Foreign key constraints (also called referential integrity constraints) are the glue that holds the relations in a database together by making sure that values in an attribute that need to refer to a certain entity do so. It is therefore necessary to specify, when talking about a relational database, which primary keys and integrity constraints are supposed to hold. Here, as an example, is how our model would be represented in a relational database; for simplicity, I use a stylized syntax, with the relation name first and the attributes as a list in parenthesis. Each foreign key declaration follows the attribute name, and each relation is followed by a primary key declaration:

- **Person**((Ssn, Name, Age)); primary key: Ssn.  
- **Professor**((Ssn, Name, Age, Rank, Salary)); primary key: Ssn.  
- **Student**((Ssn, Name, Age, GPA)); primary key: Ssn.  
- **Department**((Name, Office)); primary key: Name.  
- **Dept-Phone**((Name, Phone)); primary key: (Name, Phone).  
- **Faculty**((Ssn1 foreign key refers to Professor, Name foreign key refers to Department)); primary key: Ssn1.  
- **Chairs**((Ssn1 foreign key refers to Professor, Name foreign key refers to Department, Date)); primary key: Ssn1.  
- **Teaches**((Ssn1 foreign key refers to Professor, Ssn2 foreign key refers to Student)); primary key: (Ssn1, Ssn2).

Note that each entity gives rise to a table, and each relationship does, too. There is another option in the translation, affecting one-to-many and one-to-one relationships. A one-to-many relationship, such as Faculty, could be represented in the table corresponding to the entity in the one side (in this case, Professor); and so could one-to-one relationships; in fact, in one-to-one relationships there is a choice of tables). Many-to-many relationships need their own separate tables. Also, the multivalued attribute “Phones” cannot be added to the Department table for the reasons already cited (first normal form).

Because having several tables repeating all department information, and each with a different phone, would create redundancy, a separate table is created, with the department name representing the whole department (because Name is the primary key of Department). This is the process of normalization, on which relational database design is based. As for the class/subclass relation, there is no direct facility to capture it in relational databases; it must be simulated by one of two methods. Both methods have one table for the superclass, and one for each of the subclasses. The first method puts, on the tables corresponding to the subclasses, the attributes proper of the subclass only; the second method combines, on the tables corresponding to the subclasses, the attributes of the subclass and the superclass. I have chosen this second option on the database. Note that the table corresponding to the superclass is still needed in the second method in case there are elements that are objects of the superclass only and not of any subclasses.

**MAIN THRUST: ADVANCED DATA MODELS**

**The Object-Oriented Data Model**

There are many variations of the object-oriented data model. In this article, we use the object data management group (ODMG) model (Catell et al., 2000) and its object database language (ODL), because they set the standard data model for object-oriented databases.

The basic building blocks of the ODMG data model are objects and literals. A literal’s value may be simple or complex. There are three types of literals: atomic, collection, and structured. Atomic literals correspond to basic data types: integers (long, short, unsigned), float (float, double), Boolean, single character, string, and enumeration types. Structured literals have a tuple structure; they include types such as Date, Time,... Also, the user can define structured literals as needed, using a “Struct” construct. Collection literals specify a collection of objects or literals. Types of collections are Set, Bag, List (homogeneous), Array, and Dictionary. Each collection has a group of built-in operators. On the other hand, objects have object identifiers (OIDs) and a value, unlike literals, which have value but no OID. Objects may have a name and can be of atomic or collection type. Atomic objects are not objects without internal structure; they correspond to atomic or structured literals. For each object, properties (i.e., attributes and relationships) and operations are specified. Values of attributes are typically literals (atomic or complex) but can be OIDs. Values of relationships are always object names or a collection applied to object names.

Object definition language (ODL) is used to implement the ODMG data model. In ODL, classes are declared by giving them an interface, using the keyword interface; an interface declares the structure of an