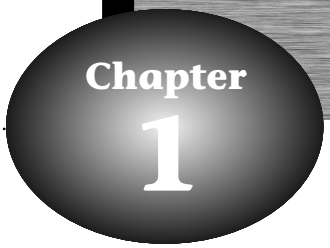


Introduction to Object-Relational Database Development

Overview

This book describes the *Object-Relational Database Management Systems (ORDBMS)* technology implemented in the *INFORMIX Dynamic Server (IDS)* product, and explains how to use it. This first chapter introduces the basic ideas behind object-relational, or extensible, DBMSs. It is intended as a road map to guide readers in their own exploration of the material.

We begin with a little historical background explaining the motivation behind an ORDBMS product such as IDS. At this early stage of the book, it is enough to say that ORDBMS technology significantly changes the way you should think about using a database. In contrast with the more byte-oriented *Relational DataBase Management System (RDBMS)* technology, an object-relational database organizes *the data and behavior of business objects* within an *abstract data model*. Object-Relational query statements deal with objects personal name, part, code, polygon and video, instead of `INTEGER`, `VARCHAR` or `DECIMAL` data values.

The chapter continues with a high-level description of the features and functionality of IDS. We introduce the *Object-Relational (OR)* data model, type and function extensibility, storage manager extensibility, and active database features. Later chapters of the book address each topic in more detail.

Moving along, we take a little time to examine how an ORDBMS is implemented internally. This digression is important because it provides a framework for understanding the best way to use the technology. An ORDBMS is a lot like an operating system. It manages resources such as memory, I/O, thread scheduling, and interprocess communications. Developers can acquire these resources from the ORDBMS for their own programs, which run under the control of the ORDBMS (instead of the operating system). Where the ORDBMS differs from a traditional operating system is in how it reorganizes its programs. Instead of running business logic, an ORDBMS executes it as part of a declarative query expression

We conclude this chapter with a brief description of the Movies-R-Us website—a sample database application—which provides many of this book's examples.

Evolution of Database Management Systems

The history of *DataBase Management Systems (DBMS)* is an evolutionary history. Each successive wave of innovation can be seen as a response either to some limiting aspect of the technology preceding it or to demands from new application domains. In economic terms, successful information technology lowers the costs incurred when implementing a solution to a particular problem. Lowering development costs increases the range of information management problems that can be dealt with in a cost-effective way, leading to new waves of systems development.

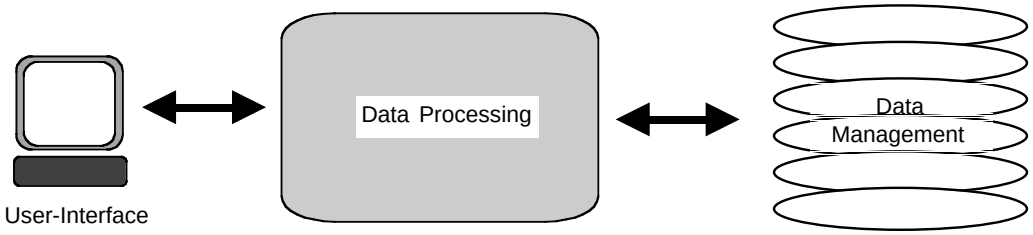


Figure 1–1. *Functional Decomposition of an Information System*

As an example of changing development costs, consider that early mainframe information systems became popular because they could make numerical calculations with unprecedented speed and accuracy (by the standards of the time). This had a major impact on how businesses managed financial assets and how engineering projects were undertaken.

It took another evolutionary step—the invention of higher level languages such as COBOL—to get to the point that the economic return from a production management system justified the development cost. It took still another innovation—the adoption of relational DBMSs—before customer management and human resource systems could be automated.

Today it is difficult to imagine how you could use COBOL and ISAM to build the number and variety of e-commerce Web sites that have sprung up over the last three years. And looking ahead, as we move into an era of ubiquitous, mobile computers and the growing importance of digital media, it is difficult to see how RDBMS technology can cope.

Early Information Systems

Pioneering information systems of the late 1960s and early 1970s were constrained by the capacity of the computers on which they ran. Hardware was relatively expensive, and by today's standards, quite limited. Consequently, considerable human effort was devoted to optimizing programming code. Diverting effort from functionality to efficiency constrained the ambition of early developers, but the diversion of effort was necessary because it was the only way to develop an information system that ran with adequate performance.

Information systems of this era were implemented as monolithic programs combining user-interface, data processing logic, and data management operations into a single executable on a single machine. Such programs intermingled low-level descriptions of data structure details—how records were laid out on disk, how records were interrelated—with user-interface management code. User-interface management code would dictate, for example, where an element from a disk record went on the screen. Maintaining and evolving these systems, even to perform tasks as simple as adding new data fields and indexes, required programmers to make changes involving most aspects of the system.

Experience has shown that development is the least expensive stage of an information system's life cycle. Programming and debugging take time and money, but by far the largest percentage of the total cost of an information system is incurred after it goes into production. As the business environment changes, there is considerable pressure to

change systems with it. Changing a pre-relational information system, either to fix bugs or to make enhancements, is extremely difficult. Yet, there is always considerable pressure to do so. In addition, hardware constantly gets faster and cheaper.

A second problem with these early systems is that as their complexity increased, it became increasingly important to detect errors in the design stage. The most pervasive design problem was redundancy, which occurs when storage structure designs record an item of information twice.

Relational DataBase Management Systems

In 1970 an IBM researcher named Ted Codd wrote a paper that described a new approach to the management of "large shared data banks." In his paper Codd identifies two objectives for managing shared data. The first of these is *data independence*, which dictates that applications using a database be maintained independent of the physical details by which the database organizes itself. Second, he describes a series of rules to ensure that the shared data is *consistent* by eliminating any redundancy in the database's design.

Codd's paper deliberately ignores any consideration of how his model might be implemented. He was attempting to define an *abstraction* of the problem of information management: a general framework for thinking about and working with information.

The Relational Model Codd described had three parts: a data model, a means of expressing operations in a high-level language, and a set of design principles that ensured the elimination of certain kinds of redundant data problems. Codd's relational model views data as being stored in *tables* containing a variable number of *rows* (or records), each of which has a fixed number of *columns*. Something like a telephone directory or a registry of births, deaths, and marriages, is a good analogy for a table. Each entry contains different information but is identical in its form to all other entries of the same kind.

The relational model also describes a number of logical operations that could be performed over the data. These operations included a means of getting a subset of rows (all names in the telephone directory with the surname "Brown"), a subset of columns (just the name and number), or data from a combination of tables (a person who is married to a person with a particular phone number).

By borrowing mathematical techniques from predicate logic, Codd was able to derive a series of design principles that could be used to guarantee that the database's structure was free of the kinds of redundancy so

problematic in other systems. Greatly expanded by later writers, these ideas formed the basis of the theory of normal forms. Properly applied, the system of normal form rules can ensure that the database's logical design is free of redundancy and, therefore, any possibility of anomalies in the data.

Relational Database Implementation

During the early 1970s, several parallel research projects set out to implement a working RDBMS. This turned out to be very hard. It wasn't until the late 1980s that RDBMS products worked acceptably in the kinds of high-end, online transactions processing applications served so well by earlier technologies.

Despite the technical shortcomings RDBMS technology exploded in popularity because even the earliest products made it cheaper, quicker, and easier to build information systems. For an increasing number of applications, economics favored spending more on hardware and less on people. RDBMS technology made it possible to develop information systems that, while desirable from a business management point of view, had been deemed too expensive.

To emphasize the difference between the relational and pre-relational approaches, a four hundred line C program can be replaced by the SQL-92 expression in Listing 1–1.

```
CREATE TABLE Employees (
    Name         VARCHAR(128),
    DOB          DATE,
    Salary       DECIMAL(10,2),
    Address             VARCHAR(128)
);
```

Listing 1–1. *Simple SQL-92 Translation of the Previous C Code*

The code in Listing 1–1 implements considerably more functionality than a C program because RDBMSs provide transactional guarantees for data changes. They automatically employ locking, logging, and backup and recovery facilities to guarantee the integrity of the data they store. Also, RDBMSs provide elaborate security features. Different tables in the same database can be made accessible only by different groups of users. All of this built-in functionality means that developers focus more of their effort on their system's functionality and less on complex technical details.

With today's RDBMSs, the simple Employees table we introduced in Listing 1–1 is more usually defined as shown in Listing 1–2.

```
CREATE TABLE Employees (
     FirstName     VARCHAR(32)   NOT NULL,
     Surname              VARCHAR(64)   NOT NULL,
     DOB           DATE          NOT NULL,
     Salary        DECIMAL(10,2) NOT NULL
                          CHECK ( Salary > 0.0 ),
     Address_1     VARCHAR(64)   NOT NULL,
     Address_2     VARCHAR(64)   NOT NULL,
     City          VARCHAR(48)   NOT NULL,
     State         CHAR(2)             NOT NULL,
     ZipCode              INTEGER             NOT NULL,
               PRIMARY KEY ( Surname, FirstName, DOB )
);
```

Listing 1–2. *Regular SQL-92 Version of the Employees Table*

Today the global market for RDBMS software, services, and applications that use relational databases exceeds $50 billion annually. SQL-92 databases remain a simple, familiar, and flexible model for managing most kinds of business data. The engineering inside modern relational database systems enables them to achieve acceptable performance levels in terms of both data throughput and query response times over very large amounts of information.

Problems with RDBMSs

Starting in the late 1980s, several deficiencies in relational DBMS products began receiving a lot of attention. The first deficiency is that the dominant relational language, SQL-92, is limiting in several important respects. For instance, SQL-92 supports a restricted set of built-in types that accommodate only numbers and strings, but many database applications began to include deal with complex objects such as geographic points, text, and digital signal data. A related problem concerns how this data is used. Conceptually simple questions involving complex data structures turn into lengthy SQL-92 queries.

The second deficiency is that the relational model suffers from certain structural shortcomings. Relational tables are flat and do not provide good support for nested structures, such as sets and arrays. Also, certain kinds of relationships, such as subtyping, between database

objects are hard to represent in the model. (Subtyping occurs when we say that one kind of thing, such as a SalesPerson, is a subtype of another kid of thing, such as an Employee.) SQL-92 supports only independent tables of rows and columns.

The third deficiency is that RDBMS technology did not take advantage of object-oriented (OO) approaches to software engineering which have gained widespread acceptance in industry. OO techniques reduce development costs and improve information system quality by adopting an object-centric view of software development. This involves integrating the data and behavior of a real-world entity into a single software module or *component*. A complex data structure or algorithmically sophisticated operation can be hidden behind a set of interfaces. This allows another programmer to make use of the complex functionality without having to understand how it is implemented.

The relational model did a pretty good job handling most information management problems. But for an emerging class of problems RDBMS technology could be improved upon.

Object-Oriented DBMS

Object-Oriented Database Management Systems (OODBMS) are an extension of OO programming language techniques into the field of persistent data management. For many applications, the performance, flexibility, and development cost of OODBMSs are significantly better than RDBMSs or ORDBMSs. The chief advantage of OODBMSs lies in the way they can achieve a tighter integration between OO languages and the DBMS. Indeed, the main standards body for OODBMSs, the Object Database Management Group (ODMG) defines an OODBMS as a system that integrates database capabilities with object-oriented programming language capabilities.

The idea behind this is that so far as an application developer is concerned, it would be useful to ignore not only questions of how an object is implemented behind its interface, but also how it is stored and retrieved. All developers have to do is implement their application using their favorite OO programming language, such as C++, Smalltalk, or Java, and the OODBMS takes care of data caching, concurrency control, and disk storage.

In addition to this seamless integration, OODBMSs possess a number of interesting and useful features derived mainly from the object model. In order to solve the finite type system problem that constrains SQL-92, most OODBMSs feature an *extensible* type system. Using this technique, an OODBMS can take the complex objects that are part of

the application and store them directly. An OODBMS can be used to invoke methods on these objects, either through a direct call to the object or through a query interface. And finally, many of the structural deficiencies in SQL-92 are overcome by the use of OO ideas such as inheritance and allowing sets and arrays.

OODBMS products saw a surge of academic and commercial interest in the early 1990s, and today the annual global market for OODBMS products runs at about $50 million per year. In many application domains, most notably computer-aided design or manufacturing (CAD/CAM), the expense of building a monolithic system to manage both database and application is balanced by the kinds of performance such systems deliver.

Problems with OODBMS

Regrettably, much of the considerable energy of the OODBMS community has been expended relearning the lessons of twenty years ago. First, OODBMS vendors have rediscovered the difficulties of tying database design too closely to application design. Maintaining and evolving an OODBMS-based information system is an arduous undertaking. Second, they relearned that declarative languages such as SQL-92 bring such tremendous productivity gains that organizations will pay for the additional computational resources they require. You can always buy hardware, but not time. Third, they re-discovered the fact that a lack of a standard data model leads to design errors and inconsistencies.

In spite of these shortcomings OODBMS technology provides effective solutions to a range of data management problems. Many ideas pioneered by OODBMSs have proven themselves to be very useful and are also found in ORDBMSs. Object-relational systems include features such as complex object extensibility, encapsulation, inheritance, and better interfaces to OO languages.

Object-Relational DBMS

ORDBMSs synthesize the features of RDBMSs with the best ideas of OODBMSs.

- Although ORDBMSs reuse the relational model as SQL interprets it, the OR data model is opened up in novel ways. New data types and functions can be implemented using general-purpose languages such as C and Java. In other words, ORDBMSs allow developers to embed new classes of data objects into the relational data model abstraction.

- ORDBMS schema have additional features not present in RDBMS schema. Several OO structural features such as inheritance and polymorphism are part of the OR data model.
- ORDBMSs adopt the RDBMS query-centric approach to data management. All data access in an ORDBMS is handled with declarative SQL statements. There is no procedural, or object-at-a-time, navigational interface. ORDBMSs persist with the idea of a data language that is fundamentally declarative, and therefore mismatched with procedural OO host languages. This significantly affects the internal design of the ORDBMS engine, and it has profound implications for the interfaces developers use to access the database.
- From a systems architecture point of view, ORDBMSs are implemented as a central server program rather than as a distributed data architectures typical in OODBMS products. However, ORDBMSs extend the functionality of DBMS products significantly, and an information system using an ORDBMS can be deployed over multiple machines.

To see what this looks such as, consider again the Employees example. Using an ORDBMS, you would represent this data as shown in Listing 1–3.

```
CREATE TABLE Employees (
        Name            PersonName      NOT NULL,
        DOB             DATE            NOT NULL,
        Salary          Currency        NOT NULL,
        Address         StreetAddress   NOT NULL
                        PRIMARY KEY ( Name, DOB )
);
```

Listing 1–3. *Object-Relational Version of Employees Table*

For readers familiar with relational databases, this `CREATE TABLE` statement should be reassuringly familiar. An object-relational table is structurally very similar to its relational counterpart and the same data integrity and physical organization rules can be enforced over it. The difference between object-relational and relational tables can be seen in the section stipulating column types. In the object-relational table, readers familiar with RDBMSs should recognize the `DATE` type, but the other column types are completely new. From an object-oriented point of view, these types correspond to class names, which are software modules that encapsulate state and (as we shall see) behavior.

As another example of the ORDBMS data model's new functionality, consider a company supplying skilled temporary workers at short notice. Such a company would need to record each employee's resumes, the geographic location where they live, and a set of Periods (fixed intervals in the time line) during which they are available, in

addition to the regular employee information. Listing 1–4 illustrates how this is done.

```
CREATE TABLE Temporary_Employees (
         Resume        DOCUMENT          NOT NULL,
         LivesAt       GEOPOINT          NOT NULL,
         Booked        SET( Period NOT NULL )
) UNDER Employees;
```

Listing 1–4. *Object-Relational Version of the Temporary_Employees Table*[1]

Readers familiar with earlier RDBMS releases of Informix Dynamic Server (IDS) will also be struck by the use of the UNDER keyword. UNDER signifies that the Temporary_Employees table inherits from the Employees table. All the columns in the Employee table are also in the Temporary_Employees table, and the rows in the Temporary_Employees table can be accessed through the Employee table.

Answering business questions about temporary employees requires that the information system be able to support concepts such as "Is Point in Circle," "Is Word in Document," and "Is some Period Available given some set of Booked Periods." In the IDS product such behaviors are added to the query language. In Listing 1–5, we present a query demonstrating OR-SQL.

"Show me the names of Temporary Employees living within 60 miles of the coordinates (-122.514, 37.221), whose resumes include references to both INFORMIX and 'database administrator,' and who are not booked for the period between today and seven days ahead."

```
SELECT Print(E.Name)
   FROM Temporary_Employees E
  WHERE Contains (GeoCircle('(-122.514, 37.221)',
                            '60 miles')),
                E.LivesAt )
    AND DocContains ( E.Resume,
                      'INFORMIX and Database Administra-
tor')
    AND NOT IsBooked ( Period(TODAY, TODAY + 7),
E.Booked );
```

Listing 1–5. *Object-Relational Query against the Employees Table*

[1] Note that this is illegal syntax. This figure is intended to illustrate a data modeling principle, rather than to demonstrate a use of OR-SQL. Refer to Chapter 2 for more details.

Again, many readers will be familiar with the general form of this SELECT statement. But this query contains no expressions that are defined in the SQL-92 language standard. In addition to accommodating new data structures, an ORDBMS can integrate logic implementing the behavior associated with the objects. Each expression, or *function name*, in this query corresponds to a behaviorial interface defined for one of the object classes mentioned in the table's creation. Developing an object-relational database means integrating whatever the application needs into the ORDBMS.

The examples in Listings 1–5 and 1–6 are both obvious extensions of SQL. But an ORDBMS is extensible in other ways too. Tables in an object-relational database can be more than data storage structures. They can be active interfaces to external data or functionality. This allows you, for example, to integrate software that interfaces with a paging system directly into the ORDBMS, and use it as shown in Listing 1–6.

"Send a page to all Temporary_Employees living within 60 miles of the coordinates (-122.514, 37.221), whose resumes includes references to both INFORMIX and 'database administrator,' and who are not booked for the period between today and seven days ahead."

```
INSERT INTO SendPage
( Pager_Number, Pass_Code, Message )
 SELECT E.Pager_Number,
        E.Pass_Code,
        Print(E.Name) ||
    ': Call 1-800-TEMPS-R-US for immediate INFORMIX DBA
job'
   FROM Temporary_Employees E
  WHERE Contains (GeoCircle('(-122.514, 37.221)',
                            '60 miles')),
         E.LivesAt )
    AND DocContains ( E.Resume,
                      'INFORMIX and Database
Administrator')
    AND NOT IsBooked ( Period(TODAY, TODAY + 7),
E.Booked );
```

Listing 1–6. *Object-Relational Query Illustrating External Systems Integration*

In a SQL-92 DBMS, SendPage could be only a table. The effect of this query would then be to insert some rows into the SendPage table. However, in an ORDBMS, SendPage might actually be an active table, which is an interface to the communications system used to send elec-

tronic messages. The effect of this query would then be to communicate with the matching temporary workers!

Object-Relational DBMS Applications

Extensible databases provide a significant boost for developers building traditional business data processing applications. By implementing a database that constitutes a better model of the application's problem domain the information system can be made to provide more flexibility and functionality at lower development cost. Business questions such as the one in Listing 1–6 might be answered in systems using an SQL-92 and C. Doing so, however, involves a more complex implementation and requires considerably more effort.

The more important effect of the technology is that it makes it possible to build information systems to address data management problems usually considered to be too difficult. In Table 1–1, we present a list of applications that early adopters of ORDBMS technology have built successfully. Other technology changes are accelerating demand for these kinds of systems.

One way to characterize applications in which an object-relational DBMS is the best platform is to focus on the kind of data involved. For thirty years, software engineers have used the term "data entry" to describe the process by which information enters the system. Human users *enter* data using a keyboard. Today many information systems employ electronics to capture information. Video cameras, environmental sensors, and specialized monitoring equipment record data in rich media systems, industrial routing applications, and medical imaging systems. Object-relational DBMS technology excels at this kind of application.

It would be a mistake to say that ORDBMSs are only good for digital content applications. As we shall see in this book OR techniques provide considerable advantages over more low-level RDBMS approaches even in traditional business data processing applications. But as other technology changes move us towards applications in which data is *recorded* rather than *entered*, ORDBMSs will become increasingly necessary.

ORDBMS Concepts and Terminology

In this section, we introduce some of the terminology used to describe extensible database technology. For reference purposes, this book also

Table 1–1. Extensible Database Problem Domains

Application Domain	*Description*
Complex data analysis	You can integrate sophisticated statistical and special purpose analytic algorithms into the ORDBMS and use them in knowledge discovery or data mining applications. For example, it is possible to answer questions such as "Which attribute of my potential customers indicates most strongly that they will spend money with me?"
Text and documents	Simple cases, such as Listing 1–5, permit you to find all documents that include some word or phrase. More complex uses would include creating a network that reflected similarity between documents.
Digital asset management	The ORDBMS can manage digital media such as video, audio, and still images. In this context, manage means more than store and retrieve. It also means "convert format," "detect scene changes in video and extract first frame from new scene," and even "What MP3 audio tracks do I have that include this sound?"
Geographic data	For traditional applications, this might involve "Show me the lat/long coordinates corresponding to this street address." This might be extended to answer requests such as "Show me all house and contents policy holders within a quarter mile of a tidal water body." For next-generation applications, with a GPS device integrated with a cellular phone, it might even be able to answer the perpetual puzzler "Car 54, where are you?"
Bio-medical	Modern medicine gathers lots of digital signal data such as CAT scans and ultrasound imagery. In the simplest case, you can use these images to filter out "all cell cultures with probable abnormality." In the more advanced uses, you can also answer questions such as "show me all the cardiograms which are 'like' this cardiogram."

includes a glossary that defines much of the language you will encounter. Subject areas corresponding to each heading in this section are covered in much more detail in later chapters. Note that this book goes beyond merely describing ORDBMS technology. It also contains chapters dealing with subjects such as OR database analysis and schema design, as well as the detailed ins and outs of writing your own user-defined extensions.

Data Model

A data model is a way of thinking about data, and the object-relational data model amounts to *objects in a relational framework*. An ORDBMS's chief task is to provide a flexible framework for organizing and manipulating software objects corresponding to real-world phenomenon.

The object-relational data model can be broken into three areas:

- Structural Features. This aspect of the data model deals with how a database's data can be structured or organized.
- Manipulation. Because a single data set often needs to support multiple user views, and because data values need to be continually updated to reflect the state of the world, the data model provides a means to manipulate data.
- Integrity and Security. A DBMS's data model allows the developers to declare rules that ensure the correctness of the data values in the database.

In the first two chapters of this book, we introduce and describe the features of an ORDBMS that developers use to build information systems. We then spend two chapters describing the second important aspect of the ORDBMS data model: user-defined type and function extensibility.

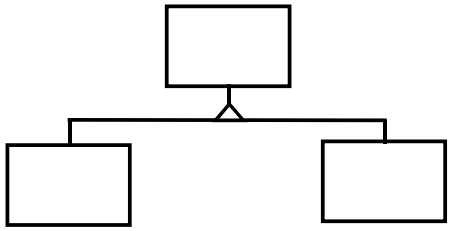
Enhanced Table Structures

An OR database consists of group a of tables made up of rows. All rows in a table are structurally identical in that they all consist of a fixed number of values of specific *data types* stored in columns that are named as part of the table's definition. The most important distinction between SQL-92 tables and object-relational database tables is the way that ORDBMS columns are not limited to a standardized set of data types. Figure 1–2 illustrates what an object-relational table looks like.

The first thing to note about this table is the way that its column headings consist of both a name and a data type. Second, note how several columns have internal structure. In a SQL-92 DBMS, such structure would be broken up into several separate columns, and oper-

People

Name::PersonName	DOB::date
(Grossmann , Marcel)	
(Millikan , Robert)	
(Mach , Ernst)	
(Ishiwara , Jun)	



Employees

Name::PersonName	DOB::date	Salary::Currency	Address::MailAddress	LivesAt::GeoPoint	Resume::Document
(Einstein , Albert)	03-14-1879	DM125,000	(12 Gehrenstrasse. .)	()	Physics, theoretical . . .
(Curie , Marie)		F125,000	(19a Rue de Seine . .)	()	Physics, experimental . . .
(Planck , Max)		DM115,000	(153 Volkenstrasse .)	()	Physics, experimental . .
(Hilbert , David)		SF210,000	(10 Geneva Avenue .)	()	Mathematics, politics. . .

Figure 1–3. *Inheritance in an Object-Relational Database*

ations over a data value such as Employee's Name would need to list every component column. Third, this table contains several instances of unconventional data types. LivesAt is a geographic point, which is a latitude/longitude pair that describes a position on the globe. Resume contains documents, which is a kind of Binary Large Object (BLOB).

Employees

Name::PersonName	DOB::date	Address::MailAddress	LivesAt::Point	Resume::Document
('Einstein','Albert')	03-14-1879	('12 Gehrenstrasse . .)	'(-127.4, 45.1)'	'Physics, theoretical
('Curie','Marie')	11-07-1867	('19a Rue de Seine . .)	'(-115.3, 27.5)'	'Physics, experimental
('Planck','Max')	04-23-1858	('153 Volkenstrasse .)	'(-121.8, 31.1)'	'Physics, experimental
('Hilbert','David')	01-23-1862	('10 Geneva Avenue .)	'(-119.2,37.81)'	'Mathematics, politics

Figure 1–2. *Structure and Data for an Object-Relational Table*

In addition to defining the structure of a table, you can include integrity constraints in its definition. Tables should all have a *key*, which is a subset of attributes whose data values can never be repeated in the table. Keys are not absolutely required as part of the table's definition, but they are a very good idea. A table can have several keys, but only one of these is granted the title of *primary key*. In our example table, the combination of the Name and DOB columns contains data values that are unique within the table. On balance, it is far more likely that an end user made a data entry mistake than two employees share names and dates of birth.

Object-Oriented Schema Features

Another difference between relational DBMSs and ORDBMSs is the way in which object-relational database schema supports features co-opted from object-oriented approaches to software engineering. We have already seen that an object-relational table can contain

Temporary_Employees

Name::PersonName	DOB::date	Booked::SET(Period NOT NULL)
('Szilard','Leo')	'2/11/1898'	{ '[6/15/1943 – 6/21/1943]','[8/21/1943 – 9/22/1943]'}
('Fermi','Enrico')	'9/29/1901'	{ '[6/10/1938 – 10/10/1939]','[6/15/1943 – 12/1/1945]', '[9/15/1951 – 9/21/1951]' }

Figure 1–4. *Non-First Normal Form (nested Relational) Table Structure*

exotic data types. In addition, object-relational tables can be organized into new kinds of relationships, and a table's columns can contain sets of data objects.

In an ORDBMS, tables can be *typed*; that is, developers can create a table with a record structure that corresponds to the definition of a data type. The type system includes a notion of *inheritance* in which data types can be organized into hierarchies. This naturally supplies a mechanism whereby tables can be arranged into hierarchies too. Figure 1–3 illustrates how the Employees table in Figure 1–2 might look as part of such a hierarchy.

In most object-oriented development environments, the concept of inheritance is limited to the structure and behavior of object classes. However, in an object-relational database, queries can address data values through the hierarchy. When you write an OR-SQL statement that addresses a table, all the records in its subtables become involved in the query too.

Another difference between ORDBMSs and traditional relational DBMSs can be seen in the Booked column of the Temporary_Employees table. The table in Figure 1–4 illustrates how this might look.

SQL-92 DBMS columns can contain at most one data value. This is called the *first normal form* constraint. In a traditional RDBMS, situations in which a group of values are combined into a single data object, are handled by creating an entirely separate table to store the group. In an ORDBMS this group of values can be carried in rows using a *COLLECTION*, which can contain multiple data values.

In Chapter 2 we explain how to create table hierarchies and tables with COLLECTION columns. In Chapter 3 we explain how they are integrated into the query language, and in Chapter 4, we explain the part they play in the extensible type system.

Table 1–2. Data Type Extensibility: Partial List of Functions for Geographic Quadrilateral Data Type

Data Type: GeoQuad

Expression:	***Explanation:***	***Example Query:***
GeoQuad(GeoPoint, GeoPoint)	Constructor function. Takes two corner points and produces a new Quadrilateral data type instance.	`INSERT INTO QuadTable VALUES ( GeoQuad ( '(0,0)', '(10,10)' ) );`

(continued)

Table 1–2. Data Type Extensibility: Partial List of Functions for Geographic Quadrilateral Data Type (continued)

Data Type: GeoQuad

Expression:	***Explanation:***	***Example Query:***
GeoQuad (double, double, double, double)	Constructor function. Takes four doubles that correspond to the X and Y values of the lower left and upper right corner.	`SELECT GeoQuad (MIN(X), MIN(Y), MAX(X), MAX(Y) ) FROM Locations;`
Contains (GeoPoint, GeoQuad)	Operator function. Returns true if the first point falls within the geographic extent of the quadrilateral.	`SELECT COUNT(*) FROM QuadTable T WHERE Contains( '(5,5)', T.Quad);`
Union (GeoQuad, GeoQuad)	Support function that computes a new quadrilateral based on the furthest corners of the two quadrilaterals inputs.	Used internally by the ORDBMS as part of R-Tree indexing.
GeoQuad (String)	Constructor function. Creates new GeoQuad using the string. A symmetric function implements the reverse.	Used to convert literal strings in queries into the internal format for the type.
GeoQuad (String)	Constructor function. Creates new GeoQuad using the string. A symmetric function implements the reverse.	Used to convert literal strings in queries into the internal format for the type.

The ORDBMS data model is considerably richer than the RDBMS data model. Unfortunately, this new richness complicates database design. There are more alternative designs that can be used to represent a particular situation, and it is not always obvious which to pick. Unthinkingly applying some of these features, such as the COLLECTION columns, creates problems. However there are data modeling problems for which a COLLECTION is an elegant solution. An important objective of this book is to provide some guidance on the subject of object-relational database design. Chapters 8 and 9 are devoted to questions of analysis and design.

Extensibility: User-Defined Types and User-Defined Functions

The concept of *extensibility* is a principal innovation of ORDBMS technology. One of the problems you encounter developing information systems using SQL-92 is that modeling complex data structures and implementing complex functions can be difficult. One way to deal with this problem is for the DBMS vendor to build more data types and functions into their products. Because the number of interesting new data types is very large, however, this is not a reasonable solution. A better approach is to build the DBMS engine so that it can accept the addition of new, application-specific functionality.

Developers can specialize or extend many of the features of an ORDBMS: the data types, OR-SQL expressions, the aggregates, and so on. In fact, it is useful to think of the core ORDBMS as being a kind of software backplane, which is a framework into which specialized software modules are embedded.

Data Types

SQL-92 specifies the syntax and behavior for about ten data types. SQL-3, the next revision of the language, standardizes perhaps a hundred more, including geographic, temporal, text types, and so on. Support for these common extensions is provided in the form of commercial DataBlade™ products. A separate tutorial in this book described the range of currently shipping DataBlade™ products.

However, relying on the DBMS vendor to supply all of the innovation does not address the fundamental problem. Reality is indifferent to programming language standards. With an object-relational DBMS developers can implement their own application specific data types and the behavior that goes with them. Table 1–2 lists some of the expressions added to the OR-SQL language to handle a geographic quadrilateral data type (an object with four corner points and parallel opposite edges).

To extend the ORDBMS with the GeoQuad type introduced in Table 1–2, a programmer would:

1. Implement each function in one of the supported procedural languages: C, Java, or SPL.
2. Compile those programs into runnable modules (shared executable libraries or Java JAR files) and place the files somewhere that the ORDBMS can read them.
3. Declare them to the database using the kind of syntax shown in Listing 1–7.

```
CREATE OPAQUE TYPE GeoQuad (
    internallength = 32
);
--
CREATE FUNCTION GeoQuad ( lvarchar )
RETURNING GeoQuad
WITH ( NOT VARIANT,
       PARALLELIZABLE )
EXTERNAL NAME
"$INFORMIXDIR/extend/2DSpat/2DSpat.bld(GeoQuadInput)"
LANGUAGE C;
--
CREATE CAST ( lvarchar AS GeoQuad WITH GeoQuad );
--
CREATE FUNCTION GeoQuad ( double precision,
      double precision,
      double precision,
      double precision )
RETURNING GeoQuad
WITH ( NOT VARIANT,
       PARALLELIZABLE )
EXTERNAL NAME
"$INFORMIXDIR/extend/2DSpat/2DSpat.bld(GeoQuadFromDoubles
)"
LANGUAGE C;
```

Listing 1–7. *Data Type Extensibility: Partial SQL Declaration of User-Defined Data Type with External Implementation in C*

In Chapters 4 and 5, we describe the extent of this functionality, and in Chapter 10, we spend considerable time on the question of how to use C to implement the fastest possible data type extensions. A special Java tutorial explores how to achieve the same thing using the Java™ language, and the SQL-J standard.

SQL-92 includes some general-purpose analytic features through its aggregates (`MIN`, `MAX`, `COUNT`, `AVG`, and so on). Aggregate functions can be used in conjunction with other aspects of SQL-92 to answer questions about groups of data. For example, the query in Listing 1–8 uses the SQL-92 table to find out what is the mean salary of each group of Employees in different cities.

"What is the average salary of Employees for each of the cities where they live?"

```
SELECT E.City,
       AVG(E.Salary)
  FROM Employees E
 GROUP BY E.City;
```

Listing 1–8. *Simple Analysis Query Using a SQL-92 Aggregate Function*

The range of aggregates in SQL-92 is limited. In data mining, and in the larger field of statistical analysis, researchers have described a tremendous number of useful analytic algorithms. For example, it might make more sense to calculate the median salary, rather than the mean salary, in the query above. Or it might be more useful to create a histogram showing how many employees fall into certain ranges of salaries. SQL-92 does not provide the means to answer such analytic questions directly. With the ORDBMS, however, these algorithms can be integrated into the server and used in the same way as built-in aggregates.

The OR-SQL query in Listing 1–9 uses the Employees table defined in Listing 1– 4. It calculates the correlation between the number of words in an Employees resume and his or her salary.

"What is the correlation coefficient between Employee salaries and the number of words in their resumes?"

```
SELECT Correlation ( E.Salary,  WordCount( E.Resume
) )
  FROM Employees E;
```

Listing 1–9. *Analysis Query with a User-Defined Aggregate (Correlation)*

In this book we take the view that an aggregate is a special kind of function, so we introduce the idea in Chapter 6. However, much of the detailed material external C description in Chapter 10 is also relevant to user-defined aggregates.

Database Stored Procedures

Almost all RDBMSs allow you to create database procedures that implement business processes. This allows developers to move considerable portions of an information system's total functionality into the DBMS. Although centralizing CPU and memory requirements on a single machine can limit scalability, in many situations it can improve the system's overall throughput and simplify its management.

By implementing application objects within the server, using Java, for examples, it becomes possible, though not always desirable, to push code implementing one of an application-level object's behaviors into the ORDBMS. The interface in the external program simply passes the work back into the IDS engine. Figure 1–5 represents the contrasting approaches. An important point to remember is that with Java, the same logic can be deployed either within the ORDBMS or within

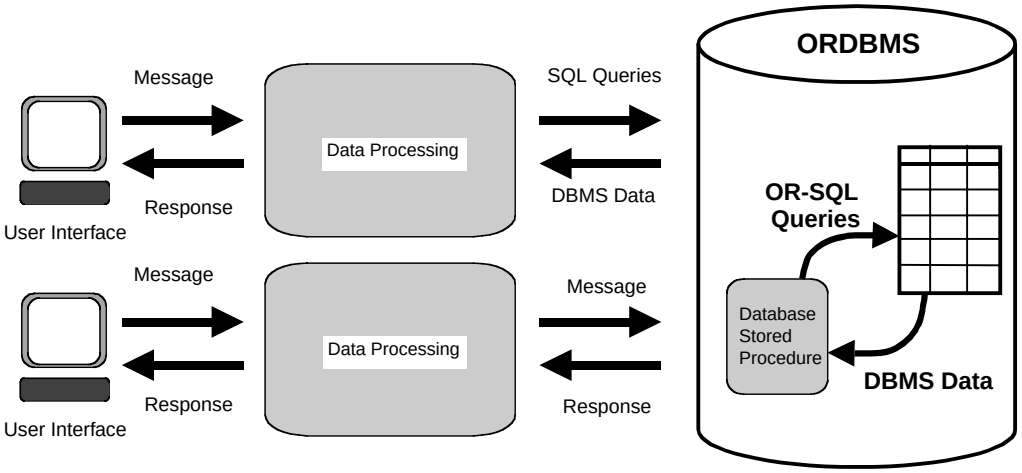


Figure 1–5. *Routine Extensibility and the ORDBMS as the Object-Server Architecture*

an external program without changing the code in any way, or even recompiling it.

In Chapter 6, we introduce the novel idea that the ORDBMS can be used to implement many of the operational features of certain kinds of middleware. Routine extensibility, and particularly the way it can provide the kind of functionality illustrated in Figure 1–5, is a practical application of these ideas. But making such system scalable requires using other features of the ORDBMS: the distributed database functionality, commercially available gateways, and the open storage manager (introduced below). Combining these facilities provides the kind of *location transparency* necessary for the development of distributed information systems.

Storage Management

Traditionally the main purpose of a DBMS was to centralize and organize data storage. A DBMS program ran on a single, large machine. It would take blocks of that machine's disk space under its control and store data in them. Over time, RDBMS products came to include ever more sophisticated data structures and ever more efficient techniques for memory caching, data scanning, and storage of large data objects.

In spite of these improvements, only a fraction of an organization's data can ever be brought together into one physical location. Data is often distributed among many systems, which is the consequence of

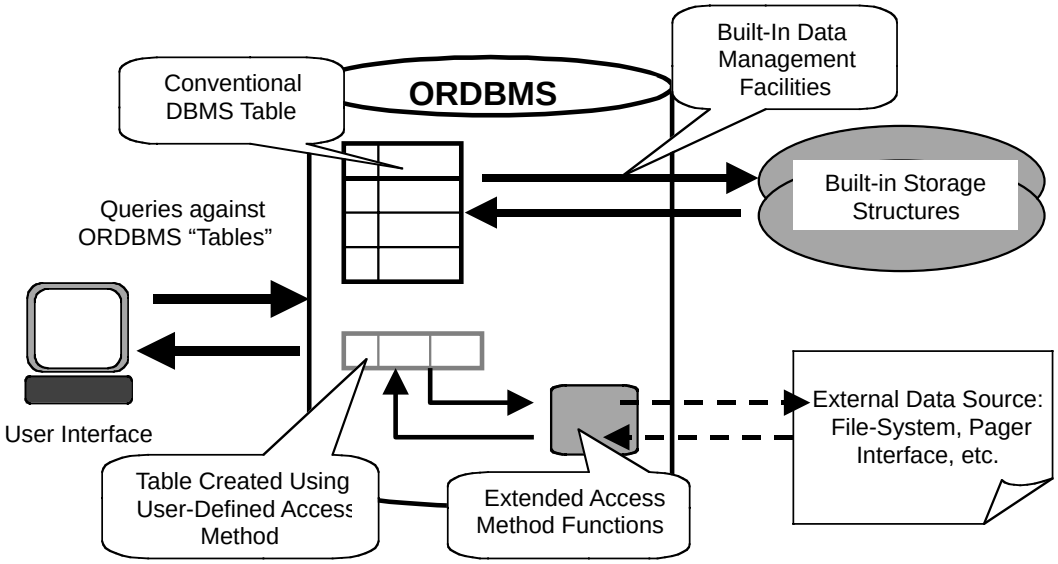


Figure 1–6. *Extensible Storage Manager*

autonomous information systems development using a variety of technologies, or through organizational mergers, or because the data is simply not suitable for storage in any DBMS. To address this, the IDS product adds a new extensible storage management facility. In Figure 1–6, we illustrate this *Virtual Table Interface* concept.

ORDBMSs possess storage manager facilities similar to RDBMSs. Disk space is taken under the control of the ORDBMS, and data is written into it according to whatever administrative rules are specified. All the indexing, query processing, and cache management techniques that are part of an RDBMS are also used in an ORDBMS. Further, distributed database techniques can be adapted to incorporate user-defined types and functions. However, all of these mechanisms must be re-implemented to generalize them so that they can work for user-defined types. For example, page management is generalized to cope with variable length OPAQUE type objects.

You can also integrate code into the engine to implement an entirely new storage manager. Developers still use OR-SQL as the primary interface to this data, but instead of relying on internal storage, the ORDBMS can use the external file system to store data. Any data set that can be represented as a table can be accessed using this technique.

Developers can also use this technique to get a snapshot of the current state of live information. It is possible to represent the current state of the operating system's processes or the current state of the file system as a table. You can imagine, for example, an application intended to help manage a fleet of trucks and employees servicing air conditioners or elevators. Each truck has a device combining a Global Positioning System (GPS) with a cellular phone that allows a central service to poll all trucks and to have them "phone in" their current location. With the ORDBMS, you can embed Java code that activates the paging and response service to implement a virtual table, and then write queries over this new table, as shown in Listing 1–10.

"Find repair trucks and drivers within '50 miles' of 'Alan Turing' who are qualified to repair the 'T-20' air conditioning unit."

```
SELECT T.Location, T.DriversName
  FROM Trucks T, Customers C
 WHERE Distance (T.Location, C.Location) < '50 miles'
   AND C.Name = 'Alan Turing'
   AND DocContains ( T.DriverQualifications,
                     'Repair for T-20');
```

Listing 1–10. *Query Reaching to Data Outside the ORDBMS*

This functionality should fundamentally change the way you think about a DBMS. In an object-relational DBMS, SQL becomes the medium used to combine components to perform operations on data. Most data will be stored on a local disk under the control of the DBMS, but that is not necessarily the case.

The Virtual Table Interface tutorial describes these features in more detail.

Distributed Deployment

Often the volume of data in a single information system, or the workload imposed by its users, is too much for any one computer. Storing shared data, and providing efficient access to it, requires that the system be *partitioned* or *distributed* across several machines. Combining

[2] It is worth noting that whether this turns them into object-relational systems is still an open question. The internal design of an ORDBMS is greatly influenced by the need to support declarative operations in a transactional system. Retrofitting either of these functions to a system that was not designed primarily to support them has been shown to be extremely difficult.

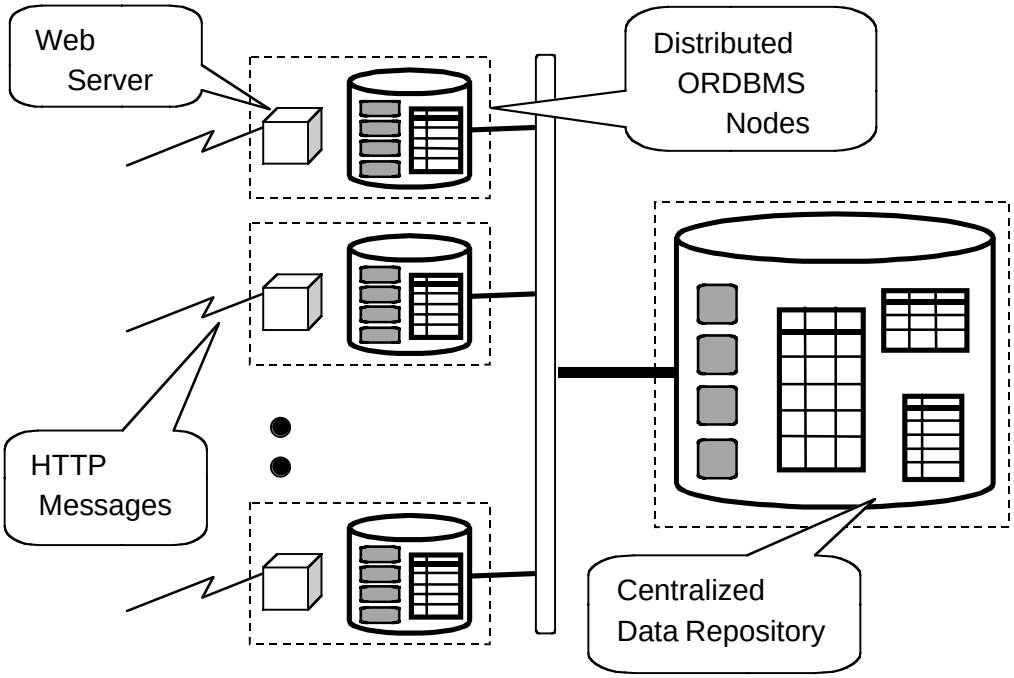


Figure 1–7. *Distributed Information System Deployment*

extensibility with distributed database features makes new system architectures possible. This concept is illustrated in Figure 1–7.

This figure illustrates a Web site configuration. A large central machine contains canonical copies of all data. Surrounding it is a cloud of other, smaller installations. These might cache replicated (read only) copies of a sub-set of the total data. All this data and all the modules of logic implementing the various extended types and functions can be queried from everywhere. The effect of this kind of architecture is to distribute the load across several machines, without compromising from data integrity.

In Chapter 6, we discuss the distributed query and replication features that make this kind of deployment possible.

Ad-Hoc Query Language

You manipulate data stored in an ORDBMS using an extended version of the SQL declarative data language that we call OR-SQL. OR-SQL is a reasonably complete implementation of a *declarative relational language*. We say it is *declarative* because OR-SQL expres-

sions describe *what* it is that is wanted, rather than a procedural, step-by-step algorithm defining *how* the task is to be accomplished. OR-SQL expressions are strings that the ORDBMS can parse, optimize, and execute at runtime.

We say OR-SQL is *relational* because it expresses operations over relational schema objects such as tables and views. OR-SQL statements are *value-oriented*. Queries make no reference to physical data management details, such as pointers, or to memory management. In fact, the same OR-SQL statement can be processed internally by the ORDBMS in many different ways. The actual operational schedule depends on factors such as the physical configuration of the database, the presence of indices on certain columns, the number of rows the ORDBMS needs to manage at various stages of the query processing, and so on.

Using declarative, value-oriented interfaces to database data distinguishes ORDBMSs from most OODBMSs. OODBMSs traditionally rejected query-centric interfaces as being too cumbersome. Recently, the OODBMS community has moved to embrace the Object Query Language (OQL) as a standard declarative interface to their systems.[2] But OR-SQL is such likely to be the dominant language because SQL is "inter-galactic database speak."

The OR-SQL implemented in IDS is a very different language from what is described in the SQL-92 language standard. The SQL-92 standard provides a definition for an entire language: a grammar, keywords, a set of data types, and a list of language expressions that can be used with these types. The OR-SQL philosophy is somewhat different. Although the same four basic data operations present in SQL-92—`INSERT`, `DELETE`, `UPDATE` and `SELECT`—form the basis of OR-SQL, the rest of the language is left "as an exercise for the implementor."

Listings 1–5 and 1–10 show examples of ORDBMS `SELECT` queries. In Listing 1–11, we introduce several queries that manipulate data.

"Delete the record associated with Employee Marie Curie."

```
DELETE FROM Employees
 WHERE Name = PersonName('Curie','Marie')
   AND DOB  = '11-07-1866';
```

"Update Albert Einstein's Employee record with a new resume taken from the supplied file."

```
UPDATE Employees
   SET Resume = GetDocument('C:\temp\Albert.doc',
                            'client')
 WHERE Name = PersonName('Einstein','Albert')
   AND DOB  = '03-14-1879';
```

"Insert an Employee record for 'Ernst Mach'."

```
INSERT INTO Employees
( Name, DOB, Salary, Address, Resume, LivesAt, Booked )
VALUES
( PersonName('Mach','Ernst'),
  '03-18-1838',
  Currency(120000.00,'DM'),
  Address('123 Gehrenstrasse',
          'Berlin','GERMANY', 8485),
  GetDocument('C:\temp\Albert.doc','client'),
  '(11.54, 47.6722 )'::GeoPoint,
  SET()
 );
```

Listing 1–11. *SQL Write Queries*

Chapter 2 and Chapter 3 describe OR-SQL in detail. Numerous examples throughout this book illustrate different kinds of query expressions. Readers wanting more information on how to write OR-SQL are advised to consult the many books on SQL-92. Obviously, these books overlook OR-SQL specific issues, but the way they generally talk about using SQL remains valid.

Application Programming Interfaces

There is always more to an information system than a database. Other, external programs are responsible for communication and for managing user interfaces. An information system will use a variety of languages to implement the non-DBMS parts of the final system. Consequently, there are several mechanisms for handling *applications programming interfaces (API)*.

OR-SQL queries can be embedded into external programs and then passed into the ORDBMS at runtime. In return, the ORDBMS sends result data to the external program. Another, very powerful way to use SQL is to write logic that actually creates SQL queries based on end-user input. Tools such as Microsoft Access, although they do not take advantage of many of the innovative features described in this book, epitomize this approach.

The most traditional API approach involves embedding OR-SQL statements directly into C programs and using a pre-parser to turn embedded OR-SQL into a procedural program. More recent approaches involve functional interfaces such as ODBC and the JDBC standard for Java programs. For Web applications, you can use mark-up tags that convey to the Web server that a OR-SQL query needs to be executed and its results formatted according to a set of mark-up rules.

We describe three of these interfaces, ESQL/C, JDBC and the Web Blade, in detail in Chapter 7. Another interface—the Server API, or SAPI—is used by C logic executing within the ORDBMS. We do not cover the SAPI OR-SQL facilities in this book.

ORDBMS Advantages

Why is all of this useful? What advantages does this give you? ORDBMS technology improves upon what came before it in three ways.

The first improvement is that can enhance a system's overall *performance*. The IDS product can achieve higher levels of data throughput or better response times than is possible using RDBMS technology and external logic because ORDBMS extensibility makes it possible to move logic to where the data resides, instead of always moving data to where the logic is running. This effect is particularly pronounced for data-intensive applications such as decision support systems and in situations in which the objects in your application are large, such as digital signal or time series data. But in general, any application can benefit.

The second improvement relates to the way that integrating functional aspects of the information system within the framework provided by the ORDBMS improves the *flexibility* of the overall system. Multiple

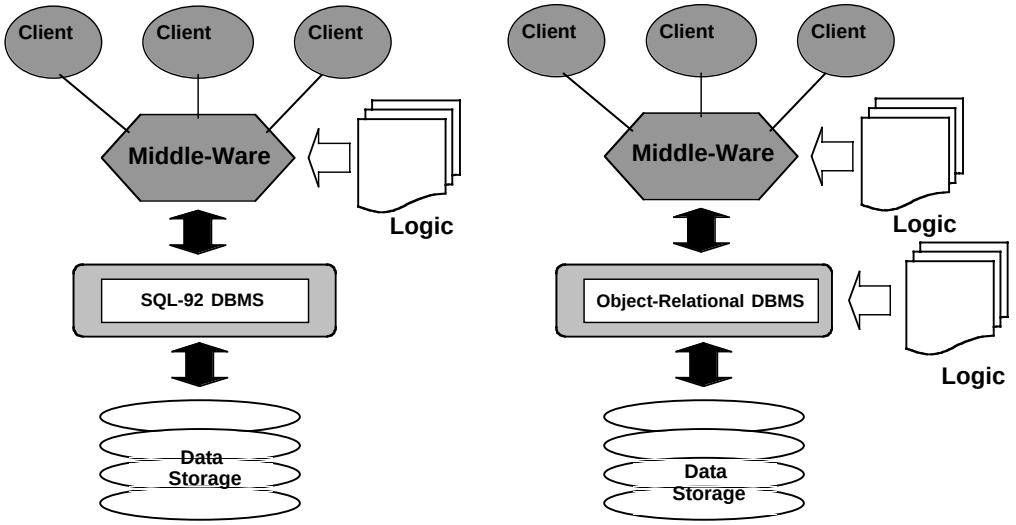


Figure 1–8. *Alternative Extensible Architectures*

Table 1–3. Latency Timings for Low-Level Data Movement Operations

Operation	*Typical Mechanism Involved*	*~Latency (secs)*
Random disk read or write	fopen(), read(), write(), seek(), and so on	0.01
Local area network (LAN)	Sockets	0.000 1
Interprocess communication	pipes and shared memory	0.000 001
Block memory copy	Memcpy	0.000 000 01

unrelated object definitions can be combined within a single ORDBMS database. At runtime, they can be mingled within a query expression created to answer some high-level question. Such flexibility is very important because it reduces the costs associated with information system development and ongoing maintenance.

The third benefit of an ORDBMS relates to the way information systems are built and managed. An ORDBMS's system catalogs become a metadata repository that records information about the modules of programming logic integrated into the ORDBMS. Over time, as new functionality is added to the application and as the programming staff changes, the system's catalogs can be used to determine the extent of the current system's functionality and how it all fits together.

The fourth benefit is that the IDS product's features makes it possible to incorporate into the database data sets that are stored outside it. This allows you to build federated databases. From within single servers, you can access data distributed over multiple places.

In this section of the chapter, we explain each advantage in detail and in turn.

Performance

It is important to understand that in a well-implemented extensible DBMS, the code you embed within it does not run on a client machine or in a middle-ware layer wrapped around the database core. Rather, the code runs inside the ORDBMS engine, as close as possible to where the data is stored. Figure 1–8 illustrates the difference.

[3] Register caching on modern CPUs is adding another level of performance difference. In general it can be 10 to 100 times as expensive to read data from RAM than from a register.

The diagram on the left illustrates how procedural logic and relational databases are deployed today. Consider what happens when you need to apply some kind of complex logic to distinguish interesting rows in a table from uninteresting ones. Achieving this with SQL-92 requires that each row retrieved from the DBMS be filtered using logic in the middle-ware. On the right, the OR approach is presented. Here, the filtering logic is embedded within the ORDBMS and invoked as part of query processing.

Moving logic into the ORDBMS can have significant performance benefits. To explain why, we first need to introduce some of the basic performance properties of computer systems. Table 1–3 lists the approximate latencies of several common, low-level operations. Each row of this table corresponds to a data movement operation, and each latency time reflects the average amount of time taken to perform the operation on 1 K of data. You can think of these as per-operation taxes paid to the operating environment and computer hardware.

Some of these numbers are fairly obvious and are included as references. Their relative sizes are due to the fundamental physics involved. The reason that it takes such an enormous amount of time (relatively speaking) to move data from magnetic disk into physical memory is because this is the only stage in the data management food chain that involves mechanical movement. At the other extreme, mempy() compiles down to a handful of low-level instructions that are executed very quickly.[3]

Between these extremes, moving data across a local-area network requires that you involve your computer's operating system, which in turn needs to interact with the networking hardware, and then the remote computer's operating system, often many times. Similarly, the time taken even to move data between two processes running on the same computer is significantly longer than the memcpy(). This is due to the operating system's mediation of the exchange. Such a transfer involves many more instructions and necessitates several process context switches.

The implication of this is obvious: the less you move data, the less overhead you incur, and the better your overall system performs. Of course, some data movement is unavoidable. Data must travel from its disk storage to where the user's eyeballs can see it. A good design principle to adopt, however, is to position the functional modules of your system in such a way that you minimize the volume of data being moved about by the information system's workload.

By providing the ability to host more procedural logic adjacent to the data, the ORDBMS opens up new possibilities, especially in the newer kinds of applications.

Working an Example

A major financial customer saw the performance possibilities of ORDBMSs early. Their problem was in calculating the value of certain complex financial instruments. Slightly simplified, the data they were dealing with was stored as shown in Listing 1–12.

```
CREATE TABLE Instruments (
           ID           INTEGER             NOT NULL
PRIMARY KEY,
           Portfolio    INTEGER             NOT NULL,
           Issuer       VARCHAR(32)  NOT NULL,
           Issuer_Class CHAR(4)             NOT NULL,
           Principal    MONEY        NOT NULL,
           Issue_Date   DATE         NOT NULL,
           Rate         FLOAT        NOT NULL
);
```

Listing 1–12. *Table for Financial Instrument Data*

The primary purpose of the application was to how much a particular portfolio (a set of instruments) was worth. Calculating this value required figuring out the value of each instrument in the portfolio on some date, and then adding up all the values. To solve this problem, they wrote the program in Listing 1–13.

```
dec_t *
CalculatePortfolioValue (
EXEC SQL BEGIN DECLARE SECTION;
PARAMATER integer Portfolio_Argument;

EXEC SQL END DECLARE SECTION;
{
    EXEC SQL BEGIN DECLARE SECTION;
           dec_t  * decTotal;
    EXEC SQL BEGIN DECLARE SECTION;
    DecTotal = malloc( sizeof(dec_t) );

EXEC SQL DECLARE Curs1 CURSOR FOR
    SELECT Issuer_Class,
           Principal,
           Issue_Date,
           Rate
```

[4] For the record, these were linear algebra operations: find the dot product of a matrix of correlation coefficients and a vector of values supplied with the instrument.

```
        FROM Instruments
```

```
     WHERE Portfolio = :Portfolio_Argument;

    EXEC SQL OPEN UpdCurs1 Curs;
    while(1)
    {
           EXEC SQL FETCH Upd_Curs INTO
           :chIssuer,:decPrincipal, :nIssueDate, :fRate;
           if ((ret = exp_chk("fetch", 1)) == 100)
                  break;
           decTotal = decadd ( decTotal,
                       CalcValue (chIssuer, decPrincipal,
               nIssueDate, fRate ));
}
EXEC SQL Close Curs1;
return decTotal;
}
```

Listing 1–13. *External Code for Portfolio Calculation*

Figuring out how much something is worth is never simple. The most valuable part of this program was the CalcValue() function, which we do not show here. CalcValue() was quite complex (and secret) because the firm had figured out how to incorporates the risk of the issuer defaulting into its calculations of an instrument's value. Government instruments, having no risk of default, generate predictable income. On the other hand, debt issued by riskier organizations tends to have higher nominal returns, but sometimes the lender may go broke. Intuitively, you consider money deposited in a bank account quite differently from money lent to an unreliable relative.

Internally, CalcValue() used a number of rather sophisticated mathematical procedures. These made it extremely difficult to implement the calculation using the stored procedure language.[4]

The financial firm's idea was to take that CalcValue() function and to integrate it into IDS. This reduced the whole code fragment in Listing 1–13 to what is shown in Listing 1–14.

"What is the total value of Instruments in portfolio X?"

```
    SELECT SUM ( CalcValue (Issuer_Class,
                                    Principal,
Issue_Date,
                                    Rate
                                ))
     FROM Instruments
     WHERE Portfolio =:Portfolio_Argument;
```

Listing 1–14. *Object-Relational SQL for Portfolio Calculation*

The performance improvement was spectacular. Calculating a portfolio's value was reduced from about two hours to about ten minutes. The difference was entirely due to the reduction in data movement, and also to the way that the ORDBMS could take advantage of parallelism while processing this query. Instead of moving all the matching rows from the Instruments table out of the DBMS, the calculation was instead performed "in place."

Flexibility and Functionality

Explaining flexibility is more difficult because the advantages of flexibility are harder to quantify. But it is probably more valuable than performance over the long run. To understand why, let's continue with the financial company example.

As it turned out, the more profound result of the integration effort undertaken by our financial firm was that the CalcValue() operation was liberated from its procedural setting. Before, developers had to write and compile (and debug) a procedural C program every time they wanted to use CalcValue() to answer a business question. With the ORDBMS, they found that they could simply write a new OR-SQL query instead. For example, the query in Listing 1–15 is a join involving a table that, while in the database, was beyond the scope of the original (portfolio) problem.

"What is the SUM() of the values of instruments in our portfolio grouped by region and sector?"

```
SELECT IS.Country,
         IS.Region,
         SUM( CalcValue ( I.Issuer_Class,
                          I.Principal,
                          I.Issue_Date,
                          I.Rate
            ))
  FROM Instruments I, Issuers IS
  WHERE I.Portfolio = :Portfolio_Argument
    AND I.Issuer    =  IS.Name
  GROUP BY IS.Country, IS.Region;
```

Listing 1–15. *Object-Relational SQL for Portfolio Calculation*

With the addition of a small Web front end, end users could use the database to find out which was the most valuable instrument in their portfolio or which issuer's instruments performed most poorly, and so on. It was known that the system had the data necessary to answer all these questions before. The problem was that the cost of answering them using C and SQL-92 was simply too high.

Maintainability

After some investigation, the developers discovered that, over time, there had been several versions of the CalcValue() function. Also, once the CalcValue() algorithm was explained to end users, they had suggestions that might improve it. With the previous system, such speculative changes were extremely difficult to implement. They required a recode-recompile-relink-redistribute cycle. But with the ORDBMS, the alternative algorithms could be integrated with ease. In fact, none of the components of the overall information system had to be brought down. The developers simply wrote the alternative function in C, compiled it into a shared library, and dynamically linked it into the engine with another name.

What all of this demonstrates is that the ORDBMS permitted the financial services company to react to changing business requirements far more rapidly than it could before. By embedding the most valuable modules of logic into a declarative language, they reduced the amount of code they had to write and the amount of time required to implement new system functionality. None of this was interesting when viewed through the narrow perspective of system performance. But is made a tremendous difference to the business's operational efficiency.

Systems Architecture Possibilities

Having implemented several such functions, the next problem confronting the development team was that although their system worked fine, it would be more useful to include data that was stored in another database. In fact, because their operations were international, they needed to transfer data across significant distances and convert data along the way. Bulk copy and load was considered unfeasible, because the remote data was volatile, and although intersite queries were rare, accurate answers were critical.

None of the other groups wanted to migrate off their efficient and stable production systems. The basic problem was that the data lived "out there" in flat files, in another information system, or in another DBMS. Nonetheless, the local business users still wanted to access it.

As an experiment, the developers decided to use the external data access features of the ORDBMS to turn it into a federated database system. Achieving this required a mix of technologies. First, simply getting to the data was a problem. In one case, the data lived in an RDBMS. Fortunately the ORDBMS product possessed a *gateway* to the other RDBMS. However, the more difficult problem was currency conversion, which was achieved by including routines to perform the translation in the ORDBMS.

Another data set was stored in a text file and manipulated by a set of Perl programs. Incorporating this into the database was harder. You might move the flat-file implementation entirely into the ORDBMS, but that would require a rewrite of the Perl scripts. In the end, the developers wrote a very simple interface that created a new storage manager to understand the external file's format and to make its data available through OR-SQL within the ORDBMS. Once this development was complete, the final system looked something like Figure 1–5.

ORDBMS Engineering

Another way to answer the question "What is an ORDBMS?" is to describe how an OR engine is constructed. An ORDBMS is an example of what is known as a *component-centric software system*. It works by allowing developers to combine many different extensions within the framework it provides. In this section, we explain how an ORDBMS works by describing how it processes a query. Along the way, we show you how integrated extensions are managed.

Processing an Object-Relational Query

When it receives an OR-SQL query, the ORDBMS breaks it up into a series of smaller, simpler operations. Each simple operation corresponds to an algorithm that manipulates a set of row data in some way. An important part of what the ORDBMS does is optimizing these operations. Optimization involves ordering the sequence to minimize the total amount of computer resources needed to resolve the query. Consider the query in Listing 1–16.

"Show me the names of Employees born on January 1st, 1967, and print the list in name order."

```
SELECT E.Name
   FROM Employees E
```

```
WHERE E.DOB    = '01-01-1967'
ORDER BY E.Name;
```

Listing 1–16. *SQL Example*

A ORDBMS query processor parses this query and converts it internally into a series of steps, as shown in Listing 1–17.

```
1.  TABLESCAN( Employees,
                    (DOB = '01-01-1967') )
2.  PROJECT(1., Name)
3.  SORT(2., Name)
```

Listing 1–17. *Query Plan*

In Step 1 of the series, the ORDBMS scans all records in the `Employees` table. If the record satisfies the predicate that is supplied as the second argument to the `TABLESCAN` operation `(DOB = '01-01-1967')`, the matching record is passed into Step 2. In Step 2 the Name column is stripped out of the record and a list consisting of just the names of matching employees is passed into Step 3. In this final step, the names are sorted. The ordered result, the consequence of three separate operations, is returned to the user who submitted the query in the first place.

Clearly, there are many opportunities for optimizations here. For example, if an index existed on the `Employees.DOB` column, the database may elect to use an `INDEXSCAN` rather than the `TABLESCAN`. Also, it might be advantageous to use a slightly different plan that strips out Name and DOB query processing *before* checking if the date-of-birth matches.

The complete list of operations—TABLESCAN, INDEXSCAN, PROJECT, JOIN, SORT, UNION, and so on—is quite extensive, and the IDS product includes several different algorithms for performing the same logical operation. Consider the sorting operation, for example. In Don Knuth's *The Art of Computer Programming Vol. 3: Sorting and Searching*, the author describes over a dozen sorting algorithms, each with slightly different properties. Some work better when the data fits into main memory and others are more efficient when the data must be written to disk. One of the query planner's tasks is to figure out which of the alternate algorithms is the best one in any given circumstance.

ORDBMS Evolution from RDBMS

Let's look at sorting in more detail. One of the most popular sort algorithms is a called *insertion sort*. Insertion sort is an O (N^2) algorithm.

Roughly speaking, the time it takes to sort an array of records increases with the square of the number of records involved, so it should only be used for record sets containing less than about 25 rows. But insertion sort is extremely efficient when the input is almost in sorted order, such as when you are sorting the result of an index scan.

Listing 1–18 presents an implementation of the basic insertion sort algorithm for an array of integers.

```
InsertSort( integer arTypeInput[] )
{
     integer             nTypeTemp;
     integer             InSizeArray, nOuter, nInner;

     nSizeArray = arTypeInput[].getSize();

     for ( nOuter = 2; nOuter <= nSizeArray; nOuter ++)
{
            vTypeTemp = arTypeInput[nOuter];
            nInner = nOuter - 1;

            while (( nInner > 0 ) &&
                    (arTypeInput[nInner] > vTypeTemp)) {
                   arTypeInput[nInner+1] =
arTypeInput[nInner];
                   nInner--;
            }

            arTypeInput[nInner+1] = vTypeTemp;
}
}
```

Listing 1–18. *Straight Insertion Sort Algorithm*

Sorting algorithms such as this can be generalized to make them work with any data type. A generalized version of this insert sort algorithm appears in Listing 1–19. All this algorithm requires is logic to compare two type instances. If one value is greater, the algorithm swaps the two values.

In the generalized version of the algorithm, a *function pointer* is passed into the sort as an argument. A function pointer is simply a reference to a memory address where the `Compare()` function's actual implementation can be found. At the critical point, when the algorithm decides whether to swap two values, it passes the data to this function and makes its decision based on the function's return result.

```
InsertSort( Type arTypeInput[],
             (int) Compare( Type, Type) )
{
```

```
        Type         vTypeTemp;
        integer      InSizeArray, nOuter, nInner;

        nSizeArray = arTypeInput[].getSize();

        for ( nOuter = 2; nOuter <= nSizeArray; nOuter ++)
         {
          vTypeTemp = arTypeInput[nOuter];
          nInner = nOuter - 1;
          while (( nInner > 0 ) &&
                Compare(arTypeInput[nInner],vTypeTemp) > 0 )
          {
            Swap(arTypeInput[nInner+1],arTypeInput[nInner]);
            nInner--;
          }
          Swap(arTypeInput[nInner+1], vTypeTemp);
          }
    }
```

Listing 1–19. *Generalized Sort Algorithm*

Note how the functionality of the Swap() operation is something that can be handled by IDS without requiring a type specific routine. The ORDBMS knows how big the object being swapped is, and whether the object is in memory or is written out to disk. To use the generalized algorithm in Listing 1–19, all that IDS needs is the `Compare()` logic. The ORDBMS handles the looping, branching, and exception handling.

Almost all sorting algorithms involve looping and branching around a `Compare()`, as does B-Tree indexing and aggregate algorithms such as `MIN()` and `MAX()`. Part of the process of extending the ORDBMS framework with new data types involves creating functions such as `Compare()` that IDS can use to manage instances of the type.

All data management operations implemented in the ORDBMS are generalized in this way. In an RDBMS, the number of data types was small so that the `Compare()` routines for each could be hard-coded within the engine. When a query such as the one in Listing 1–16 was received, Step 3 of the query execution plan would involve a call to a function that looked like that which is contained in Listing 1–20.

The first two arguments to this function are a pointer to the records to be sorted and a pointer to a structure that specifies that part of the records should be sorted. Within this function, the engine calls the InsertSort routine introduced above, and passes in the same array of records it received. In addition, however, it passes a pointer to the appropriate compare function for the built-in type.

```
Sort( void ** parRecords, IFX_TYPE * pRecKey ... )
```

```
{
switch(pRecKey->typenum)
      {
         case IFX_INT_TYPE:
         case IFX_CHAR_TYPE:
           InsertSort(parRecords, IntCompare);
         break;
         case IFX_FLOAT_TYPE:
           InsertSort(parRecords, DoubleCompare);
         break;
            // etc for each SQL-92 type
         default:
           ifx_internal_raiseerror('No Such Type');
         break;
       }
}
```

Listing 1–20. *SQL-92 Sort Utility*

How does an ORDBMS know what to pass into the InsertSort() routine? To turn an RDBMS into an ORDBMS, you need to modify the code shown in Listing 1–20 to allow the engine to access `Compare()` routines other than the ones it has built-in. If the data type passed as the second argument is one of the SQL-92 types, the sorting function proceeds as it did before. But if the data type is not one of the SQL-92 types, the ORDBMS assumes it is being asked to sort an extended type.

Every user-defined function embedded within the ORDBMS is recorded in its *system catalogs*, which are tables that the DBMS uses to store information about databases. When asked to sort a pile of records using a user-defined type, the ORDBMS looks in these system catalogs to find a user-defined function called `compare` that takes two instances of the type and returns an `INTEGER`. If such a function exists, the ORDBMS uses it in place of a built-in logic. If the function is not found, IDS generates an exception. Listing 1–21 shows the modified sorting facility.

```
Sort( void ** parRecords, IFX_TYPE * pRecKey . . . )
{
    switch(pRecKey->typenum)
  {
    case IFX_INT_TYPE:
    case IFX_CHAR_TYPE:
           InsertSort(parRecords, IntCompare);
    break;
    case IFX_FLOAT_TYPE:
```

```
                    InsertSort (parRecords, DoubleCompare);
          break;
        // etc for each SQL-92 type
        . .
        default: // ah! Not SQL-92. Must be user-defined.
          if( (pUDRCompare=udr_lookup(pRecKey->typenum,

                                          "Compare")) ==NULL)
              ifx_internal_error("No Such Function.');
           else
              InsertSort (parRecords, pUDRCompare );
          break;
      }
  }
```

Listing 1–21. *ORDBMS Generalization of the Sort Utility*

Using the ORDBMS generalization of the sort utility has several implications:

- When you take a database application developed to use the previous generation of RDBMS technology and upgrade to an ORDBMS, you should see no changes at all. The size of the ORDBMS exe-

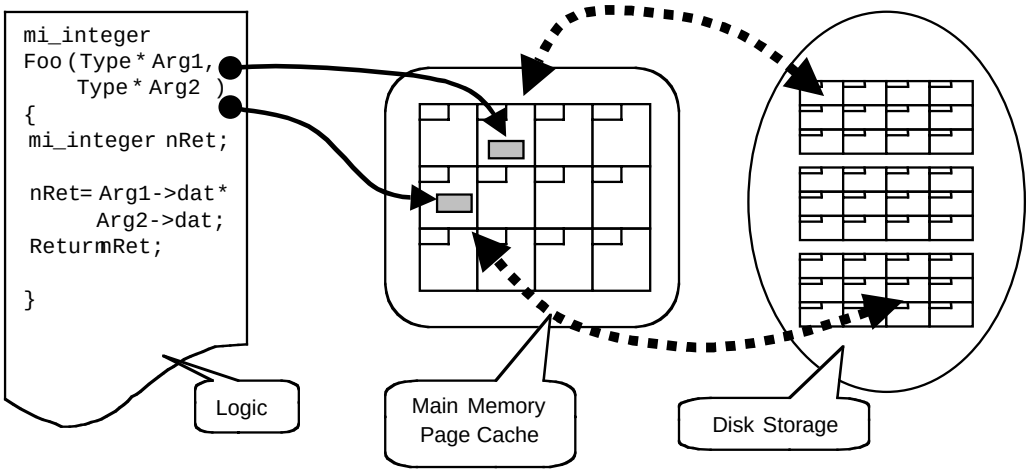


Figure 1–9. *Overview of ORDBMS Table Data Store Management*

[5] Actually, the ORDBMS first copies the data from the page to another memory location. This is done for performance and reliability reasons. User-defined logic is unable to affect what is on pages directly, and the ORDBMS would like to minimize the amount of time a page must be kept resident in the cache.

cutable is slightly increased, and there are some new directories and install options, but if all you do is to run the RDBMS application on the ORDBMS, the new code and facilities are never invoked.
- This scheme implies extensive modifications to the RDBMS code. You not only need to find every place in the engine that such modifications are necessary, but also need to provide the infrastructure in the engine to allow the extensions to execute within it. In Chapter 10, we go into considerable detail about this execution environment.
- Finally, you should note how general such an extensibility mechanism is. As long as you can specify the structure of your data type, and define an algorithm to compare one instance of the type with another, you can use the engine's facilities to sort or index instances of the type, and you can use OR-SQL to specify how you want this done.

In practice, renovating an RDBMS in this manner is an incredibly complex and demanding undertaking.

Data Storage in an ORDBMS

Data storage in an ORDBMS is more or less unchanged from what it was in the RDBMS. Data is organized into pages. A set of pages (possibly very

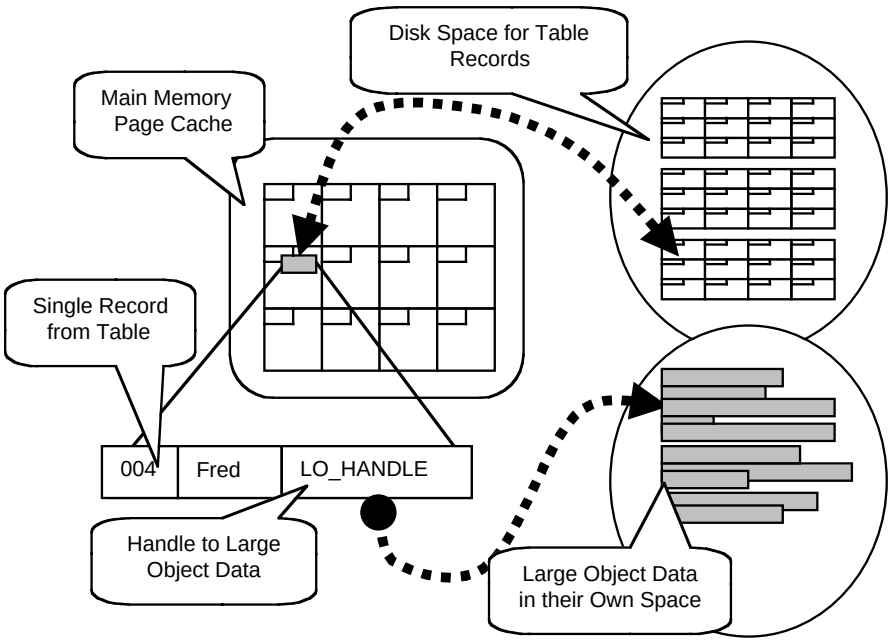


Figure 1–10. *ORDBMS Management of Large Object Data*

many) holds all data for a particular table. Blocks of memory are reserved by the ORDBMS for caching frequently accessed pages to avoid the cost of going to disk each time the page is touched. As data is accessed through queries, pages are read from disk and ejected from the memory cache.

Figure 1–9 illustrates how IDS combines data and logic at run-time. Whenever the ORDBMS invokes some user-defined logic (Foo() in this example) as part of query execution it first reads relevant data pages from disk into the memory cache. Changed pages, or infrequently accessed pages, may be written back to disk if the cache is full. As it invokes the logic, the ORDBMS passes pointers to the memory resident data as arguments.[5] The ORDBMS doesn't care what the function does, or how it does it. All it cares about is the logic's return value.

Each time a single execution of the embedded logic is completed, the ORDBMS is free to read more data from pages in memory and to invoke the logic again with different arguments. The important point is that the ORDBMS does not store the logic with the object data. It binds them together at run-time.

Later in this book we investigate certain aspects of data storage in considerable detail. Chapter 3, for instance, explains how new data types can be created, in Chapter 10, we describe the low-level facilities provided by the ORDBMS for 'C', and Tutorial 1 describes a number of details about IDS physical storage.

Large Object Data Storage

Large object data presents a particular set of challenges. Because extensible database applications tend to involve a lot of large data objects, dealing with them efficiently is particularly important. Data pages used for table records are relatively small: traditionally only a few kilobytes in size. Requiring that new types fit within these pages would be an unacceptable constraint, so IDS provides special mechanisms for supporting data objects of any size.

Storing large object data, such as the polygon corresponding to a state boundary, with table data, such as the state's name and current population, would result in a massive increase in the size of the table. Assuming that fewer queries access the large object data, such a strategy would dramatically increase the time taken to complete the majority of queries. Therefore, the ORDBMS separates the storage of large object data from table data. When a data object's value is stored as a large object only, a *handle for the large object* data is stored with the row. User-defined functions implementing behavior for large objects use this handle to open the large object and access its contents.

Figure 1–10 illustrates how the mechanism works. First, the table's data page is read from its storage location. If one of the table's columns

contains a large data object, space in the record is reserved to hold a string that uniquely identifies a large object stored elsewhere on the local disk. Interface functions within the ORDBMS allow logic components to use this handle like a file name and to access bytes within the large object. This strategy makes it possible to combine large objects with more conventional data within a table's row and to embed logic that manipulates large object data into OR-SQL.

Development Example

This book is intended to be a practical manual of ORDBMS development, so throughout, we provide a great many examples. For consistency, almost all are drawn from a single application: a Web site providing information services about movies. The idea is to provide realistic demonstrations of how to use an ORDBMS and a body of working code to read and draw upon.

Movies-R-Us Web Site Description

This example system depicts a fairly typical electronic commerce (e-commerce) Web site. The primary purpose of the hypothetical site is promotional. The site contains information about the movies produced

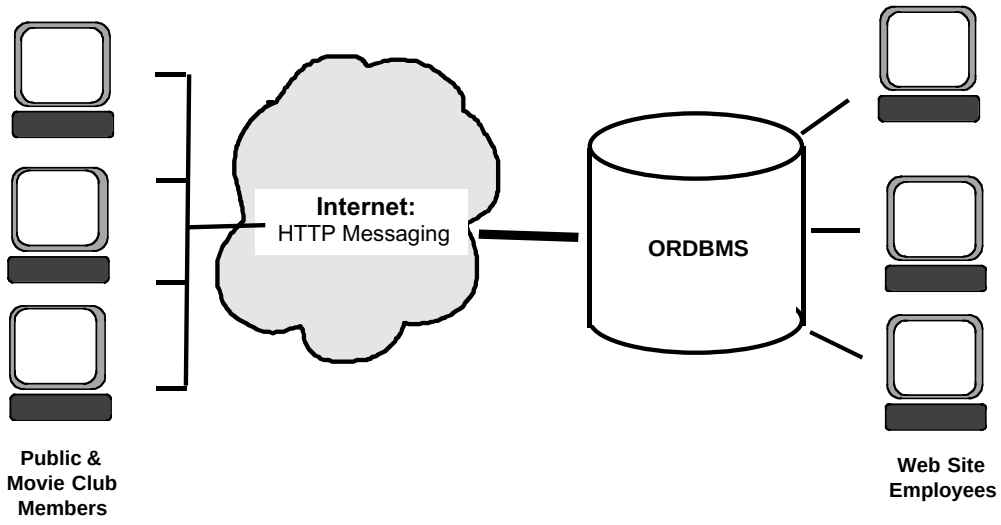


Figure 1–11. *Architecture of Movies-R-Us Web Site*

by various studios for release around the world. Site employees collect and organize marketing material, stills from the movie production process, off-cuts from editing, interviews with stars and movie makers, industry gossip, and so on. The site also sells promotional merchandise: t-shirts, mugs, compact discs, posters, and so on.

The mission of the Web site is to become a single point of reference for movie-going consumers. They can review new releases, find out about screening times and locations, and order merchandise. For example, a visitor to the site might begin by browsing new releases, plot synopses, ratings, and reviews. The visitor may elect to purchase some of the movie merchandise or ask the site for information about where and when the movie is playing. The user may offer reviews of the movies he or she has or join a cinema club through which the movie studio offers discounts.

Site Architecture

The Movies-R-Us Web site is split between a back-of-shop operation and front-of-shop operation. The only component shared by both halves of the system is the ORDBMS. The database mostly stores data used to service the demand of people visiting the site. These public users run a commercial *browser* that sends messages to the site and accepts marked-up pages to display in return. These pages contain content, such as digital image data and text, and sophisticated layout instructions. The browser formats the content according to the layout instructions. In addition, WWW technology includes support for features that allow the public users to upload information to the site. This data can then be inserted into the database.

Employees of the movie studio, on the other hand, use a broader variety of interfaces. They access the public site to review and test it, but they are heavier users of specialist tools for tasks such as multimedia content editing, inserting and updating material in the site, processing merchandising orders, and analyzing usage patterns.

The way that this single database supports multiple user views is consistent with our earlier observation of how database technology is used. Different employees of Movies-R-Us have different tasks relating to the site's data. Public users, depending on their status (whether they are members of the cinema club or have purchased merchandise in the past, for example) get different views of the site.

This set of requirements yields a site architecture that looks like Figure 1–11.

Included with this book is the complete set of source code used in the example. This source code includes the following:

- About a dozen complete database extensions, which are user-

defined types and associated functions that implement various aspects of the overall system
- A set of scripts that create the schema within the ORDBMS
- A few rows of legitimate sample data for the Web Site schema, together with a body of reference data that populates other tables
- Scripts to create bulk data to perform scalability tests on the database implementation
- A set of client-side examples that illustrate how to use the various APIs to interact with the ORDBMS

Structure of this Book

This book is divided into two sections. In the first section, consisting of Chapters 2 through 7, we describe the features and functionality of the ORDBMS in some detail. We explain the ORDBMS data model, query language, the basics of extensibility, certain ideas concerning middleware and the ORDBMS, and the APIs used to integrate the ORDBMS with other components of the overall information system.

Beginning in Chapter 8, we describe how to use these features to build a complete system. This second section describes a semi-formal approach or methodological road map to follow during development and digresses into several important subject areas that are meant for the consumption of programmers working with specialized aspects of the technology.

Along the way there are several tutorials. These are separated from the main body of the text because they are subjects that can stand alone and read at any time. They are included in the book because they cover topics of interest to both the beginning and advanced programmer. They may be skipped on a first reading.

Note on Development Philosophy

In addition to being a book about a specific software technology, this is a book that describes a unique, interesting and hopefully useful way of looking at the whole problem of information systems development. In his great 1905 paper that introduced the idea that light could be thought of as a particle, rather than a wave phenomenon, Albert Einstein referred to his insight as a heuristic viewpoint. By this, he meant

that it was simply a different way of looking at the entire problem and useful to the extent that it helped resolve certain inconsistencies.

The design of this book is also guided by a heuristic viewpoint. We refer to the new idea as *organic* information system development. Historically, software engineers cycled through code-compile-deploy iterations when building information systems. Once deployed, an information system went into "maintenance." The problem with looking at development this way is that "maintenance" seems to consume the majority of development effort.

Organically developed information systems have no maintenance mode, and they are not monolithic systems that must be shut down when the time comes to upgrade them. Instead the approach calls for rapid response to changing end-user requirements by either combining pre-existing components in new ways (through OR-SQL) or adding new components to the system (using extensibility). In general, this can be accomplished without affecting other components or dependent programs. The result is an information system that *grows*, rather than one that is *developed* in the conventional sense.

Some readers will think, perhaps rightly, that this whole idea is just full of it. It is hoped that such readers can nevertheless learn something of interest in these pages.

Chapter Summary

In this chapter we have introduced object-relational DBMS technology, illustrated something of the role ORDBMSs will play in developing information systems to support the modern enterprise, and briefly described the major components of the technology. An ORDBMS

- is a data repository that can be extended to manage any kind of data and to organize any kind of data processing,
- represents the next logical step in the evolution of DBMS technology. It both preserves those aspects of relational DBMS technology that proved so successful (abstraction, parallelism, and transaction management) and augments the features with ideas, such as extensibility and advanced data modeling, derived from object-oriented approaches to software engineering,
- can be thought of as a *software back plane*, which is a general framework within which new pieces can be integrated as occasion demands.