25 OBJECT-DATABASE SYSTEMS

with Joseph M. Hellerstein
U. C. Berkeley

You know my methods, Watson. Apply them.

—Arthur Conan Doyle, *The Memoirs of Sherlock Holmes*

Relational database systems support a small, fixed collection of data types (e.g., integers, dates, strings), which has proven adequate for traditional application domains such as administrative data processing. In many application domains, however, much more complex kinds of data must be handled. Typically this complex data has been stored in OS file systems or specialized data structures, rather than in a DBMS. Examples of domains with complex data include computer-aided design and modeling (CAD/CAM), multimedia repositories, and document management.

As the amount of data grows, the many features offered by a DBMS—for example, reduced application development time, concurrency control and recovery, indexing support, and query capabilities—become increasingly attractive and, ultimately, necessary. In order to support such applications, a DBMS must support complex data types. Object-oriented concepts have strongly influenced efforts to enhance database support for complex data and have led to the development of object-database systems, which we discuss in this chapter.

Object-database systems have developed along two distinct paths:

- **Object-oriented database systems:** Object-oriented database systems are proposed as an alternative to relational systems and are aimed at application domains where complex objects play a central role. The approach is heavily influenced by object-oriented programming languages and can be understood as an attempt to add DBMS functionality to a programming language environment.
- **Object-relational database systems:** Object-relational database systems can be thought of as an attempt to extend relational database systems with the functionality necessary to support a broader class of applications and, in many ways, provide a bridge between the relational and object-oriented paradigms.

We will use acronyms for relational database management systems **(RDBMS)**, object-oriented database management systems **(OODBMS)**, and object-relational database management systems **(ORDBMS)**. In this chapter we focus on ORDBMSs and emphasize how they can be viewed as a development of RDBMSs, rather than as an entirely different paradigm.

The SQL:1999 standard is based on the ORDBMS model, rather than the OODBMS model. The standard includes support for many of the complex data type features discussed in this chapter. We have concentrated on developing the fundamental concepts, rather than on presenting SQL:1999; some of the features that we discuss are not included in SQL:1999. We have tried to be consistent with SQL:1999 for notation, although we have occasionally diverged slightly for clarity. It is important to recognize that the main concepts discussed are common to both ORDBMSs and OODBMSs, and we discuss how they are supported in the ODL/OQL standard proposed for OODBMSs in Section 25.8.

RDBMS vendors, including IBM, Informix, and Oracle, are adding ORDBMS functionality (to varying degrees) in their products, and it is important to recognize how the existing body of knowledge about the design and implementation of relational databases can be leveraged to deal with the ORDBMS extensions. It is also important to understand the challenges and opportunities that these extensions present to database users, designers, and implementors.

In this chapter, sections 25.1 through 25.5 motivate and introduce object-oriented concepts. The concepts discussed in these sections are common to both OODBMSs and ORDBMSs, even though our syntax is similar to SQL:1999. We begin by presenting an example in Section 25.1 that illustrates why extensions to the relational model are needed to cope with some new application domains. This is used as a running example throughout the chapter. We discuss how abstract data types can be defined and manipulated in Section 25.2 and how types can be composed into structured types in Section 25.3. We then consider objects and object identity in Section 25.4 and inheritance and type hierarchies in Section 25.5.

We consider how to take advantage of the new object-oriented concepts to do ORDBMS database design in Section 25.6. In Section 25.7, we discuss some of the new implementation challenges posed by object-relational systems. We discuss ODL and OQL, the standards for OODBMSs, in Section 25.8, and then present a brief comparison of ORDBMSs and OODBMSs in Section 25.9.

25.1 MOTIVATING EXAMPLE

As a specific example of the need for object-relational systems, we focus on a new business data processing problem that is both harder and (in our view) more entertaining than the dollars and cents bookkeeping of previous decades. Today, companies in industries such as entertainment are in the business of selling *bits*; their basic corporate assets are not tangible products, but rather software artifacts such as video and audio.

We consider the fictional Dinky Entertainment Company, a large Hollywood conglomerate whose main assets are a collection of cartoon characters, especially the cuddly and internationally beloved Herbert the Worm. Dinky has a number of Herbert the Worm films, many of which are being shown in theaters around the world at any given time. Dinky also makes a good deal of money licensing Herbert's image, voice, and video footage for various purposes: action figures, video games, product endorsements, and so on. Dinky's database is used to manage the sales and leasing records for the various Herbert-related products, as well as the video and audio data that make up Herbert's many films.

25.1.1 New Data Types

A basic problem confronting Dinky's database designers is that they need support for considerably richer data types than is available in a relational DBMS:

- **User-defined abstract data types (ADTs):** Dinky's assets include Herbert's image, voice, and video footage, and these must be stored in the database. Further, we need special functions to manipulate these objects. For example, we may want to write functions that produce a compressed version of an image or a lower-resolution image. (See Section 25.2.)
- **Structured types:** In this application, as indeed in many traditional business data processing applications, we need new types built up from atomic types using constructors for creating sets, tuples, arrays, sequences, and so on. (See Section 25.3.)
- **Inheritance:** As the number of data types grows, it is important to recognize the commonality between different types and to take advantage of it. For example, compressed images and lower-resolution images are both, at some level, just images. It is therefore desirable to *inherit* some features of image objects while defining (and later manipulating) compressed image objects and lower-resolution image objects. (See Section 25.5.)

How might we address these issues in an RDBMS? We could store images, videos, and so on as *BLOBs* in current relational systems. A **binary large object (BLOB)** is just a long stream of bytes, and the DBMS's support consists of storing and retrieving BLOBs in such a manner that a user does not have to worry about the size of the BLOB; a BLOB can span several pages, unlike a traditional attribute. All further processing of the BLOB has to be done by the user's application program, in the host language in which the SQL code is embedded. This solution is not efficient because we

Large objects in SQL: SQL:1999 includes a new data type called `LARGE OBJECT` or `LOB`, with two variants called `BLOB` (binary large object) and `CLOB` (character large object). This standardizes the large object support found in many current relational DBMSs. LOBs cannot be included in primary keys, `GROUP BY`, or `ORDER BY` clauses. They can be compared using equality, inequality, and substring operations. A LOB has a **locator** that is essentially a unique id and allows LOBs to be manipulated without extensive copying.
LOBs are typically stored separately from the data records in whose fields they appear. IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all support LOBs.

are forced to retrieve all BLOBs in a collection even if most of them could be filtered out of the answer by applying user-defined functions (within the DBMS). It is not satisfactory from a data consistency standpoint either, because the semantics of the data is now heavily dependent on the host language application code and cannot be enforced by the DBMS.

As for structured types and inheritance, there is simply no support in the relational model. We are forced to map data with such complex structure into a collection of flat tables. (We saw examples of such mappings when we discussed the translation from ER diagrams with inheritance to relations in Chapter 2.)

This application clearly requires features that are not available in the relational model. As an illustration of these features, Figure 25.1 presents SQL:1999 DDL statements for a portion of Dinky's ORDBMS schema that will be used in subsequent examples. Although the DDL is very similar to that of a traditional relational system, it has some important distinctions that highlight the new data modeling capabilities of an ORDBMS. A quick glance at the DDL statements is sufficient for now; we will study them in detail in the next section, after presenting some of the basic concepts that our sample application suggests are needed in a next-generation DBMS.

25.1.2 Manipulating the New Kinds of Data

Thus far, we have described the new kinds of data that must be stored in the Dinky database. We have not yet said anything about how to *use* these new types in queries, so let's study two queries that Dinky's database needs to support. The syntax of the queries is not critical; it is sufficient to understand what they express. We will return to the specifics of the queries' syntax as we proceed.

Our first challenge comes from the Clog breakfast cereal company. Clog produces a cereal called Delirios, and it wants to lease an image of Herbert the Worm in front of

```
1. CREATE TABLE Frames
      (frameno integer, image jpeg_image, category integer);
2. CREATE TABLE Categories
      (cid integer, name text, lease_price float, comments text);
3. CREATE TYPE theater_t AS
      ROW(tno integer, name text, address text, phone text);
4. CREATE TABLE Theaters OF theater_t;
5. CREATE TABLE Nowshowing
      (film integer, theater ref(theater_t) with scope Theaters, start date, end date
6. CREATE TABLE Films
      (filmno integer, title text, stars setof(text),
      director text, budget float);
7. CREATE TABLE Countries
      (name text, boundary polygon, population integer, language text);
```

Figure 25.1 SQL:1999 DDL Statements for Dinky Schema

a sunrise, to incorporate in the Delirios box design. A query to present a collection of possible images and their lease prices can be expressed in SQL-like syntax as in Figure 25.2. Dinky has a number of methods written in an imperative language like Java and registered with the database system. These methods can be used in queries in the same way as built-in methods, such as $=, +, -, <, >$, are used in a relational language like SQL. The *thumbnail* method in the `Select` clause produces a small version of its full-size input image. The *is_sunrise* method is a boolean function that analyzes an image and returns *true* if the image contains a sunrise; the *is_herbert* method returns *true* if the image contains a picture of Herbert. The query produces the frame code number, image thumbnail, and price for all frames that contain Herbert and a sunrise.

```
SELECT  F.frameno, thumbnail(F.image), C.lease_price
FROM    Frames F, Categories C
WHERE   F.category = C.cid AND is_sunrise(F.image) AND is_herbert(F.image)
```

Figure 25.2 Extended SQL to Find Pictures of Herbert at Sunrise

The second challenge comes from Dinky's executives. They know that Delirios is exceedingly popular in the tiny country of Andorra, so they want to make sure that a number of Herbert films are playing at theaters near Andorra when the cereal hits the shelves. To check on the current state of affairs, the executives want to find the names of all theaters showing Herbert films within 100 kilometers of Andorra. Figure 25.3 shows this query in an SQL-like syntax.

```
SELECT  N.theater–>name, N.theater–>address, F.title
FROM    Nowshowing N, Films F, Countries C
WHERE   N.film = F.filmno AND
        overlaps(C.boundary, radius(N.theater–>address, 100)) AND
        C.name = ‘Andorra’ AND ‘Herbert the Worm’ ∈ F.stars
```

Figure 25.3 Extended SQL to Find Herbert Films Playing near Andorra

The *theater* attribute of the Nowshowing table is a reference to an object in another table, which has attributes *name, address*, and *location.* This object referencing allows for the notation *N.theater–>name* and *N.theater–>address*, each of which refers to attributes of the `theater_t` object referenced in the Nowshowing row N. The *stars* attribute of the *films* table is a set of names of each film's stars. The *radius* method returns a circle centered at its first argument with radius equal to its second argument. The `overlaps` method tests for spatial overlap. Thus, Nowshowing and Films are joined by the equijoin clause, while Nowshowing and Countries are joined by the spatial overlap clause. The selections to ‘Andorra’ and films containing ‘Herbert the Worm’ complete the query.

These two object-relational queries are similar to SQL-92 queries but have some unusual features:

- **User-defined methods:** User-defined abstract types are manipulated via their methods, for example, *is_herbert* (Section 25.2).
- **Operators for structured types:** Along with the structured types available in the data model, ORDBMSs provide the natural methods for those types. For example, the **setof** types have the standard set methods $\in, \ni, \subset, \subseteq, =, \supseteq, \supset, \cup, \cap$, and $-$ (Section 25.3.1).
- **Operators for reference types:** Reference types are *dereferenced* via an arrow (–>) notation (Section 25.4.2).

To summarize the points highlighted by our motivating example, traditional relational systems offer limited flexibility in the data types available. Data is stored in tables, and the type of each field value is limited to a simple atomic type (e.g., integer or string), with a small, fixed set of such types to choose from. This limited type system can be extended in three main ways: *user-defined abstract data types, structured types,* and *reference types.* Collectively, we refer to these new types as **complex types**. In the rest of this chapter we consider how a DBMS can be extended to provide support for defining new complex types and manipulating objects of these new types.

25.2 USER-DEFINED ABSTRACT DATA TYPES

Consider the Frames table of Figure 25.1. It has a column *image* of type `jpeg_image`, which stores a compressed image representing a single frame of a film. The `jpeg_image` type is not one of the DBMS's built-in types and was defined by a user for the Dinky application to store image data compressed using the JPEG standard. As another example, the Countries table defined in Line 7 of Figure 25.1 has a column *boundary* of type `polygon`, which contains representations of the shapes of countries' outlines on a world map.

Allowing users to define arbitrary new data types is a key feature of ORDBMSs. The DBMS allows users to store and retrieve objects of type `jpeg_image`, just like an object of any other type, such as `integer`. New atomic data types usually need to have type-specific operations defined by the user who creates them. For example, one might define operations on an image data type such as `compress`, `rotate`, `shrink`, and `crop`. The combination of an atomic data type and its associated methods is called an **abstract data type**, or **ADT**. Traditional SQL comes with built-in ADTs, such as integers (with the associated arithmetic methods), or strings (with the equality, comparison, and `LIKE` methods). Object-relational systems include these ADTs and also allow users to define their own ADTs.

The label 'abstract' is applied to these data types because the database system does not need to know how an ADT's data is stored nor how the ADT's methods work. It merely needs to know what methods are available and the input and output types for the methods. Hiding of ADT internals is called **encapsulation**.[1] Note that even in a relational system, atomic types such as integers have associated methods that are encapsulated into ADTs. In the case of integers, the standard methods for the ADT are the usual arithmetic operators and comparators. To evaluate the addition operator on integers, the database system need not understand the laws of addition—it merely needs to know how to invoke the addition operator's code and what type of data to expect in return.

In an object-relational system, the simplification due to encapsulation is critical because it hides any substantive distinctions between data types and allows an ORDBMS to be implemented without anticipating the types and methods that users might want to add. For example, adding integers and overlaying images can be treated uniformly by the system, with the only significant distinctions being that different code is invoked for the two operations and differently typed objects are expected to be returned from that code.

[1]Some ORDBMSs actually refer to ADTs as **opaque types** because they are encapsulated and hence one cannot see their details.

Packaged ORDBMS extensions: Developing a set of user-defined types and methods for a particular application—say image management—can involve a significant amount of work and domain-specific expertise. As a result, most ORDBMS vendors partner with third parties to sell prepackaged sets of ADTs for particular domains. Informix calls these extensions *DataBlades*, Oracle calls them *Data Cartridges*, IBM calls them *DB2 Extenders*, and so on. These packages include the ADT method code, DDL scripts to automate loading the ADTs into the system, and in some cases specialized access methods for the data type. Packaged ADT extensions are analogous to class libraries that are available for object-oriented programming languages: They provide a set of objects that together address a common task.

25.2.1 Defining Methods of an ADT

At a minimum, for each new atomic type a user must define methods that enable the DBMS to read in and to output objects of this type and to compute the amount of storage needed to hold the object. The user who creates a new atomic type must **register** the following methods with the DBMS:

- **Size**: Returns the number of bytes of storage required for items of the type or the special value *variable*, if items vary in size.
- **Import**: Creates new items of this type from textual inputs (e.g., `INSERT` statements).
- **Export**: Maps items of this type to a form suitable for printing, or for use in an application program (e.g., an ASCII string or a file handle).

In order to register a new method for an atomic type, users must write the code for the method and then inform the database system about the method. The code to be written depends on the languages supported by the DBMS, and possibly the operating system in question. For example, the ORDBMS may handle Java code in the Linux operating system. In this case the method code must be written in Java and compiled into a Java bytecode file stored in a Linux file system. Then an SQL-style method registration command is given to the ORDBMS so that it recognizes the new method:

```
CREATE FUNCTION is_sunrise(jpeg_image) RETURNS boolean
       AS EXTERNAL NAME ‘/a/b/c/dinky.class’ LANGUAGE ’java’;
```

This statement defines the salient aspects of the method: the type of the associated ADT, the return type, and the location of the code. Once the method is registered, the DBMS uses a Java virtual machine to execute the code[2]. Figure 25.4 presents a number of method registration commands for our Dinky database.

```
1. CREATE FUNCTION thumbnail(jpeg_image) RETURNS jpeg_image
        AS EXTERNAL NAME '/a/b/c/dinky.class' LANGUAGE 'java';
2. CREATE FUNCTION is_sunrise(jpeg_image) RETURNS boolean
        AS EXTERNAL NAME '/a/b/c/dinky.class' LANGUAGE 'java';
3. CREATE FUNCTION is_herbert(jpeg_image) RETURNS boolean
        AS EXTERNAL NAME '/a/b/c/dinky.class' LANGUAGE 'java';
4. CREATE FUNCTION radius(polygon, float) RETURNS polygon
        AS EXTERNAL NAME '/a/b/c/dinky.class' LANGUAGE 'java';
5. CREATE FUNCTION overlaps(polygon, polygon) RETURNS boolean
        AS EXTERNAL NAME '/a/b/c/dinky.class' LANGUAGE 'java';
```

Figure 25.4 Method Registration Commands for the Dinky Database

Type definition statements for the user-defined atomic data types in the Dinky schema are given in Figure 25.5.

1. `CREATE ABSTRACT DATA TYPE jpeg_image`
   (*internallength* = `VARIABLE`, *input* = jpeg_in, *output* = jpeg_out);
2. `CREATE ABSTRACT DATA TYPE polygon`
   (*internallength* = `VARIABLE`, *input* = poly_in, *output* = poly_out);

Figure 25.5 Atomic Type Declaration Commands for Dinky Database

25.3 STRUCTURED TYPES

Atomic types and user-defined types can be combined to describe more complex structures using **type constructors**. For example, Line 6 of Figure 25.1 defines a column *stars* of type `setof(text)`; each entry in that column is a set of text strings, representing the stars in a film. The `setof` syntax is an example of a type constructor. Other common type constructors include:

- `ROW`(n_1 t_1, ..., n_n t_n): A type representing a row, or tuple, of n fields with fields $n_1, ..., n_n$ of types $t_1, ..., t_n$ respectively.
- `listof(base)`: A type representing a sequence of `base`-type items.
- `ARRAY(base)`: A type representing an array of `base`-type items.
- `setof(base)`: A type representing a *set* of `base`-type items. Sets cannot contain duplicate elements.

[2]In the case of non-portable compiled code – written, for example, in a language like C++ – the DBMS uses the operating system's dynamic linking facility to link the method code into the database system so that it can be invoked.

Structured data types in SQL: The `theater_t` type in Figure 25.1 illustrates the new `ROW` data type in SQL:1999; a value of `ROW` type can appear in a field of a tuple. In SQL:1999 the `ROW` type has a special role because every table is a collection of rows—every table is a set of rows or a multiset of rows. SQL:1999 also includes a data type called `ARRAY`, which allows a field value to be an array. The `ROW` and `ARRAY` type constructors can be freely interleaved and nested to build structured objects. The `listof`, `bagof`, and `setof` type constructors are not included in SQL:1999. IBM DB2, Informix UDS, and Oracle 8 support the `ROW` constructor.

- `bagof(base)`: A type representing a *bag* or *multiset* of `base`-type items.

To fully appreciate the power of type constructors, observe that they can be composed; for example, `ARRAY(ROW(`*age:* `integer`, *sal:* `integer))`. Types defined using type constructors are called **structured types**. Those using `listof`, `ARRAY`, `bagof`, or `setof` as the outermost type constructor are sometimes referred to as **collection types**, or **bulk data types**.

The introduction of structured types changes a fundamental characteristic of relational databases, which is that all fields contain atomic values. A relation that contains a structured type object is not in first normal form! We discuss this point further in Section 25.6.

25.3.1 Manipulating Data of Structured Types

The DBMS provides built-in methods for the types supported through type constructors. These methods are analogous to built-in operations such as addition and multiplication for atomic types such as integers. In this section we present the methods for various type constructors and illustrate how SQL queries can create and manipulate values with structured types.

Built-in Operators for Structured Types

We now consider built-in operators for each of the structured types that we presented in Section 25.3.

Rows: Given an item i whose type is `ROW(`$n_1\ t_1, ..., n_n\ t_n$`)`, the field extraction method allows us to access an individual field n_k using the traditional dot notation $i.n_k$. If row constructors are nested in a type definition, dots may be nested to access the fields of the nested row; for example $i.n_k.m_l$. If we have a collection of rows, the dot notation gives us a collection as a result. For example, if i is a list of rows, $i.n_k$ gives us a list of items of type t_n; if i is a set of rows, $i.n_k$ gives us a set of items of type t_n.

This nested-dot notation is often called a **path expression** because it describes a path through the nested structure.

Sets and multisets: Set objects can be compared using the traditional set methods $\subset, \subseteq, =, \supseteq, \supset$. An item of type `setof(foo)` can be compared with an item of type `foo` using the $\in$ method, as illustrated in Figure 25.3, which contains the comparison *'Herbert the Worm'* $\in$ *F.stars*. Two set objects (having elements of the same type) can be combined to form a new object using the $\cup$, $\cap$, and $-$ operators.

Each of the methods for sets can be defined for multisets, taking the number of copies of elements into account. The $\cup$ operation simply adds up the number of copies of an element, the $\cap$ operation counts the lesser number of times a given element appears in the two input multisets, and $-$ subtracts the number of times a given element appears in the second multiset from the number of times it appears in the first multiset. For example, using multiset semantics $\cup(\{1,2,2,2\}, \{2,2,3\}) = \{1,2,2,2,2,2,3\}$; $\cap(\{1,2,2,2\}, \{2,2,3\}) = \{2,2\}$; and $-(\{1,2,2,2\}, \{2,2,3\}) = \{1,2\}$.

Lists: Traditional list operations include *head*, which returns the first element; *tail*, which returns the list obtained by removing the first element; *prepend*, which takes an element and inserts it as the first element in a list; and *append*, which appends one list to another.

Arrays: Array types support an 'array index' method to allow users to access array items at a particular offset. A postfix 'square bracket' syntax is usually used; for example, `foo_array[5]`.

Other: The operators listed above are just a sample. We also have the aggregate operators *count*, *sum*, *avg*, *max*, and *min*, which can (in principle) be applied to any object of a collection type. Operators for type conversions are also common. For example, we can provide operators to convert a multiset object to a set object by eliminating duplicates.

Examples of Queries Involving Nested Collections

We now present some examples to illustrate how relations that contain nested collections can be queried, using SQL syntax. Consider the Films relation. Each tuple describes a film, uniquely identified by *filmno*, and contains a set (of stars in the film) as a field value. Our first example illustrates how we can apply an aggregate operator to such a nested set. It identifies films with more than two stars by counting the number of stars; the *count* operator is applied once per Films tuple.[3]

```
SELECT F.filmno
FROM   Films F
WHERE  count(F.stars) > 2
```

Our second query illustrates an operation called **unnesting**. Consider the instance of Films shown in Figure 25.6; we have omitted the *director* and *budget* fields (included in the Films schema in Figure 25.1) for simplicity. A flat version of the same information is shown in Figure 25.7; for each film and star in the film, we have a tuple in Films_flat.

filmno	*title*	*stars*
98	Casablanca	{Bogart, Bergman}
54	Earth Worms Are Juicy	{Herbert, Wanda}

Figure 25.6 A Nested Relation, Films

filmno	*title*	*star*
98	Casablanca	Bogart
98	Casablanca	Bergman
54	Earth Worms Are Juicy	Herbert
54	Earth Worms Are Juicy	Wanda

Figure 25.7 A Flat Version, Films_flat

The following query generates the instance of Films_flat from Films:

```
SELECT F.filmno, F.title, S AS star
FROM   Films F, F.stars AS S
```

The variable F is successively bound to tuples in Films, and for each value of F, the variable S is successively bound to the set in the *stars* field of F. Conversely, we may want to generate the instance of Films from Films_flat. We can generate the Films instance using a generalized form of SQL's GROUP BY construct, as the following query illustrates:

```
SELECT   F.filmno, F.title, set_gen(F.star)
FROM     Films_flat F
GROUP BY F.filmno, F.title
```

[3]SQL:1999 limits the use of aggregate operators on nested collections; to emphasize this restriction, we have used *count* rather than COUNT, which we reserve for legal uses of the operator in SQL.

Objects and oids: In SQL:1999 every tuple in a table can be given an oid by defining the table in terms of a structured type, as in the definition of the Theaters table in Line 4 of Figure 25.1. Contrast this with the definition of the Countries table in Line 7; Countries tuples do not have associated oids. SQL:1999 also assigns oids to large objects: this is the locator for the object.
There is a special type called `REF` whose values are the unique identifiers or oids. SQL:1999 requires that a given `REF` type must be associated with a specific structured type and that the table it refers to must be known at compilation time, i.e., the scope of each reference must be a table known at compilation time. For example, Line 5 of Figure 25.1 defines a column *theater* of type `ref(theater_t)`. Items in this column are references to objects of type `theater_t`, specifically the rows in the Theaters table, which is defined in Line 4. IBM DB2, Informix UDS, and Oracle 8 support `REF` types.

The operator *set_gen*, to be used with `GROUP BY`, requires some explanation. The `GROUP BY` clause partitions the Films_flat table by sorting on the *filmno* attribute; all tuples in a given partition have the same *filmno* (and therefore the same *title*). Consider the set of values in the *star* column of a given partition. This set cannot be returned in the result of an SQL-92 query, and we have to summarize it by applying an aggregate operator such as `COUNT`. Now that we allow relations to contain sets as field values, however, we would like to return the set of *star* values as a field value in a single answer tuple; the answer tuple also contains the *filmno* of the corresponding partition. The *set_gen* operator collects the set of *star* values in a partition and creates a set-valued object. This operation is called **nesting**. We can imagine similar generator functions for creating multisets, lists, and so on. However, such generators are not included in SQL:1999.

25.4 OBJECTS, OBJECT IDENTITY, AND REFERENCE TYPES

In object-database systems, data objects can be given an **object identifier (oid)**, which is some value that is unique in the database across time. The DBMS is responsible for generating oids and ensuring that an oid identifies an object uniquely over its entire lifetime. In some systems, all tuples stored in any table are objects and are automatically assigned unique oids; in other systems, a user can specify the tables for which the tuples are to be assigned oids. Often, there are also facilities for generating oids for larger structures (e.g., tables) as well as smaller structures (e.g., instances of data values such as a copy of the integer 5, or a JPEG image).

An object's oid can be used to refer (or 'point') to it from elsewhere in the data. Such a reference has a type (similar to the type of a pointer in a programming language), with a corresponding type constructor:

URLs and oids: It is instructive to note the differences between Internet URLs and the oids in object systems. First, oids uniquely identify a single object over all time, whereas the web resource pointed at by an URL can change over time. Second, oids are simply identifiers and carry no physical information about the objects they identify—this makes it possible to change the storage location of an object without modifying pointers to the object. In contrast, URLs include network addresses and often file-system names as well, meaning that if the resource identified by the URL has to move to another file or network address, then all links to that resource will either be incorrect or require a 'forwarding' mechanism. Third, oids are automatically generated by the DBMS for each object, whereas URLs are user-generated. Since users generate URLs, they often embed semantic information into the URL via machine, directory, or file names; this can become confusing if the object's properties change over time.
In the case of both URLs and oids, deletions can be troublesome: In an object database this can result in runtime errors during dereferencing; on the web this is the notorious '404 Page Not Found' error. The relational mechanisms for referential integrity are not available in either case.

`ref(base)`: a type representing a reference to an object of type `base`.

The `ref` type constructor can be interleaved with the type constructors for structured types; for example, `ROW(ref(ARRAY(integer)))`.

25.4.1 Notions of Equality

The distinction between reference types and reference-free structured types raises another issue: the definition of equality. Two objects having the same type are defined to be **deep equal** if and only if:

- The objects are of atomic type and have the same value, or
- The objects are of reference type, and the *deep equals* operator is true for the two referenced objects, or
- The objects are of structured type, and the *deep equals* operator is true for all the corresponding subparts of the two objects.

Two objects that have the same reference type are defined to be **shallow equal** if they both refer to the same object (i.e., both references use the same oid). The definition of shallow equality can be extended to objects of arbitrary type by taking the definition of deep equality and replacing *deep equals* by *shallow equals* in parts (2) and (3).

As an example, consider the complex objects `ROW(`538, **`t89`**, 6-3-97,8-7-97`)` and `ROW(`538, **`t33`**, 6-3-97,8-7-97`)`, whose type is the type of rows in the table Nowshowing (Line 5 of Figure 25.1). These two objects are not shallow equal because they differ in the second attribute value. Nonetheless, they might be deep equal, if, for instance, the oids **`t89`** and **`t33`** refer to objects of type `theater_t` that have the same value; for example, `tuple(`54, 'Majestic', '115 King', '2556698'`)`.

While two deep equal objects may not be shallow equal, as the example illustrates, two shallow equal objects are always deep equal, of course. The default choice of deep versus shallow equality for reference types is different across systems, although typically we are given syntax to specify either semantics.

25.4.2 Dereferencing Reference Types

An item of reference type `ref(foo)` is not the same as the `foo` item to which it points. In order to access the referenced `foo` item, a built-in `deref()` method is provided along with the `ref` type constructor. For example, given a tuple from the Nowshowing table, one can access the *name* field of the referenced `theater_t` object with the syntax Nowshowing.`deref`*(theater).name.* Since references to tuple types are common, some systems provide a java-style arrow operator, which combines a postfix version of the dereference operator with a tuple-type dot operator. Using the arrow notation, the name of the referenced theater can be accessed with the equivalent syntax Nowshowing.*theater–>name*, as in Figure 25.3.

At this point we have covered all the basic type extensions used in the Dinky schema in Figure 25.1. The reader is invited to revisit the schema and to examine the structure and content of each table and how the new features are used in the various sample queries.

25.5 INHERITANCE

We considered the concept of inheritance in the context of the ER model in Chapter 2 and discussed how ER diagrams with inheritance were translated into tables. In object-database systems, unlike relational systems, inheritance is supported directly and allows type definitions to be reused and refined very easily. It can be very helpful when modeling similar but slightly different classes of objects. In object-database systems, inheritance can be used in two ways: for reusing and refining types, and for creating hierarchies of collections of similar but not identical objects.

25.5.1 Defining Types with Inheritance

In the Dinky database, we model movie theaters with the type `theater_t`. Dinky also wants their database to represent a new marketing technique in the theater business: the *theater-cafe*, which serves pizza and other meals while screening movies. Theater-cafes require additional information to be represented in the database. In particular, a theater-cafe is just like a theater, but has an additional attribute representing the theater's menu. Inheritance allows us to capture this 'specialization' explicitly in the database design with the following DDL statement:

```
CREATE TYPE theatercafe_t UNDER theater_t (menu text);
```

This statement creates a new type, `theatercafe_t`, which has the same attributes and methods as `theater_t`, along with one additional attribute *menu* of type `text`. Methods defined on `theater_t` apply to objects of type `theatercafe_t`, but not vice versa. We say that `theatercafe_t` **inherits** the attributes and methods of `theater_t`.

Note that the inheritance mechanism is not merely a 'macro' to shorten `CREATE` statements. It creates an explicit relationship in the database between the **subtype** (`theatercafe_t`) and the **supertype** (`theater_t`): *An object of the subtype is also considered to be an object of the supertype.* This treatment means that any operations that apply to the supertype (methods as well as query operators such as projection or join) also apply to the subtype. This is generally expressed in the following principle:

> **The Substitution Principle**: Given a supertype A and a subtype B, it is always possible to substitute an object of type B into a legal expression written for objects of type A, without producing type errors.

This principle enables easy code reuse because queries and methods written for the supertype can be applied to the subtype without modification.

Note that inheritance can also be used for atomic types, in addition to row types. Given a supertype `image_t` with methods *title(), number_of_colors(),* and *display()*, we can define a subtype `thumbnail_image_t` for small images that inherits the methods of `image_t`.

25.5.2 Binding of Methods

In defining a subtype, it is sometimes useful to replace a method for the supertype with a new version that operates differently on the subtype. Consider the `image_t` type, and the subtype `jpeg_image_t` from the Dinky database. Unfortunately, the *display()* method for standard images does not work for JPEG images, which are specially compressed. Thus, in creating type `jpeg_image_t`, we write a special *display()* method for JPEG images and register it with the database system using the `CREATE FUNCTION` command:

```
CREATE FUNCTION display(jpeg_image) RETURNS jpeg_image
       AS EXTERNAL NAME '/a/b/c/jpeg.class' LANGUAGE 'java';
```

Registering a new method with the same name as an old method is called **overloading** the method name.

Because of overloading, the system must understand which method is intended in a particular expression. For example, when the system needs to invoke the *display()* method on an object of type `jpeg_image_t`, it uses the specialized *display* method. When it needs to invoke *display* on an object of type `image_t` that is not otherwise subtyped, it invokes the standard *display* method. The process of deciding which method to invoke is called **binding** the method to the object. In certain situations, this binding can be done when an expression is parsed (**early binding**), but in other cases the most specific type of an object cannot be known until runtime, so the method cannot be bound until then (**late binding**). Late binding facilties add flexibility, but can make it harder for the user to reason about the methods that get invoked for a given query expression.

25.5.3 Collection Hierarchies, Type Extents, and Queries

Type inheritance was invented for object-oriented programming languages, and our discussion of inheritance up to this point differs little from the discussion one might find in a book on an object-oriented language such as C++ or Java.

However, because database systems provide query languages over tabular datasets, the mechanisms from programming languages are enhanced in object databases to deal with tables and queries as well. In particular, in object-relational systems we can define a table containing objects of a particular type, such as the Theaters table in the Dinky schema. Given a new subtype such as `theater_cafe`, we would like to create another table Theater_cafes to store the information about theater cafes. But when writing a query over the Theaters table, it is sometimes desirable to ask the same query over the Theater_cafes table; after all, if we project out the additional columns, an instance of the Theater_cafes table can be regarded as an instance of the Theaters table.

Rather than requiring the user to specify a separate query for each such table, we can inform the system that a new table of the subtype is to be treated as part of a table of the supertype, with respect to queries over the latter table. In our example, we can say:

```
CREATE TABLE Theater_cafes OF TYPE theater_cafe_t UNDER Theaters;
```

This statement tells the system that queries over the `theaters` table should actually be run over all tuples in both the `theaters` and `Theater_cafes` tables. In such cases, if the subtype definition involves method overloading, late-binding is used to ensure that the appropriate methods are called for each tuple.

In general, the `UNDER` clause can be used to generate an arbitrary tree of tables, called a **collection hierarchy**. Queries over a particular table T in the hierarchy are run over all tuples in T and its descendants. Sometimes, a user may want the query to run only on T, and not on the descendants; additional syntax, for example, the keyword `ONLY`, can be used in the query's `FROM` clause to achieve this effect.

Some systems automatically create special tables for each type, which contain references to every instance of the type that exists in the database. These tables are called **type extents** and allow queries over all objects of a given type, regardless of where the objects actually reside in the database. Type extents naturally form a collection hierarchy that parallels the type hierarchy.

25.6 DATABASE DESIGN FOR AN ORDBMS

The rich variety of data types in an ORDBMS offers a database designer many opportunities for a more natural or more efficient design. In this section we illustrate the differences between RDBMS and ORDBMS database design through several examples.

25.6.1 Structured Types and ADTs

Our first example involves several space probes, each of which continuously records a video. A single video stream is associated with each probe, and while this stream was collected over a certain time period, we assume that it is now a complete object associated with the probe. During the time period over which the video was collected, the probe's location was periodically recorded (such information can easily be 'piggy-backed' onto the header portion of a video stream conforming to the MPEG standard). Thus, the information associated with a probe has three parts: (1) a *probe id* that identifies a probe uniquely, (2) a *video stream*, and (3) a *location sequence* of $\langle$*time, location*$\rangle$ pairs. What kind of a database schema should we use to store this information?

An RDBMS Database Design

In an RDBMS, we must store each video stream as a BLOB and each location sequence as tuples in a table. A possible RDBMS database design is illustrated below:

Probes(*pid:* `integer`, *time:* `timestamp`, *lat:* `real`, *long:* `real`, *camera:* `string`, *video:* `BLOB`)

There is a single table called Probes, and it has several rows for each probe. Each of these rows has the same *pid*, *camera*, and *video* values, but different *time*, *lat*, and *long* values. (We have used latitude and longitude to denote location.) The key for this table can be represented as a functional dependency: $PTLN \rightarrow CV$, where N stands for longitude. There is another dependency: $P \rightarrow CV$. This relation is therefore not in BCNF; indeed, it is not even in 3NF. We can decompose Probes to obtain a BCNF schema:

Probes_Loc(*pid:* `integer`, *time:* `timestamp`, *lat:* `real`, *long:* `real`)
Probes_Video(*pid:* `integer`, *camera:* `string`, *video:* `BLOB`)

This design is about the best we can achieve in an RDBMS. However, it suffers from several drawbacks.

First, representing videos as BLOBs means that we have to write application code in an external language to manipulate a video object in the database. Consider this query: "For probe 10, display the video recorded between 1:10 p.m. and 1:15 p.m. on May 10 1996." We have to retrieve the entire video object associated with probe 10, recorded over several hours, in order to display a segment recorded over 5 minutes.

Next, the fact that each probe has an associated sequence of location readings is obscured, and the sequence information associated with a probe is dispersed across several tuples. A third drawback is that we are forced to separate the video information from the sequence information for a probe. These limitations are exposed by queries that require us to consider all the information associated with each probe; for example, "For each probe, print the earliest time at which it recorded, and the camera type." This query now involves a join of Probes_Loc and Probes_Video on the *pid* field.

An ORDBMS Database Design

An ORDBMS supports a much better solution. First, we can store the video as an ADT object and write methods that capture any special manipulation that we wish to perform. Second, because we are allowed to store structured types such as lists, we can store the location sequence for a probe in a single tuple, along with the video information! This layout eliminates the need for joins in queries that involve both the sequence and video information. An ORDBMS design for our example consists of a single relation called Probes_AllInfo:

Probes_AllInfo(*pid:* `integer`, *locseq:* `location_seq`, *camera:* `string`,
video: `mpeg_stream`)

This definition involves two new types, `location_seq` and `mpeg_stream`. The `mpeg_stream` type is defined as an ADT, with a method *display()* that takes a start time and an end time and displays the portion of the video recorded during that interval. This method can be implemented efficiently by looking at the total recording duration and the total length of the video and interpolating to extract the segment recorded during the interval specified in the query.

Our first query is shown below in extended SQL syntax; using this *display* method: We now retrieve only the required segment of the video, rather than the entire video.

```
SELECT  display(P.video, 1:10 p.m. May 10 1996, 1:15 p.m. May 10 1996)
FROM    Probes_AllInfo P
WHERE   P.pid = 10
```

Now consider the `location_seq` type. We could define it as a `list` type, containing a list of `ROW` type objects:

```
CREATE TYPE location_seq listof
       (row (time: timestamp, lat: real, long: real))
```

Consider the *locseq* field in a row for a given probe. This field contains a list of rows, each of which has three fields. If the ORDBMS implements collection types in their full generality, we should be able to extract the *time* column from this list to obtain a list of `timestamp` values, and to apply the `MIN` aggregate operator to this list to find the earliest time at which the given probe recorded. Such support for collection types would enable us to express our second query as shown below:

```
SELECT   P.pid, MIN(P.locseq.time)
FROM     Probes_AllInfo P
```

Current ORDBMSs are not as general and clean as this example query suggests. For instance, the system may not recognize that projecting the *time* column from a list of rows gives us a list of timestamp values; or the system may allow us to apply an aggregate operator only to a table and not to a nested list value.

Continuing with our example, we may want to do specialized operations on our location sequences that go beyond the standard aggregate operators. For instance, we may want to define a method that takes a time interval and computes the distance traveled by the probe during this interval. The code for this method must understand details of a probe's trajectory and geospatial coordinate systems. For these reasons, we might choose to define `location_seq` as an ADT.

Clearly, an (ideal) ORDBMS gives us many useful design options that are not available in an RDBMS.

25.6.2 Object Identity

We now discuss some of the consequences of using reference types or oids. The use of oids is especially significant when the size of the object is large, either because it is a structured data type or because it is a big object such as an image.

Although reference types and structured types seem similar, they are actually quite different. For example, consider a structured type `my_theater tuple(`*tno* `integer`, *name* `text`, *address* `text`, *phone* `text)` and the reference type `theater ref(theater_t)` of Figure 25.1. There are important differences in the way that database updates affect these two types:

- **Deletion:** Objects with references can be affected by the deletion of objects that they reference, while reference-free structured objects are not affected by deletion of other objects. For example, if the Theaters table were dropped from the database, an object of type `theater` might change value to *null*, because the `theater_t` object that it refers to has been deleted, while a similar object of type `my_theater` would not change value.
- **Update:** Objects of reference types will change value if the referenced object is updated. Objects of reference-free structured types change value only if updated directly.
- **Sharing versus copying:** An identified object can be referenced by multiple reference-type items, so that each update to the object is reflected in many places. To get a similar affect in reference-free types requires updating all 'copies' of an object.

There are also important storage distinctions between reference types and nonreference types, which might affect performance:

- **Storage overhead:** Storing copies of a large value in multiple structured type objects may use much more space than storing the value once and referring to it elsewhere through reference type objects. This additional storage requirement can affect both disk usage and buffer management (if many copies are accessed at once).
- **Clustering:** The subparts of a structured object are typically stored together on disk. Objects with references may point to other objects that are far away on the disk, and the disk arm may require significant movement to assemble the object and its references together. Structured objects can thus be more efficient than reference types if they are typically accessed in their entirety.

Many of these issues also arise in traditional programming languages such as C or Pascal, which distinguish between the notions of referring to objects *by value* and *by*

Oids and referential integrity: In SQL:1999, all the oids that appear in a column of a relation are required to reference the same target relation. This 'scoping' makes it possible to check oid references for 'referential integrity' just as foreign key references are checked. While current ORDBMS products supporting oids do not support such checks, it is likely that they will do so in future releases. This will make it much safer to use oids.

reference. In database design, the choice between using a structured type or a reference type will typically include consideration of the storage costs, clustering issues, and the effect of updates.

Object Identity versus Foreign Keys

Using an oid to refer to an object is similar to using a foreign key to refer to a tuple in another relation, but not quite the same: An oid can point to an object of `theater_t` that is stored *anywhere* in the database, even in a field, whereas a foreign key reference is constrained to point to an object in a particular referenced relation. This restriction makes it possible for the DBMS to provide much greater support for referential integrity than for arbitrary oid pointers. In general, if an object is deleted while there are still oid-pointers to it, the best the DBMS can do is to recognize the situation by maintaining a reference count. (Even this limited support becomes impossible if oids can be copied freely.) Thus, the responsibility for avoiding dangling references rests largely with the user if oids are used to refer to objects. This burdensome responsibility suggests that we should use oids with great caution and use foreign keys instead whenever possible.

25.6.3 Extending the ER Model

The ER model as we described it in Chapter 2 is not adequate for ORDBMS design. We have to use an extended ER model that supports structured attributes (i.e., sets, lists, arrays as attribute values), distinguishes whether entities have object ids, and allows us to model entities whose attributes include methods. We illustrate these comments using an extended ER diagram to describe the space probe data in Figure 25.8; our notational conventions are ad hoc, and only for illustrative purposes.

The definition of Probes in Figure 25.8 has two new aspects. First, it has a structured-type attribute `listof(row(`*time, lat, long*`))`; each value assigned to this attribute in a Probes entity is a list of tuples with three fields. Second, Probes has an attribute called videos that is an abstract data type object, which is indicated by a dark oval for this attribute with a dark line connecting it to Probes. Further, this attribute has an 'attribute' of its own, which is a method of the ADT.

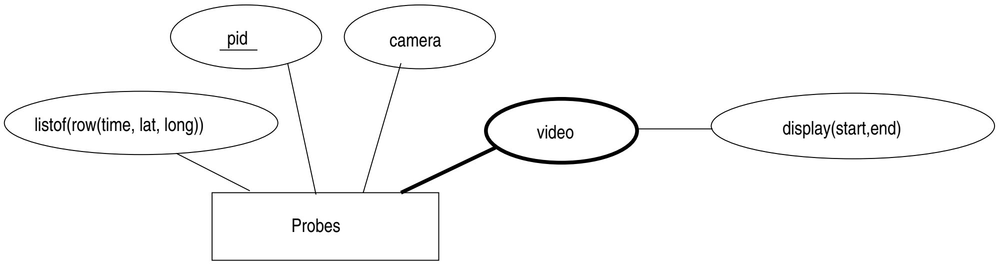

Figure 25.8 The Space Probe Entity Set

Alternatively, we could model each video as an entity by using an entity set called Videos. The association between Probes entities and Videos entities could then be captured by defining a relationship set that links them. Since each video is collected by precisely one probe, and every video is collected by some probe, this relationship can be maintained by simply storing a reference to a probe object with each Videos entity; this technique is essentially the second translation approach from ER diagrams to tables discussed in Section 2.4.1.

If we also make Videos a weak entity set in this alternative design, we can add a referential integrity constraint that causes a Videos entity to be deleted when the corresponding Probes entity is deleted. More generally, this alternative design illustrates a strong similarity between storing references to objects and foreign keys; the foreign key mechanism achieves the same effect as storing oids, but in a controlled manner. If oids are used, the user must ensure that there are no dangling references when an object is deleted, with very little support from the DBMS.

Finally, we note that a significant extension to the ER model is required to support the design of nested collections. For example, if a location sequence is modeled as an entity, and we want to define an attribute of Probes that contains a set of such entities, there is no way to do this without extending the ER model. We will not discuss this point further at the level of ER diagrams, but consider an example below that illustrates when to use a nested collection.

25.6.4 Using Nested Collections

Nested collections offer great modeling power, but also raise difficult design decisions. Consider the following way to model location sequences (other information about probes is omitted here to simplify the discussion):

Probes1(<u>*pid:* `integer`</u>, *locseq:* `location_seq`)

This is a good choice if the important queries in the workload require us to look at the location sequence for a particular probe, as in the query "For each probe, print the earliest time at which it recorded, and the camera type." On the other hand, consider a query that requires us to look at all location sequences: "Find the earliest time at which a recording exists for *lat=5, long=90*." This query can be answered more efficiently if the following schema is used:

Probes2(*pid:* `integer`, *time:* `timestamp`, *lat:* `real`, *long:* `real`)

The choice of schema must therefore be guided by the expected workload (as always!). As another example, consider the following schema:

Can_Teach1(*cid:* `integer`, *teachers:* `setof`(*ssn:* `string`), *sal:* `integer`)

If tuples in this table are to be interpreted as "Course *cid* can be taught by any of the teachers in the *teachers* field, at a cost *sal*" then we have the option of using the following schema instead:

Can_Teach2(*cid:* `integer`, *teacher_ssn:* `string`, *sal:* `integer`)

A choice between these two alternatives can be made based on how we expect to query this table. On the other hand, suppose that tuples in Can_Teach1 are to be interpreted as "Course *cid* can be taught by the team *teachers*, at a combined cost of *sal*." Can_Teach2 is no longer a viable alternative. If we wanted to flatten Can_Teach1, we would have to use a separate table to encode teams:

Can_Teach2(*cid:* `integer`, *team_id:* `oid`, *sal:* `integer`)
Teams(*tid:* `oid`, *ssn:* `string`)

As these examples illustrate, nested collections are appropriate in certain situations, but this feature can easily be misused; nested collections should therefore be used with care.

25.7 NEW CHALLENGES IN IMPLEMENTING AN ORDBMS

The enhanced functionality of ORDBMSs raises several implementation challenges. Some of these are well understood and solutions have been implemented in products; others are subjects of current research. In this section we examine a few of the key challenges that arise in implementing an efficient, fully functional ORDBMS. Many more issues are involved than those discussed here; the interested reader is encouraged to revisit the previous chapters in this book and consider whether the implementation techniques described there apply naturally to ORDBMSs or not.

25.7.1 Storage and Access Methods

Since object-relational databases store new types of data, ORDBMS implementors need to revisit some of the storage and indexing issues discussed in earlier chapters. In particular, the system must efficiently store ADT objects and structured objects and provide efficient indexed access to both.

Storing Large ADT and Structured Type Objects

Large ADT objects and structured objects complicate the layout of data on disk. This problem is well understood and has been solved in essentially all ORDBMSs and OODBMSs. We present some of the main issues here.

User-defined ADTs can be quite large. In particular, they can be bigger than a single disk page. Large ADTs, like BLOBs, require special storage, typically in a different location on disk from the tuples that contain them. Disk-based pointers are maintained from the tuples to the objects they contain.

Structured objects can also be large, but unlike ADT objects they often vary in size during the lifetime of a database. For example, consider the *stars* attribute of the *films* table in Figure 25.1. As the years pass, some of the 'bit actors' in an old movie may become famous.[4] When a bit actor becomes famous, Dinky might want to advertise his or her presence in the earlier films. This involves an insertion into the *stars* attribute of an individual tuple in *films*. Because these bulk attributes can grow arbitrarily, flexible disk layout mechanisms are required.

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

$$A_{11}, \ldots A_{1n}, A_{21}, \ldots, A_{2n}, \ldots A_{m1}, \ldots, A_{mn}$$

However, queries may often request subarrays that are not stored contiguously on disk (e.g., $A_{11}, A_{21}, \ldots, A_{m1}$). Such requests can result in a very high I/O cost for retrieving the subarray. In order to reduce the number of I/Os required in general, arrays are often broken into contiguous *chunks*, which are then stored in some order on disk. Although each chunk is some contiguous region of the array, chunks need not be row-by-row or column-by-column. For example, a chunk of size 4 might be $A_{11}, A_{12}, A_{21}, A_{22}$, which is a square region if we think of the array as being arranged row-by-row in two dimensions.

[4] A well-known example is Marilyn Monroe, who had a bit part in the Bette Davis classic *All About Eve*.

Indexing New Types

One important reason for users to place their data in a database is to allow for efficient access via indexes. Unfortunately, the standard RDBMS index structures support only equality conditions (B+ trees and hash indexes) and range conditions (B+ trees). An important issue for ORDBMSs is to provide efficient indexes for ADT methods and operators on structured objects.

Many specialized index structures have been proposed by researchers for particular applications such as cartography, genome research, multimedia repositories, Web search, and so on. An ORDBMS company cannot possibly implement every index that has been invented. Instead, the set of index structures in an ORDBMS should be user-extensible. Extensibility would allow an expert in cartography, for example, to not only register an ADT for points on a map (i.e., latitude/longitude pairs), but also implement an index structure that supports natural map queries (e.g., the R-tree, which matches conditions such as "Find me all theaters within 100 miles of Andorra"). (See Chapter 26 for more on R-trees and other spatial indexes.)

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

25.7.2 Query Processing

ADTs and structured types call for new functionality in processing queries in ORDBMSs. They also change a number of assumptions that affect the efficiency of queries. In this section we look at two functionality issues (user-defined aggregates and security) and two efficiency issues (method caching and pointer swizzling).

User-Defined Aggregation Functions

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

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

Method Security

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

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

An alternative mechanism is to allow user methods to be compiled from a general-purpose programming language such as C++, but to run those methods in a different address space than the DBMS. In this case the DBMS sends explicit interprocess communications (IPCs) to the user method, which sends IPCs back in return. This approach prevents bugs in the user methods (e.g., stray pointers) from corrupting the state of the DBMS or database and prevents malicious methods from reading or modifying the DBMS state or database as well. Note that the user writing the method need not know that the DBMS is running the method in a separate process: The user code can be linked with a 'wrapper' that turns method invocations and return values into IPCs.

Method Caching

User-defined ADT methods can be very expensive to execute and can account for the bulk of the time spent in processing a query. During query processing it may make sense to cache the results of methods, in case they are invoked multiple times with the same argument. Within the scope of a single query, one can avoid calling a method twice on duplicate values in a column by either sorting the table on that column or using a hash-based scheme much like that used for aggregation (see Section 12.7). An alternative is to maintain a *cache* of method inputs and matching outputs as a table in the database. Then to find the value of a method on particular inputs, we essentially join the input tuples with the cache table. These two approaches can also be combined.

Pointer Swizzling

In some applications, objects are retrieved into memory and accessed frequently through their oids; dereferencing must be implemented very efficiently. Some systems maintain a table of oids of objects that are (currently) in memory. When an object O is brought into memory, they check each oid contained in O and replace oids of in-memory objects by in-memory pointers to those objects. This technique is called **pointer swizzling** and makes references to in-memory objects very fast. The downside is that when an object is paged out, in-memory references to it must somehow be invalidated and replaced with its oid.

25.7.3 Query Optimization

New indexes and query processing techniques widen the choices available to a query optimizer. In order to handle the new query processing functionality, an optimizer must know about the new functionality and use it appropriately. In this section we discuss two issues in exposing information to the optimizer (new indexes and ADT method estimation) and an issue in query planning that was ignored in relational systems (expensive selection optimization).

Registering Indexes with the Optimizer

As new index structures are added to a system—either via external interfaces or built-in template structures like GiSTs—the optimizer must be informed of their existence, and their costs of access. In particular, for a given index structure the optimizer must know (a) what `WHERE`-clause conditions are matched by that index, and (b) what the

Optimizer extensibility: As an example, consider the Oracle 8i optimizer, which is extensible and supports user defined 'domain' indexes and methods. The support includes user defined statistics and cost functions that the optimizer will use in tandem with system statistics. Suppose that there is a domain index for text on the *resume* column and a regular Oracle B-tree index on *hiringdate*. A query with a selection on both these fields can be evaluated by converting the rids from the two indexes into bitmaps, performing a bitmap `AND`, and converting the resulting bitmap to rids before accessing the table. Of course, the optimizer will also consider using the two indexes individually, as well as a full table scan.

cost of fetching a tuple is for that index. Given this information, the optimizer can use any index structure in constructing a query plan. Different ORDBMSs vary in the syntax for registering new index structures. Most systems require users to state a number representing the cost of access, but an alternative is for the DBMS to measure the structure as it is used and maintain running statistics on cost.

Reduction Factor and Cost Estimation for ADT Methods

In Section 14.2.1 we discussed how to estimate the reduction factor of various selection and join conditions including $=$, $<$, and so on. For user-defined conditions such as *is_herbert()*, the optimizer also needs to be able to estimate reduction factors. Estimating reduction factors for user-defined conditions is a difficult problem and is being actively studied. The currently popular approach is to leave it up to the user—a user who registers a method can also register an auxiliary function to estimate the method's reduction factor. If such a function is not registered, the optimizer uses an arbitrary value such as $\frac{1}{10}$.

ADT methods can be quite expensive and it is important for the optimizer to know just how much these methods cost to execute. Again, estimating method costs is open research. In current systems users who register a method are able to specify the method's cost as a number, typically in units of the cost of an I/O in the system. Such estimation is hard for users to do accurately. An attractive alternative is for the ORDBMS to run the method on objects of various sizes and attempt to estimate the method's cost automatically, but this approach has not been investigated in detail and is not implemented in commercial ORDBMSs.

Expensive selection optimization

In relational systems, selection is expected to be a zero-time operation. For example, it requires no I/Os and few CPU cycles to test if *emp.salary* $<$ *10*. However, conditions such as *is_herbert(Frames.image)* can be quite expensive because they may fetch large objects off the disk and process them in memory in complicated ways.

ORDBMS optimizers must consider carefully how to order selection conditions. For example, consider a selection query that tests tuples in the Frames table with two conditions: $Frames.frameno < 100 \wedge is\_herbert(Frame.image)$. It is probably preferable to check the *frameno* condition before testing *is_herbert*. The first condition is quick and may often return false, saving the trouble of checking the second condition. In general, the best ordering among selections is a function of their costs and reduction factors. It can be shown that selections should be ordered by increasing *rank*, where rank = (reduction factor − 1)/cost. If a selection with very high rank appears in a multi-table query, it may even make sense to postpone the selection until after performing joins. Note that this approach is the opposite of the heuristic for pushing selections presented in Section 14.3! The details of optimally placing expensive selections among joins are somewhat complicated, adding to the complexity of optimization in ORDBMSs.

25.8 OODBMS

In the introduction of this chapter, we defined an OODBMS as a programming language with support for persistent objects. While this definition reflects the origins of OODBMSs accurately, and to a certain extent the implementation focus of OODBMSs, the fact that OODBMSs support *collection types* (see Section 25.3) makes it possible to provide a query language over collections. Indeed, a standard has been developed by the Object Database Management Group (ODMG) and is called **Object Query Language**, or **OQL**.

OQL is similar to SQL, with a `SELECT–FROM–WHERE`–style syntax (even `GROUP BY`, `HAVING`, and `ORDER BY` are supported) and many of the proposed SQL:1999 extensions. Notably, OQL supports structured types, including sets, bags, arrays, and lists. The OQL treatment of collections is more uniform than SQL:1999 in that it does not give special treatment to collections of rows; for example, OQL allows the aggregate operation `COUNT` to be applied to a list to compute the length of the list. OQL also supports reference types, path expressions, ADTs and inheritance, type extents, and SQL-style nested queries. There is also a standard Data Definition Language for OODBMSs (**Object Data Language**, or **ODL**) that is similar to the DDL subset of SQL, but supports the additional features found in OODBMSs, such as ADT definitions.

25.8.1 The ODMG Data Model and ODL

The ODMG data model is the basis for an OODBMS, just like the relational data model is the basis for an RDBMS. A database contains a collection of **objects**, which

Class = interface + implementation: Properly speaking, a class consists of an interface together with an implementation of the interface. An ODL interface definition is implemented in an OODBMS by translating it into declarations of the object-oriented language (e.g., C++, Smalltalk or Java) supported by the OODBMS. If we consider C++, for instance, there is a library of classes that implement the ODL constructs. There is also an **Object Manipulation Language** (**OML**) specific to the programming language (in our example, C++), which specifies how database objects are manipulated in the programming language. The goal is to seamlessly integrate the programming language and the database features.

are similar to entities in the ER model. Every object has a unique oid, and a database contains collections of objects with similar properties; such a collection is called a **class**.

The properties of a class are specified using ODL and are of three kinds: *attributes*, *relationships*, and *methods*. **Attributes** have an atomic type or a structured type. ODL supports the `set`, `bag`, `list`, `array`, and `struct` type constructors; these are just `setof`, `bagof`, `listof`, `ARRAY`, and `ROW` in the terminology of Section 25.3.

Relationships have a type that is either a reference to an object or a collection of such references. A relationship captures how an object is related to one or more objects of the same class or of a different class. A relationship in the ODMG model is really just a binary relationship in the sense of the ER model. A relationship has a corresponding **inverse relationship**; intuitively, it is the relationship 'in the other direction.' For example, if a movie is being shown at several theaters, and each theater shows several movies, we have two relationships that are inverses of each other: *shownAt* is associated with the class of movies and is the set of theaters at which the given movie is being shown, and *nowShowing* is associated with the class of theaters and is the set of movies being shown at that theater.

Methods are functions that can be applied to objects of the class. There is no analog to methods in the ER or relational models.

The keyword `interface` is used to define a class. For each interface, we can declare an **extent**, which is the name for the current set of objects of that class. The extent is analogous to the instance of a relation, and the interface is analogous to the schema. If the user does not anticipate the need to work with the set of objects of a given class—it is sufficient to manipulate individual objects—the extent declaration can be omitted.

The following ODL definitions of the Movie and Theater classes illustrate the above concepts. (While these classes bear some resemblance to the Dinky database schema, the reader should not look for an exact parallel, since we have modified the example to highlight ODL features.)

```
interface Movie
        (extent Movies key movieName)
        { attribute date start;
        attribute date end;
        attribute string moviename;
        relationship Set⟨Theater⟩ shownAt inverse Theater::nowShowing;
        }
```

The collection of database objects whose class is Movie is called Movies. No two objects in Movies have the same *movieName* value, as the key declaration indicates. Each movie is shown at a set of theaters and is shown during the specified period. (It would be more realistic to associate a different period with each theater, since a movie is typically played at different theaters over different periods. While we can define a class that captures this detail, we have chosen a simpler definition for our discussion.) A theater is an object of class Theater, which is defined below:

```
interface Theater
        (extent Theaters key theaterName)
        { attribute string theaterName;
        attribute string address;
        attribute integer ticketPrice;
        relationship Set⟨Movie⟩ nowShowing inverse Movie::shownAt;
        float numshowing() raises(errorCountingMovies);
        }
```

Each theater shows several movies and charges the same ticket price for every movie. Observe that the *shownAt* relationship of Movie and the *nowShowing* relationship of Theater are declared to be inverses of each other. Theater also has a method *numshowing()* that can be applied to a theater object to find the number of movies being shown at that theater.

ODL also allows us to specify inheritance hierarchies, as the following class definition illustrates:

```
interface SpecialShow extends Movie
        (extent SpecialShows)
        { attribute integer maximumAttendees;
        attribute string benefitCharity;
        }
```

An object of class SpecialShow is an object of class Movie, with some additional properties, as discussed in Section 25.5.

25.8.2 OQL

The ODMG query language OQL was deliberately designed to have syntax similar to SQL, in order to make it easy for users familiar with SQL to learn OQL. Let us begin with a query that finds pairs of movies and theaters such that the movie is shown at the theater and the theater is showing more than one movie:

```
SELECT mname: M.movieName, tname: T.theaterName
FROM   Movies M, M.shownAt T
WHERE  T.numshowing() > 1
```

The SELECT clause indicates how we can give names to fields in the result; the two result fields are called *mname* and *tname*. The part of this query that differs from SQL is the FROM clause. The variable M is bound in turn to each movie in the extent Movies. For a given movie M, we bind the variable T in turn to each theater in the collection *M.shownAt*. Thus, the use of the path expression *M.shownAt* allows us to easily express a nested query. The following query illustrates the grouping construct in OQL:

```
SELECT   T.ticketPrice,
         avgNum: AVG(SELECT P.T.numshowing() FROM partition P)
FROM     Theaters T
GROUP BY T.ticketPrice
```

For each ticket price, we create a group of theaters with that ticket price. This group of theaters is the partition for that ticket price and is referred to using the OQL keyword `partition`. In the SELECT clause, for each ticket price, we compute the average number of movies shown at theaters in the partition for that ticketPrice. OQL supports an interesting variation of the grouping operation that is missing in SQL:

```
SELECT   low, high,
         avgNum: AVG(SELECT P.T.numshowing() FROM partition P)
FROM     Theaters T
GROUP BY low: T.ticketPrice < 5, high: T.ticketPrice >= 5
```

The GROUP BY clause now creates just two partitions called *low* and *high*. Each theater object T is placed in one of these partitions based on its ticket price. In the SELECT clause, *low* and *high* are boolean variables, exactly one of which is `true` in any given output tuple; `partition` is instantiated to the corresponding partition of theater objects. In our example, we get two result tuples. One of them has *low* equal to `true` and *avgNum* equal to the average number of movies shown at theaters with a low ticket price. The second tuple has *high* equal to `true` and *avgNum* equal to the average number of movies shown at theaters with a high ticket price.

The next query illustrates OQL support for queries that return collections other than set and multiset:

```
(SELECT   T.theaterName
FROM      Theaters T
ORDER BY  T.ticketPrice DESC) [0:4]
```

The `ORDER BY` clause makes the result a list of theater names ordered by ticket price. The elements of a list can be referred to by position, starting with position 0. Thus, the expression [0:4] extracts a list containing the names of the five theaters with the highest ticket prices.

OQL also supports `DISTINCT`, `HAVING`, explicit nesting of subqueries, view definitions, and other SQL features.

25.9 COMPARING RDBMS WITH OODBMS AND ORDBMS

Now that we have covered the main object-oriented DBMS extensions, it is time to consider the two main variants of object-databases, OODBMSs and ORDBMSs, and to compare them with RDBMSs. Although we have presented the concepts underlying object-databases, we still need to define the terms OODBMS and ORDBMS.

An **ORDBMS** is a relational DBMS with the extensions discussed in this chapter. (Not all ORDBMS systems support all the extensions in the general form that we have discussed them, but our concern in this section is the paradigm itself rather than specific systems.) An **OODBMS** is a programming language with a type system that supports the features discussed in this chapter and allows any data object to be **persistent**, that is, to survive across different program executions. Many current systems conform to neither definition entirely but are much closer to one or the other, and can be classified accordingly.

25.9.1 RDBMS versus ORDBMS

Comparing an RDBMS with an ORDBMS is straightforward. An RDBMS does not support the extensions discussed in this chapter. The resulting simplicity of the data model makes it easier to optimize queries for efficient execution, for example. A relational system is also easier to use because there are fewer features to master. On the other hand, it is less versatile than an ORDBMS.

25.9.2 OODBMS versus ORDBMS: Similarities

OODBMSs and ORDBMSs both support user-defined ADTs, structured types, object identity and reference types, and inheritance. Both support a query language for manipulating collection types. ORDBMSs support an extended form of SQL, and OODBMSs support ODL/OQL. The similarities are by no means accidental: ORDBMSs consciously try to add OODBMS features to an RDBMS, and OODBMSs in turn have developed query languages based on relational query languages. Both OODBMSs and ORDBMSs provide DBMS functionality such as concurrency control and recovery.

25.9.3 OODBMS versus ORDBMS: Differences

The fundamental difference is really a philosophy that is carried all the way through: OODBMSs try to add DBMS functionality to a programming language, whereas ORDBMSs try to add richer data types to a relational DBMS. Although the two kinds of object-databases are converging in terms of functionality, this difference in their underlying philosophies (and for most systems, their implementation approach) has important consequences in terms of the issues emphasized in the design of these DBMSs, and the efficiency with which various features are supported, as the following comparison indicates:

- OODBMSs aim to achieve seamless integration with a programming language such as C++, Java or Smalltalk. Such integration is not an important goal for an ORDBMS. SQL:1999, like SQL-92, allows us to embed SQL commands in a host language, but the interface is very evident to the SQL programer. (SQL:1999 also provides extended programming language constructs of its own, incidentally.)

- An OODBMS is aimed at applications where an object-centric viewpoint is appropriate; that is, typical user sessions consist of retrieving a few objects and working on them for long periods, with related objects (e.g., objects referenced by the original objects) fetched occasionally. Objects may be extremely large, and may have to be fetched in pieces; thus, attention must be paid to buffering parts of objects. It is expected that most applications will be able to cache the objects they require in memory, once the objects are retrieved from disk. Thus, considerable attention is paid to making references to in-memory objects efficient. Transactions are likely to be of very long duration and holding locks until the end of a transaction may lead to poor performance; thus, alternatives to Two Phase locking must be used.

  An ORDBMS is optimized for applications where large data collections are the focus, even though objects may have rich structure and be fairly large. It is expected that applications will retrieve data from disk extensively, and that optimizing disk accesses is still the main concern for efficient execution. Transactions are assumed to be relatively short, and traditional RDBMS techniques are typically used for concurrency control and recovery.

- The query facilities of OQL are not supported efficiently in most OODBMSs, whereas the query facilities are the centerpiece of an ORDBMS. To some extent, this situation is the result of different concentrations of effort in the development of these systems. To a significant extent, it is also a consequence of the systems' being optimized for very different kinds of applications.

25.10 POINTS TO REVIEW

- *Object-oriented database systems (OODBMSs)* add DBMS functionality to a programming language and environment. *Object-relational database systems (ORDBMSs)* extend the functionality of relational database systems. The data type system is extended with *user-defined abstract data types (ADTs)*, *structured types*, and *inheritance*. New query features include *operators for structured types*, *operators for reference types*, and *user-defined methods*. **(Section 25.1)**

- An *abstract data type* is an atomic data type and its associated methods. Users can create new ADTs. For a new atomic type, we must *register* the methods *size*, *import*, and *export* with the DBMS. Object files containing new methods can also be registered. **(Section 25.2)**

- We can construct more complex data types from atomic types and user-defined types using *type constructors*. There are type constructors for creating *row types*, *lists*, *arrays*, *sets*, and *bags* and there are built-in operators for all these types. We can *unnest* a set-valued type by creating a tuple for each element in the set. The reverse operation is called *nesting*. **(Section 25.3)**

- Data objects can have an *object identifier (oid)*, which is a unique value that identifies the object. An oid has a type called a *reference type*. Since fields within objects can be of reference types, there are two notions of equality, *deep* and *shallow equality*. Fields that contain a reference type can be *dereferenced* to access the associated object. **(Section 25.4)**

- *Inheritance* allows us to create new types (called *subtypes*) that extend existing types (called *supertypes*). Any operations that apply to the supertype also apply to the subtype. We can *overload* methods by defining the same method for sub- and supertypes. The type of the object decides which method is called; this process is called *binding*. Analogous to types, we can create an inheritance hierarchy for tables called a *collection hierarchy*. **(Section 25.5)**

- The multitude of data types in an ORDBMS allows us to design a more natural and efficient database schema. But we have to be careful to take the differences between reference types and structured types and between reference types and nonreference types into account. We should use oids with caution and use foreign keys instead whenever possible. We can extend the ER model to incorporate ADTs and methods, or we can model ADTs using existing constructs. Nested types provide great modeling power but the choice of schema should be guided by the expected workload. **(Section 25.6)**

- Implementing an ORDBMS brings new challenges. The system must store large ADTs and structured types that might be very large. Efficient and extensible index mechanisms must be provided. Examples of new functionality include *user-defined aggregation functions* (we can define new aggregation functions for our ADTs) and *method security* (the system has to prevent user-defined methods from compromising the security of the DBMS). Examples of new techniques to increase performance include *method caching* and *pointer swizzling*. The optimizer must know about the new functionality and use it appropriately. **(Section 25.7)**

- *Object Query Language (OQL)* is a query language for OODBMSs that provides constructs to query collection types. Its data definition language is called *Object Data Language (ODL)*. We discussed elements of the ODML data model and gave examples of OQL and ODL. **(Section 25.8)**

- An ORDBMS is a relational DBMS with the extensions discussed in this chapter. An OODBMS is a programming language with a type system that includes the features discussed in this chapter. ORDBMSs add OODBMS features to a RDBMS, but there are several differences to a full-fledged OODBMS. **(Section 25.9)**

EXERCISES

Exercise 25.1 Briefly answer the following questions.

1. What are the two kinds of new data types supported in object-database systems? Give an example of each, and discuss how the example situation would be handled if only an RDBMS was available.
2. What must a user do to define a new ADT?
3. Allowing users to define methods can lead to efficiency gains. Give an example.
4. What is late binding of methods? Give an example of inheritance that illustrates the need for dynamic binding.
5. What are collection hierarchies? Give an example that illustrates how collection hierarchies facilitate querying.
6. Discuss how a DBMS exploits encapsulation in implementing support for ADTs.
7. Give an example illustrating the nesting and unnesting operations.
8. Describe two objects that are deep equal but not shallow equal, or explain why this is not possible.

9. Describe two objects that are shallow equal but not deep equal, or explain why this is not possible.
10. Compare RDBMSs with ORDBMSs. Describe an application scenario for which you would choose an RDBMS, and explain why. Similarly, describe an application scenario for which you would choose an ORDBMS, and explain why.

Exercise 25.2 Consider the Dinky schema shown in Figure 25.1 and all related methods defined in the chapter. Write the following queries in extended SQL:

1. How many films were shown at theater *tno* = 5 between January 1 and February 1 of 1997?
2. What is the lowest budget for a film with at least two stars?
3. Consider theaters at which a film directed by Steven Spielberg started showing on January 1, 1997. For each such theater, print the names of all countries within a 100-mile radius. (You can use the *overlap* and *radius* methods illustrated in Figure 25.2.)

Exercise 25.3 In a company database, you need to store information about employees, departments, and children of employees. For each employee, identified by *ssn*, you must record *years* (the number of years that the employee has worked for the company), *phone*, and *photo* information. There are two subclasses of employees: contract and regular. Salary is computed by invoking a method that takes *years* as a parameter; this method has a different implementation for each subclass. Further, for each regular employee, you must record the name and age of every child. The most common queries involving children are similar to "Find the average age of Bob's children" and "Print the names of all of Bob's children."

A photo is a large image object and can be stored in one of several image formats (e.g., gif, jpeg). You want to define a *display* method for image objects; display must be defined differently for each image format. For each department, identified by *dno*, you must record *dname*, *budget*, and *workers* information. *Workers* is the set of employees who work in a given department. Typical queries involving workers include, "Find the average salary of all workers (across all departments)."

1. Using extended SQL, design an ORDBMS schema for the company database. Show all type definitions, including method definitions.
2. If you have to store this information in an RDBMS, what is the best possible design?
3. Compare the ORDBMS and RDBMS designs.
4. If you are told that a common request is to display the images of all employees in a given department, how would you use this information for physical database design?
5. If you are told that an employee's image must be displayed whenever any information about the employee is retrieved, would this affect your schema design?
6. If you are told that a common query is to find all employees who look similar to a given image, and given code that lets you create an index over all images to support retrieval of similar images, what would you do to utilize this code in an ORDBMS?

Exercise 25.4 ORDBMSs need to support efficient access over collection hierarchies. Consider the collection hierarchy of Theaters and Theater_cafes presented in the Dinky example. In your role as a DBMS implementor (not a DBA), you must evaluate three storage alternatives for these tuples:

- All tuples for all kinds of theaters are stored together on disk in an arbitrary order.
- All tuples for all kinds of theaters are stored together on disk, with the tuples that are from Theater_cafes stored directly after the last of the non-cafe tuples.
- Tuples from Theater_cafes are stored separately from the rest of the (non-cafe) theater tuples.

1. For each storage option, describe a mechanism for distinguishing plain theater tuples from Theater_cafe tuples.
2. For each storage option, describe how to handle the insertion of a new non-cafe tuple.
3. Which storage option is most efficient for queries over all theaters? Over just Theater_cafes? In terms of the number of I/Os, how much more efficient is the best technique for each type of query compared to the other two techniques?

Exercise 25.5 Different ORDBMSs use different techniques for building indexes to evaluate queries over collection hierarchies. For our Dinky example, to index theaters by name there are two common options:

- Build one B+ tree index over Theaters.*name* and another B+ tree index over Theater_cafes.*name*.
- Build one B+ tree index over the union of Theaters.*name* and Theater_cafes.*name*.

1. Describe how to efficiently evaluate the following query using each indexing option (this query is over all kinds of theater tuples):

   `SELECT * FROM` Theaters T `WHERE` T.name = 'Majestic'

   Give an estimate of the number of I/Os required in the two different scenarios, assuming there are 1,000,000 standard theaters and 1,000 theater-cafes. Which option is more efficient?
2. Perform the same analysis over the following query:

   `SELECT * FROM` Theater_cafes T `WHERE` T.name = 'Majestic'
3. For clustered indexes, does the choice of indexing technique interact with the choice of storage options? For unclustered indexes?

Exercise 25.6 Consider the following query:

```
SELECT thumbnail(I.image)
FROM   Images I
```

Given that the *I.image* column may contain duplicate values, describe how to use hashing to avoid computing the *thumbnail* function more than once per distinct value in processing this query.

Exercise 25.7 You are given a two-dimensional, $n \times n$ array of objects. Assume that you can fit 100 objects on a disk page. Describe a way to lay out (chunk) the array onto pages so that retrievals of square $m \times m$ subregions of the array are efficient. (Different queries will request subregions of different sizes, i.e., different m values, and your arrangement of the array onto pages should provide good performance, on average, for all such queries.)

Exercise 25.8 An ORDBMS optimizer is given a single-table query with n expensive selection conditions, $\sigma_n(...(\sigma_1(T)))$. For each condition σ_i, the optimizer can estimate the cost c_i of evaluating the condition on a tuple and the reduction factor of the condition r_i. Assume that there are t tuples in T.

1. How many tuples appear in the output of this query?
2. Assuming that the query is evaluated as shown (without reordering selections), what is the total cost of the query? Be sure to include the cost of scanning the table and applying the selections.
3. In Section 25.7.2 it was asserted that the optimizer should reorder selections so that they are applied to the table in order of increasing rank, where $\text{rank}_i = (r_i - 1)/c_i$. Prove that this assertion is optimal. That is, show that no other ordering could result in a query of lower cost. (Hint: It may be easiest to consider the special case where $n = 2$ first and generalize from there.)

BIBLIOGRAPHIC NOTES

A number of the object-oriented features described here are based in part on fairly old ideas in the programming languages community. [35] provides a good overview of these ideas in a database context. Stonebraker's book [630] describes the vision of ORDBMSs embodied by his company's early product, Illustra (now a product of Informix). Current commercial DBMSs with object-relational support include Informix Universal Server, IBM DB/2 CS V2, and UniSQL. An new version of Oracle is scheduled to include ORDBMS features as well.

Many of the ideas in current object-relational systems came out of a few prototypes built in the 1980s, especially POSTGRES [633], Starburst [296], and O2 [183].

The idea of an object-oriented database was first articulated in [165], which described the GemStone prototype system. Other prototypes include DASDBS [571], EXODUS [110], IRIS [235], ObjectStore [400], ODE, [14] ORION [370], SHORE [109], and THOR [417]. O2 is actually an early example of a system that was beginning to merge the themes of ORDBMSs and OODBMSs—it could fit in this list as well. [34] lists a collection of features that are generally considered to belong in an OODBMS. Current commercially available OODBMSs include GemStone, Itasca, O2, Objectivity, ObjectStore, Ontos, Poet, and Versant. [369] compares OODBMSs and RDBMSs.

Database support for ADTs was first explored in the INGRES and POSTGRES projects at U.C. Berkeley. The basic ideas are described in [627], including mechanisms for query processing and optimization with ADTs as well as extensible indexing. Support for ADTs was also investigated in the Darmstadt database system, [415]. Using the POSTGRES index extensibility correctly required intimate knowledge of DBMS-internal transaction mechanisms. Generalized search trees were proposed to solve this problem; they are described in [317], with concurrency and ARIES-based recovery details presented in [386]. [585] proposes that users must be allowed to define operators over ADT objects and properties of these operators that can be utilized for query optimization, rather than just a collection of methods.

Array chunking is described in [568]. Techniques for method caching and optimizing queries with expensive methods are presented in [315, 140]. Client-side data caching in a client-server OODBMS is studied in [242]. Clustering of objects on disk is studied in [650]. Work on nested relations was an early precursor of recent research on complex objects in OODBMSs and ORDBMSs. One of the first nested relation proposals is [439]. MVDs play an important role in reasoning about reduncancy in nested relations; see, for example, [504]. Storage structures for nested relations were studied in [181].

Formal models and query languages for object-oriented databases have been widely studied; papers include [4, 46, 62, 105, 331, 332, 366, 503, 635]. [365] proposes SQL extensions for querying object-oriented databases. An early and elegant extension of SQL with path expressions and inheritance was developed in GEM [692]. There has been much interest in combining deductive and object-oriented features. Papers in this area include [37, 247, 430, 485, 617, 694]. See [3] for a thorough textbook discussion of formal aspects of object-orientation and query languages.

[371, 373, 634, 696] include papers on DBMSs that would now be termed object-relational and/or object-oriented. [695] contains a detailed overview of schema and database evolution in object-oriented database systems. Drafts of the SQL3 standard are available electronically at URL `ftp://jerry.ece.umassd.edu/isowg3/`. The SQL:1999 standard is discussed in [200]. The incorporation of several SQL:1999 features into IBM DB2 is described in [108]. OQL is described in [120]. It is based to a large extent on the O2 query language, which is described, together with other aspects of O2, in the collection of papers [45].