PART 3

Object-based Databases and XML

Several application areas for database systems are limited by the restrictions of the relational data model. As a result, researchers have developed several data models to deal with these application domains. In this part, we study the object-oriented data model and the object-relational data model. In addition, we study XML, a language that can represent data that is less structured than that of the other data models.

The object-oriented data model, described in Chapter 8, is based on the object-oriented-programming language paradigm, which is now in wide use. Inheritance, object-identity, and encapsulation (information hiding), with methods to provide an interface to objects, are among the key concepts of object-oriented programming that have found applications in data modeling. The object-oriented data model also supports a rich type system, including structured and collection types. While inheritance and, to some extent, complex types are also present in the E-R model, encapsulation and object-identity distinguish the object-oriented data model from the E-R model.

The object-relational model, described in Chapter 9, combines features of the relational and object-oriented models. This model provides the rich type system of object-oriented databases, combined with relations as the basis for storage of data. It applies inheritance to relations, not just to types. The object-relational data model provides a smooth migration path from relational databases, which is attractive to relational database vendors. As a result, the SQL:1999 standard includes a number of object-oriented features in its type system, while continuing to use the relational model as the underlying model.

The XML language was initially designed as a way of adding markup information to text documents, but has become important because of its applications in data exchange. XML provides a way to represent data that have nested structure, and furthermore allows a great deal of flexibility in structuring of data, which is important for certain kinds of nontraditional data. Chapter 10 describes the XML language, and then presents different ways of expressing queries on data represented in XML, and transforming XML data from one form to another.

PART 8

Case Studies

This part describes how different database systems integrate the various concepts described earlier in the book. Specifically, three widely used database systems—IBM DB2, Oracle, and Microsoft SQL Server—are covered in Chapters 25, 26, and 27. These three represent three of the most widely used database systems.

Each of these chapters highlights unique features of each database system: tools, SQL variations and extensions, and system architecture, including storage organization, query processing, concurrency control and recovery, and replication.

The chapters cover only key aspects of the database products they describe, and therefore should not be regarded as a comprehensive coverage of the product. Furthermore, since products are enhanced regularly, details of the product may change. When using a particular product version, be sure to consult the user manuals for specific details.

Keep in mind that the chapters in this part use industrial rather than academic terminology. For instance, they use table instead of relation, row instead of tuple, and column instead of attribute.

CHAPTER 25

Oracle

Hakan Jakobsson
Oracle Corporation

When Oracle was founded in 1977 as Software Development Laboratories by Larry Ellison, Bob Miner, and Ed Oates, there were no commercial relational database products. The company, which was later renamed Oracle, set out to build a relational database management system as a commercial product, and was the first to reach the market. Since then, Oracle has held a leading position in the relational database market, but over the years its product and service offerings have grown beyond the relational database server. In addition to tools directly related to database development and management, Oracle sells business intelligence tools, including a multidimensional database management system (Oracle Express), query and analysis tools, data-mining products, and an application server with close integration to the database server.

In addition to database-related servers and tools, the company also offers application software for enterprise resource planning and customer-relationship management, including areas such as financials, human resources, manufacturing, marketing, sales, and supply chain management. Oracle's Business OnLine unit offers services in these areas as an application service provider.

This chapter surveys a subset of the features, options, and functionality of Oracle products. New versions of the products are being developed continually, so all product descriptions are subject to change. The feature set described here is based on the first release of Oracle9*i*.

25.1 Database Design and Querying Tools

Oracle provides a variety of tools for database design, querying, report generation and data analysis, including OLAP.

25.1.1 Database Design Tools

Most of Oracle's design tools are included in the Oracle Internet Development Suite. This is a suite of tools for various aspects of application development, including tools for forms development, data modeling, reporting, and querying. The suite supports the UML standard (see Section 2.10) for development modeling. It provides class modeling to generate code for the business components for Java framework as well as activity modeling for general-purpose control flow modeling. The suite also supports XML for data exchange with other UML tools.

The major database design tool in the suite is Oracle Designer, which translates business logic and data flows into a schema definitions and procedural scripts for application logic. It supports such modeling techniques as E-R diagrams, information engineering, and object analysis and design. Oracle Designer stores the design in Oracle Repository, which serves as a single point of metadata for the application. The metadata can then be used to generate forms and reports. Oracle Repository provides configuration management for database objects, forms applications, Java classes, XML files, and other types of files.

The suite also contains application development tools for generating forms, reports, and tools for various aspects of Java and XML-based development. The business intelligence component provides JavaBeans for analytic functionality such as data visualization, querying, and analytic calculations.

Oracle also has an application development tool for data warehousing, Oracle Warehouse Builder. Warehouse Builder is a tool for design and deployment of all aspects of a data warehouse, including schema design, data mapping and transformations, data load processing, and metadata management. Oracle Warehouse Builder supports both 3NF and star schemas and can also import designs from Oracle Designer.

25.1.2 Querying Tools

Oracle provides tools for ad-hoc querying, report generation and data analysis, including OLAP.

Oracle Discoverer is a Web-based, ad hoc query, reporting, analysis and Web publishing tool for end users and data analysts. It allows users to drill up and down on result sets, pivot data, and store calculations as reports that can be published in a variety of formats such as spreadsheets or HTML. Discoverer has wizards to help end users visualize data as graphs. Oracle9i has supports a rich set of analytical functions, such as ranking and moving aggregation in SQL. Discoverer's ad hoc query interface can generate SQL that takes advantage of this functionality and can provide end users with rich analytical functionality. Since the processing takes place in the relational database management system, Discoverer does not require a complex client-side calculation engine and there is a version of Discoverer that is browser based.

Oracle Express Server is a multidimensional database server. It supports a wide variety of analytical queries as well as forecasting, modeling, and scenario manage-

ment. It can use the relational database management system as a back end for storage or use its own multidimensional storage of the data.

With the introduction of OLAP services in Oracle9*i*, Oracle is moving away from supporting a separate storage engine and moving most of the calculations into SQL. The result is a model where all the data reside in the relational database management system and where any remaining calculations that cannot be performed in SQL are done in a calculation engine running on the database server. The model also provides a Java OLAP application programmer interface.

There are many reasons for moving away from a separate multidimensional storage engine:

- A relational engine can scale to much larger data sets.
- A common security model can be used for the analytical applications and the data warehouse.
- Multidimensional modeling can be integrated with data warehouse modeling.
- The relational database management system has a larger set of features and functionality in many areas such as high availability, backup and recovery, and third-party tool support.
- There is no need to train database administrators for two database engines.

The main challenge with moving away from a separate multidimensional database engine is to provide the same performance. A multidimensional database management system that materializes all or large parts of a data cube can offer very fast response times for many calculations. Oracle has approached this problem in two ways.

- Oracle has added SQL support for a wide range of analytical functions, including cube, rollup, grouping sets, ranks, moving aggregation, lead and lag functions, histogram buckets, linear regression, and standard deviation, along with the ability to optimize the execution of such functions in the database engine.
- Oracle has extended materialized views to permit analytical functions, in particular grouping sets. The ability to materialize parts or all of the cube is key to the performance of a multidimensional database management system and materialized views give a relational database management system the ability to do the same thing.

25.2 SQL Variations and Extensions

Oracle9*i* supports all core SQL:1999 features fully or partially, with some minor exceptions such as distinct data types. In addition, Oracle supports a large number of other language constructs, some of which conform with SQL:1999, while others are Oracle-specific in syntax or functionality. For example, Oracle supports the OLAP

operations described in Section 22.2, including ranking, moving aggregation, cube, and rollup.

A few examples of Oracle SQL extensions are:

- **connect by**, which is a form of tree traversal that allows transitive closure-style calculations in a single SQL statement. It is an Oracle-specific syntax for a feature that Oracle has had since the 1980s.
- **Upsert** and **multitable inserts**. The upsert operation combines update and insert, and is useful for merging new data with old data in data warehousing applications. If a new row has the same key value as an old row, the old row is updated (for example by adding the measure values from the new row), otherwise the new row is inserted into the table. Multitable inserts allow multiple tables to be updated based on a single scan of new data.
- **with** clause, which is described in Section 4.8.2.

25.2.1 Object-Relational Features

Oracle has extensive support for object-relational constructs, including:

- **Object types**. A single-inheritance model is supported for type hierarchies.
- **Collection types**. Oracle supports **varrays** which are variable length arrays, and nested tables.
- **Object tables**. These are used to store objects while providing a relational view of the attributes of the objects.
- **Table functions**. These are functions that produce sets of rows as output, and can be used in the **from** clause of a query. Table functions in Oracle can be nested. If a table function is used to express some form of data transformation, nesting multiple functions allows multiple transformations to be expressed in a single statement.
- **Object views**. These provide a virtual object table view of data stored in a regular relational table. They allow data to be accessed or viewed in an object-oriented style even if the data are really stored in a traditional relational format.
- **Methods**. These can be written in PL/SQL, Java, or C.
- **User-defined aggregate functions**. These can be used in SQL statements in the same way as built-in functions such as **sum** and **count**.
- **XML data types.** These can be used to store and index XML documents.

Oracle has two main procedural languages, PL/SQL and Java. PL/SQL was Oracle's original language for stored procedures and it has syntax similar to that used in the Ada language. Java is supported through a Java virtual machine inside the database engine. Oracle provides a package to encapsulate related procedures, functions, and

variables into single units. Oracle supports SQLJ (SQL embedded in Java) and JDBC, and provides a tool to generate Java class definitions corresponding to user-defined database types.

25.2.2 Triggers

Oracle provides several types of triggers and several options for when and how they are invoked. (See Section 6.4 for an introduction to triggers in SQL.) Triggers can be written in PL/SQL or Java or as C callouts.

For triggers that execute on DML statements such as insert, update, and delete, Oracle supports **row triggers** and **statement triggers**. Row triggers execute once for every row that is affected (updated or deleted, for example) by the DML operation. A statement trigger is executed just once per statement. In each case, the trigger can be defined as either a *before* or *after* trigger, depending on whether it is to be invoked before or after the DML operation is carried out.

Oracle allows the creation of **instead of** triggers for views that cannot be subject to DML operations. Depending on the view definition, it may not be possible for Oracle to translate a DML statement on a view to modifications of the underlying base tables unambiguously. Hence, DML operations on views are subject to numerous restrictions. A user can create an **instead of** trigger on a view to specify manually what operations on the base tables are to occur in response to the DML operation on the view. Oracle executes the trigger instead of the DML operation and therefore provides a mechanism to circumvent the restrictions on DML operations against views.

Oracle also has triggers that execute on a variety of other events, like database startup or shutdown, server error messages, user logon or logoff, and DDL statements such as **create**, **alter** and **drop** statements.

25.3 Storage and Indexing

In Oracle parlance, a *database* consists of information stored in files and is accessed through an *instance*, which is a shared memory area and a set of processes that interact with the data in the files.

25.3.1 Table Spaces

A database consists of one or more logical storage units called **table spaces**. Each table space, in turn, consists of one or more physical structures called **data files**. These may be either files managed by the operating system or raw devices.

Usually, an Oracle database will have the following table spaces:

- The **system** table space, which is always created. It contains the data dictionary tables and storage for triggers and stored procedures.
- Table spaces created to store user data. While user data can be stored in the **system** table space, it is often desirable to separate the user data from the system data. Usually, the decision about what other table spaces should be created is based on performance, availability, maintainability, and ease of admin-

istration. For example, having multiple table spaces can be useful for partial backup and recovery operations.

- Temporary table spaces. Many database operations require sorting the data, and the sort routine may have to store data temporarily on disk if the sort cannot be done in memory. Temporary table spaces are allocated for sorting, to make the space management operations involved in spilling to disk more efficient.

Table spaces can also be used as a means of moving data between databases. For example, it is common to move data from a transactional system to a data warehouse at regular intervals. Oracle allows moving all the data in a table space from one system to the other by simply copying the files and exporting and importing a small amount of data dictionary metadata. These operations can be much faster than unloading the data from one database and then using a loader to insert it into the other. A requirement for this feature is that both systems use the same operating system.

25.3.2 Segments

The space in a table space is divided into units, called **segments**, that each contain data for a specific data structure. There are four types of segments.

- **Data segments**. Each table in a table space has its own data segment where the table data are stored unless the table is partitioned; if so, there is one data segment per partition. (Partitioning in Oracle is described in Section 25.3.10.)
- **Index segments**. Each index in a table space has its own index segment, except for partitioned indices, which have one index segment per partition.
- **Temporary segments**. These are segments used when a sort operation needs to write data to disk or when data are inserted into a temporary table.
- **Rollback segments**. These segments contain undo information so that an uncommitted transaction can be rolled back. They also play an important roll in Oracle's concurrency control model and for database recovery, described in Sections 25.5.1 and 25.5.2.

Below the level of segment, space is allocated at a level of granularity called *extent*. Each extent consists of a set of contiguous database *blocks*. A database block is the lowest level of granularity at which Oracle performs disk I/O. A database block does not have to be the same as an operating system block in size, but should be a multiple thereof.

Oracle provides storage parameters that allow for detailed control of how space is allocated and managed, parameters such as:

- The size of a new extent that is to be allocated to provide room for rows that are inserted into a table.

- The percentage of space utilization at which a database block is considered full and at which no more rows will be inserted into that block. (Leaving some free space in a block can allow the existing rows to grow in size through updates, without running out of space in the block.)

25.3.3 Tables

A standard table in Oracle is heap organized; that is, the storage location of a row in a table is not based on the values contained in the row, and is fixed when the row is inserted. However, if the table is partitioned, the content of the row affects the partition in which it is stored. There are several features and variations.

Oracle supports nested tables; that is, a table can have a column whose data type is another table. The nested table is not stored in line in the parent table, but is stored in a separate table.

Oracle supports temporary tables where the duration of the data is either the transaction in which the data are inserted, or the user session. The data are private to the session and are automatically removed at the end of its duration.

A *cluster* is another form of organization for table data (see Section 11.7). The concept, in this context, should not be confused with other meanings of the word *cluster*, such as those relating to hardware architecture. In a cluster, rows from different tables are stored together in the same block on the basis of some common columns. For example, a department table and an employee table could be clustered so that each row in the department table is stored together with all the employee rows for those employees who work in that department. The primary key/foreign key values are used to determine the storage location. This organization gives performance benefits when the two tables are joined, but without the space penalty of a denormalized schema, since the values in the department table are not repeated for each employee. As a tradeoff, a query involving only the department table may have to involve a substantially larger number of blocks than if that table had been stored on its own.

The cluster organization implies that a row belongs in a specific place; for example, a new employee row must be inserted with the other rows for the same department. Therefore, an index on the clustering column is mandatory. An alternative organization is a *hash cluster*. Here, Oracle computes the location of a row by applying a hash function to the value for the cluster column. The hash function maps the row to a specific block in the hash cluster. Since no index traversal is needed to access a row according to its cluster column value, this organization can save significant amounts of disk I/O. However, the number of hash buckets and other storage parameters must be set carefully to avoid performance problems due to too many collisions or space wastage due to empty hash buckets.

Both the hash cluster and regular cluster organization can be applied to a single table. Storing a table as a hash cluster with the primary key column as the cluster key can allow an access based on a primary key value with a single disk I/O provided that there is no overflow for that data block.

25.3.4 Index-Organized Tables

In an *index organized* table, records are stored in an Oracle B-tree index instead of in a heap. An index-organized table requires that a unique key be identified for use as the index key. While an entry in a regular index contains the key value and row-id of the indexed row, an index-organized table replaces the row-id with the column values for the remaining columns of the row. Compared to storing the data in a regular heap table and creating an index on the key columns, index-organized table can improve both performance and space utilization. Consider looking up all the column values of a row, given its primary key value. For a heap table, that would require an index probe followed by a table access by row-id. For an index-organized table, only the index probe is necessary.

Secondary indices on nonkey columns of an index-organized table are different from indices on a regular heap table. In a heap table, each row has a fixed row-id that does not change. However, a B-tree is reorganized as it grows or shrinks when entries are inserted or deleted, and there is no guarantee that a row will stay in a fixed place inside an index-organized table. Hence, a secondary index on an index-organized table contains not normal row-ids, but **logical row-ids** instead. A logical row-id consists of two parts: a physical row-id corresponding to where the row was when the index was created or last rebuilt and a value for the unique key. The physical row-id is referred to as a "guess" since it could be incorrect if the row has been moved. If so, the other part of a logical row-id, the key value for the row, is used to access the row; however, this access is slower than if the guess had been correct, since it involves a traversal of the B-tree for the index-organized table from the root all the way to the leaf nodes, potentially incurring several disk I/Os. However, if a table is highly volatile and a large percentage of the guesses are likely to be wrong, it can be better to create the secondary index with only key values, since using an incorrect guess may result in a wasted disk I/O.

25.3.5 Indices

Oracle supports several different types of indices. The most commonly used type is a B-tree index, created on one or multiple columns. (Note: in the terminology of Oracle (as also in several other database systems) a B-tree index is what is referred to as a B^+-tree index in Chapter 12.) Index entries have the following format: For an index on columns col_1, col_2, and col_3, each row in the table where at least one of the columns has a nonnull value would result in the index entry

$$< col_1 >< col_2 >< col_3 >< row\text{-}id >$$

where $< col_i >$ denotes the value for column i and $< row\text{-}id >$ is the row-id for the row. Oracle can optionally compress the prefix of the entry to save space. For example, if there are many repeated combinations of $< col_1 >< col_2 >$ values, the representation of each distinct $< col_1 >< col_2 >$ prefix can be shared between the entries that have that combination of values, rather than stored explicitly for each such entry. Prefix compression can lead to substantial space savings.

25.3.6 Bitmap Indices

Bitmap indices (described in Section 12.9.4) use a bitmap representation for index entries, which can lead to substantial space saving (and therefore disk I/O savings), when the indexed column has a moderate number of distinct values. Bitmap indices in Oracle use the same kind of B-tree structure to store the entries as a regular index. However, where a regular index on a column would have entries of the form $< col_1 >< row\text{-}id >$, a bitmap index entry has the form

$$< col_1 >< startrow\text{-}id >< endrow\text{-}id >< compressedbitmap >$$

The bitmap conceptually represents the space of all possible rows in the table between the start and end row-id. The number of such possible rows in a block depends on how many rows can fit into a block, which is a function of the number of columns in the table and their data types. Each bit in the bitmap represents one such possible row in a block. If the column value of that row is that of the index entry, the bit is set to 1. If the row has some other value, or the row does not actually exist in the table, the bit is set to 0. (It is possible that the row does not actually exist because a table block may well have a smaller number of rows than the number that was calculated as the maximum possible.) If the difference is large, the result may be long strings of consecutive zeros in the bitmap, but the compression algorithm deals with such strings of zeros, so the negative effect is limited.

The compression algorithm is a variation of a compression technique called Byte-Aligned Bitmap Compression (BBC). Essentially, a section of the bitmap where the distance between two consecutive ones is small enough is stored as verbatim bitmaps. If the distance between two ones is sufficiently large—that is, there is a sufficient number of adjacent zeros between them—a runlength of zeros, that is the number of zeros, is stored.

Bitmap indices allow multiple indices on the same table to be combined in the same access path if there are multiple conditions on indexed columns in the **where** clause of a query. For example, for the condition

$$(col_1 = 1 \textbf{ or } col_1 = 2) \textbf{ and } col_2 > 5 \textbf{ and } col_3 <> 10$$

Oracle would be able to calculate which rows match the condition by performing Boolean operations on bitmaps from indices on the three columns. In this case, these operations would take place for each index:

- For the index on col_1, the bitmaps for key values 1 and 2 would be **or**ed.
- For the index on col_2, all the bitmaps for key values > 5 would be merged in an operation that corresponds to a logical **or**.
- For the index on col_3, the bitmaps for key values 10 and **null** would be retrieved. Then, a Boolean **and** would be performed on the results from the first two indices, followed by two Boolean minuses of the bitmaps for values 10 and **null** for col_3.

All operations are performed directly on the compressed representation of the bitmaps—no decompression is necessary—and the resulting (compressed) bitmap represents those rows that match all the logical conditions.

The ability to use the Boolean operations to combine multiple indices is not limited to bitmap indices. Oracle can convert row-ids to the compressed bitmap representation, so it can use a regular B-tree index anywhere in a Boolean tree of bitmap operation simply by putting a row-id-to-bitmap operator on top of the index access in the execution plan.

As a rule of thumb, bitmap indices tend to be more space efficient than regular B-tree indices if the number of distinct key values is less than half the number of rows in the table. For example, in a table with 1 million rows, an index on a column with less than 500,000 distinct values would probably be smaller if it were created as a bitmap index. For columns with a very small number of distinct values—for example, columns referring to properties such as country, state, gender, marital status, and various status flags—a bitmap index might require only a small fraction of the space of a regular B-tree index. Any such space advantage can also give rise to corresponding performance advantages in the form of fewer disk I/Os when the index is scanned.

25.3.7 Function-Based Indices

In addition to creating indices on one or multiple columns of a table, Oracle allows indices to be created on expressions that involve one or more columns, such as $col_1 + col_2 * 5$. For example, by creating an index on the expression *upper*(*name*), where *upper* is a function that returns the uppercase version of a string, and *name* is a column, it is possible to do case-insensitive searches on the *name* column. In order to find all rows with name "van Gogh" efficiently, the condition

$$upper(name) = \text{'VAN GOGH'}$$

would be used in the **where** clause of the query. Oracle then matches the condition with the index definition and concludes that the index can be used to retrieve all the rows matching "van Gogh" regardless of how the name was capitalized when it was stored in the database. A function-based index can be created as either a bitmap or a B-tree index.

25.3.8 Join Indices

A join index is an index where the key columns are not in the table that is referenced by the row-ids in the index. Oracle supports bitmap join indices primarily for use with star schemas (see Section 22.4.2). For example, if there is a column for product names in a product dimension table, a bitmap join index on the fact table with this key column could be used to retrieve the fact table rows that correspond to a product with a specific name, although the name is not stored in the fact table. How the rows in the fact and dimension tables correspond is based on a join condition that is specified when the index is created, and becomes part of the index metadata. When a query is

processed, the optimizer will look for the same join condition in the **where** clause of the query in order to determine if the join index is applicable.

Oracle allows bitmap join indices to have more than one key column and these columns can be in different tables. In all cases, the join conditions between the fact table on which the index is built and the dimension tables must refer to unique keys in the dimension tables; that is, an indexed row in the fact table must correspond to a unique row in each of the dimension tables.

Oracle can combine a bitmap join index on a fact table with other indices on the same table—whether join indices or not—by using the operators for Boolean bitmap operations. For example, consider a schema with a fact table for sales, and dimension tables for customers, products, and time. Suppose a query requests information about sales to customers in a certain zip code who bought products in a certain product category during a certain time period. If a multicolumn bitmap join index exists where the key columns are the constrained dimension table columns (zip code, product category and time), Oracle can use the join index to find rows in the fact table that match the constraining conditions. However, if individual, single-column indices exist for the key columns (or a subset of them), Oracle can retrieve bitmaps for fact table rows that match each individual condition, and use the Boolean and operation to generate a fact table bitmap for those rows that satisfy all the conditions. If the query contains conditions on some columns of the fact table, indices on those columns could be included in the same access path, even if they were regular B-tree indices or domain indices (domain indices are described below in Section 25.3.9).

25.3.9 Domain Indices

Oracle allows tables to be indexed by index structures that are not native to Oracle. This extensibility feature of the Oracle server allows software vendors to develop so-called **cartridges** with functionality for specific application domains, such as text, spatial data, and images, with indexing functionality beyond that provided by the standard Oracle index types. In implementing the logic for creating, maintaining, and searching the index, the index designer must ensure that it adheres to a specific protocol in its interaction with the Oracle server.

A domain index must be registered in the data dictionary, together with the operators it supports. Oracle's optimizer considers domain indices as one of the possible access paths for a table. Oracle allows cost functions to be registered with the operators so that the optimizer can compare the cost of using the domain index to those of other access paths.

For example, a domain index for advanced text searches may support an operator *contains*. Once this operator has been registered, the domain index will be considered as an access path for a query like

```
select *
from employees
where contains(resume, 'LINUX')
```

where *resume* is a text column in the *employee* table. The domain index can be stored in either an external data file or inside an Oracle index-organized table.

A domain index can be combined with other (bitmap or B-tree) indices in the same access path by converting between the row-id and bitmap representation and using Boolean bitmap operations.

25.3.10 Partitioning

Oracle supports various kinds of horizontal partitioning of tables and indices, and this feature plays a major role in Oracle's ability to support very large databases. The ability to partition a table or index has advantages in many areas.

- Backup and recovery are easier and faster, since they can be done on individual partitions rather than on the table as a whole.
- Loading operations in a data warehousing environment are less intrusive: data can be added to a partition, and then the partition added to a table, which is an instantaneous operation. Likewise, dropping a partition with obsolete data from a table is very easy in a data warehouse that maintains a rolling window of historical data.
- Query performance benefits substantially, since the optimizer can recognize that only a subset of the partitions of a table need to be accessed in order to resolve a query (partition pruning). Also, the optimizer can recognize that in a join, it is not necessary to try to match all rows in one table with all rows in the other, but that the joins need to be done only between matching pairs of partitions (partitionwise join).

Each row in a partitioned table is associated with a specific partition. This association is based on the partitioning column or columns that are part of the definition of a partitioned table. There are several ways to map column values to partitions, giving rise to several types of partitioning, each with different characteristics: range, hash, composite, and list partitioning.

25.3.10.1 Range Partitioning

In range partitioning, the partitioning criteria are ranges of values. This type of partitioning is especially well suited to date columns, in which case all rows in the same date range, say a day or a month, belong in the same partition. In a data warehouse where data are loaded from the transactional systems at regular intervals, range partitioning can be used to implement a rolling window of historical data efficiently. Each data load gets its own new partition, making the loading process faster and more efficient. The system actually loads the data into a separate table with the same column definition as the partitioned table. It can then check the data for consistency, cleanse them, and index them. After that, the system can make the separate table a new partition of the partitioned table, by a simple change to the metadata in the data dictionary—a nearly instantaneous operation.

Up until the metadata change, the loading process does not affect the existing data in the partitioned table in any way. There is no need to do any maintenance of existing indices as part of the loading. Old data can be removed from a table by simply dropping its partition; this operation does not affect the other partitions.

In addition, queries in a data warehousing environment often contain conditions that restrict them to a certain time period, such as a quarter or month. If date range partitioning is used, the query optimizer can restrict the data access to those partitions that are relevant to the query, and avoid a scan of the entire table.

25.3.10.2 Hash Partitioning

In hash partitioning, a hash function maps rows to partitions according to the values in the partitioning columns. This type of partitioning is primarily useful when it is important to distribute the rows evenly among partitions or when partitionwise joins are important for query performance.

25.3.10.3 Composite Partitioning

In composite partitioning, the table is range partitioned, but each partition is subpartitioned by using hash partitioning. This type of partitioning combines the advantages of range partitioning and hash partitioning.

25.3.10.4 List Partitioning

In list partitioning, the values associated with a particular partition are stated in a list. This type of partitioning is useful if the data in the partitioning column have a relatively small set of discrete values. For instance, a table with a state column can be implicitly partitioned by geographical region if each partition list has the states that belong in the same region.

25.3.11 Materialized Views

The materialized view feature (see Section 3.5.1) allows the result of an SQL query to be stored in a table and used for later query processing. In addition, Oracle maintains the materialized result, updating it when the tables that were referenced in the query are updated. Materialized views are used in data warehousing to speed up query processing, but the technology is also used for replication in distributed and mobile environments.

In data warehousing, a common usage for materialized views is to summarize data. For example, a common type of query asks for "the sum of sales for each quarter during the last 2 years." Precomputing the result, or some partial result, of such a query can speed up query processing dramatically compared to computing it from scratch by aggregating all detail-level sales records.

Oracle supports automatic query rewrites that take advantage of any useful materialized view when resolving a query. The rewrite consists of changing the query to use the materialized view instead of the original tables in the query. In addition, the rewrite may add additional joins or aggregate processing as may be required to get

the correct result. For example, if a query needs sales by quarter, the rewrite can take advantage of a view that materializes sales by month, by adding additional aggregation to roll up the months to quarters. Oracle has a type of metadata object called dimension that allows hierarchical relationships in tables to be defined. For example, for a time dimension table in a star schema, Oracle can define a dimension metadata object to specify how days roll up to months, months to quarters, quarters to years, and so forth. Likewise, hierarchical properties relating to geography can be specified —for example, how sales districts roll up to regions. The query rewrite logic looks at these relationships since they allow a materialized view to be used for wider classes of queries.

The container object for a materialized view is a table, which means that a materialized view can be indexed, partitioned, or subjected to other controls, to improve query performance.

When there are changes to the data in the tables referenced in the query that defines a materialized view, the materialized view must be refreshed to reflect those changes. Oracle supports both full refresh of a materialized view and fast, incremental refresh. In a full refresh, Oracle recomputes the materialized view from scratch, which may be the best option if the underlying tables have had significant changes, for example, changes due to a bulk load. In an incremental refresh, Oracle updates the view using the records that were changed in the underlying tables; the refresh to the view is immediate, that is, it is executed as part of the transaction that changed the underlying tables. Incremental refresh may be better if the number of rows that were changed is low. There are some restrictions on the classes of queries for which a materialized view can be incrementally refreshed (and others for when a materialized view can be created at all).

A materialized view is similar to an index in the sense that, while it can improve query performance, it uses up space, and creating and maintaining it consumes resources. To help resolve this tradeoff, Oracle provides a package that can advise a user of the most cost-effective materialized views, given a particular query workload as input.

25.4 Query Processing and Optimization

Oracle supports a large variety of processing techniques in its query processing engine. Some of the more important ones are described here briefly.

25.4.1 Execution Methods

Data can be accessed through a variety of access methods:

- **Full table scan.** The query processor scans the entire table by getting information about the blocks that make up the table from the extent map, and scanning those blocks.
- **Index scan.** The processor creates a start and/or stop key from conditions in the query and uses it to scan to a relevant part of the index. If there are columns that need to be retrieved, that are not part of the index, the index

scan would be followed by a table access by index row-id. If no start or stop key is available, the scan would be a full index scan.

- **Index fast full scan.** The processor scans the extents the same way as the table extent in a full table scan. If the index contains all the columns that are needed in the index, and there are no good start/stop keys that would significantly reduce that portion of the index that would be scanned in a regular index scan, this method may be the fastest way to access the data. This is because the fast full scan can take full advantage of multiblock disk I/O. However, unlike a regular full scan, which traverses the index leaf blocks in order, a fast full scan does not guarantee that the output preserves the sort order of the index.
- **Index join.** If a query needs only a small subset of the columns of a wide table, but no single index contains all those columns, the processor can use an index join to generate the relevant information without accessing the table, by joining several indices that together contain the needed columns. It performs the joins as hash joins on the row-ids from the different indices.
- **Cluster and hash cluster access.** The processor accesses the data by using the cluster key.

Oracle has several ways to combine information from multiple indices in a single access path. This ability allows multiple **where**-clause conditions to be used together to compute the result set as efficiently as possible. The functionality includes the ability to perform Boolean operations **and**, **or**, and **minus** on bitmaps representing row-ids. There are also operators that map a list of row-ids into bitmaps and vice versa, which allows regular B-tree indices and bitmap indices to be used together in the same access path. In addition, for many queries involving **count**(*) on selections on a table, the result can be computed by just counting the bits that are set in the bitmap generated by applying the **where** clause conditions, without accessing the table.

Oracle supports several types of joins in the execution engine: inner joins, outer joins, semijoins, and antijoins. (An antijoin in Oracle returns rows from the left-hand side input that do not match any row in the right-hand side input; this operation is called anti-semijoin in other literature.) It evaluates each type of join by one of three methods: hash join, sort–merge join, or nested-loop join.

25.4.2 Optimization

In Chapter 14, we discussed the general topic of query optimization. Here, we discuss optimization in the context of Oracle.

25.4.2.1 Query Transformations

Oracle does query optimization in several stages. Most of the techniques relating to query transformations and rewrites take place before access path selection, but Oracle also supports several types of cost-based query transformations that generate a complete plan and return a cost estimate for both a standard version of the query and

one that has been subjected to advanced transformations. Not all query transformation techniques are guaranteed to be beneficial for every query, but by generating a cost estimate for the best plan with and without the transformation applied, Oracle is able to make an intelligent decision.

Some of the major types of transformations and rewrites supported by Oracle are as follows:

- **View merging.** A view reference in a query is replaced by the view definition. This transformation is not applicable to all views.
- **Complex view merging.** Oracle offers this feature for certain classes of views that are not subject to regular view merging because they have a **group by** or **select distinct** in the view definition. If such a view is joined to other tables, Oracle can commute the joins and the sort operation used for the **group by** or **distinct**.
- **Subquery flattening.** Oracle has a variety of transformations that convert various classes of subqueries into joins, semijoins, or antijoins.
- **Materialized view rewrite.** Oracle has the ability to rewrite a query automatically to take advantage of materialized views. If some part of the query can be matched up with an existing materialized view, Oracle can replace that part of the query with a reference to the table in which the view is materialized. If need be, Oracle adds join conditions or **group by** operations to preserve the semantics of the query. If multiple materialized views are applicable, Oracle picks the one that gives the greatest advantage in reducing the amount of data that has to be processed. In addition, Oracle subjects both the rewritten query and the original version to the full optimization process producing an execution plan and an associated cost estimate for each. Oracle then decides whether to execute the rewritten or the original version of the query on the basis of the cost estimates.
- **Star transformation.** Oracle supports a technique for evaluating queries against star schemas, known as the star transformation. When a query contains a join of a fact table with dimension tables, and selections on attributes from the dimension tables, the query is transformed by deleting the join condition between the fact table and the dimension tables, and replacing the selection condition on each dimension table by a subquery of the form:

  fact_table.fk$_i$ **in**
  (**select** *pk* **from** *dimension_table*$_i$
  where <conditions on *dimension_table*$_i$ >)

  One such subquery is generated for each dimension that has some constraining predicate. If the dimension has a snow-flake schema (see Section 22.4), the subquery will contain a join of the applicable tables that make up the dimension.

Oracle uses the values that are returned from each subquery to probe an index on the corresponding fact table column, getting a bitmap as a result. The bitmaps generated from different subqueries are combined by a bitmap **and** operation. The resultant bitmap can be used to access matching fact table rows. Hence, only those rows in the fact table that simultaneously match the conditions on the constrained dimensions will be accessed.

Both the decision on whether the use of a subquery for a particular dimension is cost-effective, and the decision on whether the rewritten query is better than the original, are based on the optimizer's cost estimates.

25.4.2.2 Access Path Selection

Oracle has a cost-based optimizer that determines join order, join methods, and access paths. Each operation that the optimizer considers has an associated cost function, and the optimizer tries to generate the combination of operations that has the lowest overall cost.

In estimating the cost of an operation, the optimizer relies on statistics that have been computed for schema objects such as tables and indices. The statistics contain information about the size of the object, the cardinality, data distribution of table columns, and so forth. For column statistics, Oracle supports height-balanced and frequency histograms. To facilitate the collection of optimizer statistics, Oracle can monitor modification activity on tables and keep track of those tables that have been subject to enough changes that recalculating the statistics may be appropriate. Oracle also tracks what columns are used in **where** clauses of queries, which make them potential candidates for histogram creation. With a single command, a user can tell Oracle to refresh the statistics for those tables that were marked as sufficiently changed. Oracle uses sampling to speed up the process of gathering the new statistics and automatically chooses the smallest adequate sample percentage. It also determines whether the distribution of the marked columns merit the creation of histograms; if the distribution is close to uniform, Oracle uses a simpler representation of the column statistics.

Oracle uses both CPU cost and disk I/Os in the optimizer cost model. To balance the two components, it stores measures about CPU speed and disk I/O performance as part of the optimizer statistics. Oracle's package for gathering optimizer statistics computes these measures.

For queries involving a nontrivial number of joins, the search space is an issue for a query optimizer. Oracle addresses this issue in several ways. The optimizer generates an initial join order and then decides on the best join methods and access paths for that join order. It then changes the order of the tables and determines the best join methods and access paths for the new join order and so forth, while keeping the best plan that has been found so far. Oracle cuts the optimization short if the number of different join orders that have been considered becomes so large that the time spent in the optimizer may be noticeable compared to the time it would take to execute the best plan found so far. Since this cutoff depends on the cost estimate for the best plan found so far, finding a good plan early is important so that the optimization can be stopped after a smaller number of join orders, resulting in better response time.

Oracle uses several initial ordering heuristics to increase the likelihood that the first join order considered is a good one.

For each join order that is considered, the optimizer may make additional passes over the tables to decide join methods and access paths. Such additional passes would target specific global side effects of the access path selection. For instance, a specific combination of join methods and access paths may eliminate the need to perform an **order by** sort. Since such a global side effect may not be obvious when the costs of the different join methods and access paths are considered locally, a separate pass targeting a specific side effect is used to find a possible execution plan with a better overall cost.

25.4.2.3 Partition Pruning

For partitioned tables, the optimizer tries to match conditions in the **where** clause of a query with the partitioning criteria for the table, in order to avoid accessing partitions that are not needed for the result. For example, if a table is partitioned by date range and the query is constrained to data between two specific dates, the optimizer determines which partitions contain data between the specified dates and ensures that only those partitions are accessed. This scenario is very common, and the speedup can be dramatic if only a small subset of the partitions are needed.

25.4.3 Parallel Execution

Oracle allows the execution of a single SQL statement to be parallelized by dividing the work between multiple processes on a multiprocessor computer. This feature is especially useful for computationally intensive operations that would otherwise take an unacceptably long time to perform. Representative examples are decision support queries that need to process large amounts of data, data loads in a data warehouse, and index creation or rebuild.

In order to achieve good speedup through parallelism, it is important that the work involved in executing the statement be divided into granules that can be processed independently by the different parallel processors. Depending on the type of operation, Oracle has several ways to split up the work.

For operations that access base objects (tables and indices), Oracle can divide the work by horizontal slices of the data. For some operations, such as a full table scan, each such slice can be a range of blocks—each parallel query process scans the table from the block at the start of the range to the block at the end. For other operations on a partitioned table, like update and delete, the slice would be a partition. For inserts into a nonpartitioned table, the data to be inserted are randomly divided across the parallel processes.

Joins can be parallelized in several different ways. One way is to divide one of the inputs to the join between parallel processes and let each process join its slice with the other input to the join; this is the asymmetric fragment-and-replicate method of Section 20.5.2.2. For example, if a large table is joined to a small one by a hash join, Oracle divides the large table among the processes and broadcasts a copy of the small table to each process, which then joins its slice with the smaller table. If both

tables are large, it would be prohibitively expensive to broadcast one of them to all processes. In that case, Oracle achieves parallelism by partitioning the data among processes by hashing on the values of the join columns (the partitioned hash-join method of Section 20.5.2.1). Each table is scanned in parallel by a set of processes and each row in the output is passed on to one of a set of processes that are to perform the join. Which one of these processes gets the row is determined by a hash function on the values of the join column. Hence, each join process gets only rows that could potentially match, and no rows that could match could end up in different processes.

Oracle parallelizes sort operations by value ranges of the column on which the sort is performed (that is, using the range-partitioning sort of Section 20.5.1). Each process participating in the sort is sent rows with values in its range, and it sorts the rows in its range. To maximize the benefits of parallelism, the rows need to be divided as evenly as possible among the parallel processes, and the problem of determining range boundaries that generates a good distribution then arises. Oracle solves the problem by dynamically sampling a subset of the rows in the input to the sort before deciding on the range boundaries.

25.4.3.1 Process Structure

The processes involved in the parallel execution of an SQL statement consist of a coordinator process and a number of parallel server processes. The coordinator is responsible for assigning work to the parallel servers and for collecting and returning data to the user process that issued the statement. The degree of parallelism is the number of parallel server processes that are assigned to execute a primitive operation as part of the statement. The degree of parallelism is determined by the optimizer, but can be throttled back dynamically if the load on the system increases.

The parallel servers operate on a producer/consumer model. When a sequence of operations is needed to process a statement, the producer set of servers performs the first operation and passes the resulting data to the consumer set. For example, if a full table scan is followed by a sort and the degree of parallelism is 12, there would be 12 producer servers performing the table scan and passing the result to 12 consumer servers that perform the sort. If a subsequent operation is needed, like another sort, the roles of the two sets of servers switch. The servers that originally performed the table scan take on the role of consumers of the output produced by the the first sort and use it to perform the second sort. Hence, a sequence of operations proceeds by passing data back and forth between two sets of servers that alternate in their roles as producers and consumers. The servers communicate with each other through memory buffers on shared-memory hardware and through high-speed network connections on MPP (shared nothing) configurations and clustered (shared disk) systems.

For shared nothing systems, the cost of accessing data on disk is not uniform among processes. A process running on a node that has direct access to a device is able to process data on that device faster than a process that has to retrieve the data over a network. Oracle uses knowledge about device-to-node and device-to-process affinity—that is, the ability to access devices directly—when distributing work among parallel execution servers.

25.5 Concurrency Control and Recovery

Oracle supports concurrency control and recovery techniques that provide a number of useful features.

25.5.1 Concurrency Control

Oracle's multiversion concurrency control differs from the concurrency mechanisms used by most other database vendors. Read-only queries are given a read-consistent snapshot, which is a view of the database as it existed at a specific point in time, containing all updates that were committed by that point in time, and not containing any updates that were not committed at that point in time. Thus, read locks are not used and read-only queries do not interfere with other database activity in terms of locking. (This is basically the multiversion two-phase locking protocol described in Section 16.5.2.)

Oracle supports both statement and transaction level read consistency: At the beginning of the execution of either a statement or a transaction (depending on what level of consistency is used), Oracle determines the current system change number (SCN). The SCN essentially acts as a timestamp, where the time is measured in terms of transaction commits instead of wall-clock time.

If in the course of a query a data block is found that has a higher SCN than the one being associated with the query, it is evident that the data block has been modified after the time of the original query's SCN by some other transaction that may or may not have committed. Hence, the data in the block cannot be included in a consistent view of the database as it existed at the time of the query's SCN. Instead, an older version of the data in the block must be used; specifically, the one that has the highest SCN that does not exceed the SCN of the query. Oracle retrieves that version of the data from the rollback segment (rollback segments are described in Section 25.5.2). Hence, provided that the rollback segment is sufficiently large, Oracle can return a consistent result of the query even if the data items have been modified several times since the query started execution. Should the block with the desired SCN no longer exist in the rollback segment, the query will return an error. It would be an indication that the rollback segment has not been properly sized, given the activity on the system.

In the Oracle concurrency model, read operations do not block write operations and write operations do not block read operations, a property that allows a high degree of concurrency. In particular, the scheme allows for long-running queries (for example, reporting queries) to run on a system with a large amount of transactional activity. This kind of scenario is often problematic for database systems where queries use read locks, since the query may either fail to acquire them or lock large amounts of data for a long time, thereby preventing transactional activity against that data and reducing concurrency. (An alternative that is used in some systems is to use a lower degree of consistency, such as degree-two consistency, but that could result in inconsistent query results.)

Oracle's concurrency model is used as a basis for the *Flashback Query* feature. This feature allows a user to set a certain SCN number or wall-clock time in his session and

perform queries on the data that existed at that point in time (provided that the data still exist in the rollback segment). Normally in a database system, once a change has been committed, there is no way to get back to the previous state of the data other than performing point-in-time recovery from backups. However, recovery of a very large database can be very costly, especially if the goal is just to retrieve some data item that had been inadvertently deleted by a user. The Flashback Query feature provides a much simpler mechanism to deal with user errors.

Oracle supports two ANSI/ISO isolation levels, "read committed" and "serializable". There is no support for dirty reads since it is not needed. The two isolation levels correspond to whether statement-level or transaction-level read consistency is used. The level can be set for a session or an individual transaction. Statement-level read consistency is the default.

Oracle uses row-level locking. Updates to different rows do not conflict. If two writers attempt to modify the same row, one waits until the other either commits or is rolled back, and then it can either return a write-conflict error or go ahead and modify the row. Locks are held for the duration of a transaction.

In addition to row-level locks that prevent inconsistencies due to DML activity, Oracle uses table locks that prevent inconsistencies due to DDL activity. These locks prevent one user from, say, dropping a table while another user has an uncommitted transaction that is accessing that table. Oracle does not use lock escalation to convert row locks to table locks for the purpose of its regular concurrency control.

Oracle detects deadlocks automatically and resolves them by rolling back one of the transactions involved in the deadlock.

Oracle supports autonomous transactions, which are independent transactions generated within other transactions. When Oracle invokes an autonomous transaction, it generates a new transaction in a separate context. The new transaction can be either committed or rolled back before control returns to the calling transaction. Oracle supports multiple levels of nesting of autonomous transactions.

25.5.2 Basic Structures for Recovery

In order to understand how Oracle recovers from a failure, such as a disk crash, it is important to understand the basic structures that are involved. In addition to the data files that contain tables and indices, there are control files, redo logs, archived redo logs, and rollback segments.

The control file contains various metadata that are needed to operate the database, including information about backups.

Oracle records any transactional modification of a database buffer in the redo log, which consists of two or more files. It logs the modification as part of the operation that causes it and regardless of whether the transaction eventually commits. It logs changes to indices and rollback segments as well as changes to table data. As the redo logs fill up, they are archived by one or several background processes (if the database is running in **archivelog** mode).

The rollback segment contains information about older versions of the data (that is, undo information). In addition to its role in Oracle's consistency model, the infor-

mation is used to restore the old version of data items when a transaction that has modified the data items is rolled back.

To be able to recover from a storage failure, the data files and control files should be backed up regularly. The frequency of the backup determines the worst-case recovery time, since it takes longer to recover if the backup is old. Oracle supports hot backups —backups performed on an online database that is subject to transactional activity.

During recovery from a backup, Oracle performs two steps to reach a consistent state of the database as it existed just prior to the failure. First, Oracle rolls forward by applying the (archived) redo logs to the backup. This action takes the database to a state that existed at the time of the failure, but not necessarily a consistent state since the redo logs include uncommitted data. Second, Oracle rolls back uncommitted transactions by using the rollback segment. The database is now in a consistent state.

Recovery on a database that has been subject to heavy transactional activity since the last backup can be time consuming. Oracle supports parallel recovery in which several processes are used to apply redo information simultaneously. Oracle provides a GUI tool, Recovery Manager, which automates most tasks associated with backup and recovery.

25.5.3 Managed Standby Databases

To ensure high availability, Oracle provides a managed standby database feature. (This feature is the same as remote backups, described in Section 17.10.) A standby database is a copy of the regular database that is installed on a separate system. If a catastrophic failure occurs on the primary system, the standby system is activated and takes over, thereby minimizing the effect of the failure on availability. Oracle keeps the standby database up to date by constantly applying archived redo logs that are shipped from the primary database. The backup database can be brought online in read-only mode and used for reporting and decision support queries.

25.6 System Architecture

Whenever an database application executes an SQL statement, there is an operating system process that executes code in the database server. Oracle can be configured so that the operating system process is *dedicated* exclusively to the statement it is processing or so that the process can be shared among multiple statements. The latter configuration, known as the *multithreaded server*, has somewhat different properties with regard to the process and memory architecture. We shall discuss the dedicated server architecture first and the multithreaded server architecture later.

25.6.1 Dedicated Server: Memory Structures

The memory used by Oracle falls mainly into three categories: software code areas, the system global area (SGA), and the program global area (PGA).

The system code areas are the parts of the memory where the Oracle server code resides. A PGA is allocated for each process to hold its local data and control informa-

tion. This area contains stack space for various session data and the private memory for the SQL statement that it is executing. It also contains memory for sorting and hashing operations that may occur during the evaluation of the statement.

The SGA is a memory area for structures that are shared among users. It is made up by several major structures, including:

- **The buffer cache.** This cache keeps frequently accessed data blocks (from tables as well as indices) in memory to reduce the need to perform physical disk I/O. A least recently used replacement policy is used except for blocks accessed during a full table scan. However, Oracle allows multiple buffer pools to be created that have different criteria for aging out data. Some Oracle operations bypass the buffer cache and read data directly from disk.
- **The redo log buffer.** This buffer contains the part of the redo log that has not yet been written to disk.
- **The shared pool.** Oracle seeks to maximize the number of users that can use the database concurrently by minimizing the amount of memory that is needed for each user. One important concept in this context is the ability to share the internal representation of SQL statements and procedural code written in PL/SQL. When multiple users execute the same SQL statement, they can share most data structures that represent the execution plan for the statement. Only data that is local to each specific invocation of the statement needs to be kept in private memory.

  The sharable parts of the data structures representing the SQL statement are stored in the shared pool, including the text of the statement. The caching of SQL statements in the shared pool also saves compilation time, since a new invocation of a statement that is already cached does not have to go through the complete compilation process. The determination of whether an SQL statement is the same as one existing in the shared pool is based on exact text matching and the setting of certain session parameters. Oracle can automatically replace constants in an SQL statement with bind variables; future queries that are the same except for the values of constants will then match the earlier query in the shared pool. The shared pool also contains caches for dictionary information and various control structures.

25.6.2 Dedicated Server: Process Structures

There are two types of processes that execute Oracle server code: server processes that process SQL statements and background processes that perform various administrative and performance-related tasks. Some of these processes are optional, and in some cases, multiple processes of the same type can be used for performance reasons. Some of the most important types of background processes are:

- **Database writer.** When a buffer is removed from the buffer cache, it must be written back to disk if it has been modified since it entered the cache. This task

is performed by the database writer processes, which help the performance of the system by freeing up space in the buffer cache.

- **Log writer.** The log writer process writes entries in the redo log buffer to the redo log file on disk. It also writes a commit record to disk whenever a transaction commits.
- **Checkpoint.** The checkpoint process updates the headers of the data file when a checkpoint occurs.
- **System monitor.** This process performs crash recovery if needed. It is also performs some space management to reclaim unused space in temporary segments.
- **Process monitor.** This process performs process recovery for server processes that fail, releasing resources and performing various cleanup operations.
- **Recoverer.** The recoverer process resolves failures and conducts cleanup for distributed transactions.
- **Archiver.** The archiver copies the online redo log file to an archived redo log every time the online log file fills up.

25.6.3 Multithreaded Server

The multithreaded server configuration increases the number of users that a given number of server processes can support by sharing server processes among statements. It differs from the dedicated server architecture in these major aspects:

- A background dispatch process routes user requests to the next available server process. In doing so, it uses a request queue and a response queue in the SGA. The dispatcher puts a new request in the request queue where it will be picked up by a server process. As a server process completes a request, it puts the result in the response queue to be picked up by the dispatcher and returned to the user.
- Since a server process is shared among multiple SQL statements, Oracle does not keep private data in the PGA. Instead, it stores the session-specific data in the SGA.

25.6.4 Oracle9i Real Application Clusters

Oracle9i Real Application Clusters is a feature that allows multiple instances of Oracle to run against the same database. (Recall that, in Oracle terminology, an instance is the combination of background processes and memory areas.) This feature enables Oracle to run on clustered and MPP (shared disk and shared nothing) hardware architectures. This feature was called Oracle Parallel Server in earlier versions of Oracle. The ability to cluster multiple nodes has important benefits for scalability and availability that are useful in both OLTP and data warehousing environments.

The scalability benefits of the feature are obvious, since more nodes mean more processing power. Oracle further optimizes the use of the hardware through features such as affinity and partitionwise joins.

Oracle9*i* Real Application Clusters can also be used to achieve high availability. If one node fails, the remaining ones are still available to the application accessing the database. The remaining instances will automatically roll back uncommitted transactions that were being processed on the failed node in order to prevent them from blocking activity on the remaining nodes.

Having multiple instances run against the same database gives rise to some technical issues that do not exist on a single instance. While it is sometimes possible to partition an application among nodes so that nodes rarely access the same data, there is always the possibility of overlaps, which affects locking and cache management. To address this, Oracle supports a distributed lock manager and the *cache fusion* feature, which allows data blocks to flow directly among caches on different instances using the interconnect, without being written to disk.

25.7 Replication, Distribution, and External Data

Oracle provides support for replication and distributed transactions with two-phase commit.

25.7.1 Replication

Oracle supports several types of replication. (See Section 19.2.1 for an introduction to replication.) In its simplest form, data in a master site are replicated to other sites in the form of *snapshots*. (The term "snapshot" in this context should not be confused with the concept of a read-consistent snapshot in the context of the concurrency model.) A snapshot does not have to contain all the master data—it can, for example, exclude certain columns from a table for security reasons. Oracle supports two types of snapshots: *read-only* and *updatable*. An updatable snapshot can be modified at a slave site and the modifications propagated to the master table. However, read-only snapshots allow for a wider range of snapshot definitions. For instance, a read-only snapshot can be defined in terms of set operations on tables at the master site.

Oracle also supports multiple master sites for the same data, where all master sites act as peers. A replicated table can be updated at any of the master sites and the update is propagated to the other sites. The updates can be propagated either asynchronously or synchronously.

For asynchronous replication, the update information is sent in batches to the other master sites and applied. Since the same data could be subject to conflicting modifications at different sites, conflict resolution based on some business rules might be needed. Oracle provides a number of of built-in conflict resolution methods and allows users to write their own if need be.

With synchronous replication, an update to one master site is propagated immediately to all other sites. If the update transaction fails at any master site, the update is rolled back at all sites.

25.7.2 Distributed Databases

Oracle supports queries and transactions spanning multiple databases on different systems. With the use of gateways, the remote systems can include non-Oracle databases. Oracle has built-in capability to optimize a query that includes tables at different sites, retrieve the relevant data, and return the result as if it had been a normal, local query. Oracle also transparently supports transactions spanning multiple sites by a built-in two-phase-commit protocol.

25.7.3 External Data Sources

Oracle has several mechanisms for supporting external data sources. The most common usage is in data warehousing when large amounts of data are regularly loaded from a transactional system.

25.7.3.1 SQL*Loader

Oracle has a direct load utility, SQL*Loader, that supports fast parallel loads of large amounts of data from external files. It supports a variety of data formats and it can perform various filtering operations on the data being loaded.

25.7.3.2 External Tables

Oracle allows external data sources, such as flat files, to be referenced in the **from** clause of a query as if they were regular tables. An external table is defined by metadata that describe the Oracle column types and the mapping of the external data into those columns. An access driver is also needed to access the external data. Oracle provides a default driver for flat files.

The external table feature is primarily intended for extraction, transformation, and loading (ETL) operations in a data warehousing environment. Data can be loaded into the data warehouse from a flat file using

create table *table* **as**
    select ... **from** < external table >
    where ...

By adding operations on the data in either the **select** list or **where** clause, transformations and filtering can be done as part of the same SQL statement. Since these operations can be expressed either in native SQL or in functions written in PL/SQL or Java, the external table feature provides a very powerful mechanism for expressing all kinds of data transformation and filtering operations. For scalability, the access to the external table can be parallelized by Oracle's parallel execution feature.

25.8 Database Administration Tools

Oracle provides users a number of tools for system management and application development.

25.8.1 Oracle Enterprise Manager

Oracle Enterprise Manager is Oracle's main tool for database systems management. It provides an easy-to-use graphical user interface (GUI) and a variety of wizards for schema management, security management, instance management, storage management, and job scheduling. It also provides performance monitoring and tools to help an administrator tune application SQL, access paths, and instance and data storage parameters. For example, it includes a wizard that can suggest what indices are the most cost-effective to create under a given workload.

25.8.2 Database Resource Management

A database administrator needs to be able to control how the processing power of the hardware is divided among users or groups of users. Some groups may execute interactive queries where response time is critical; others may execute long-running reports that can be run as batch jobs in the background when the system load is low. It is also important to be able to prevent a user from inadvertently submitting an extremely expensive ad hoc query that will unduly delay other users.

Oracle's Database Resource Management feature allows the database administrator to divide users into resource consumer groups with different priorities and properties. For example, a group of high-priority, interactive users may be guaranteed at least 60 percent of the CPU. The remainder, plus any part of the 60 percent not used up by the high-priority group, would be allocated among resource consumer groups with lower priority. A really low-priority group could get assigned 0 percent, which would mean that queries issued by this group would run only when there are spare CPU cycles available. Limits for the degree of parallelism for parallel execution can be set for each group. The database administrator can also set time limits for how long an SQL statement is allowed to run for each group. When a users submits a statement, the Resource Manager estimates how long it would take to execute it and returns an error if the statement violates the limit. The resource manager can also limit the number of user sessions that can be active concurrently for each resource consumer group.

Bibliographical Notes

Up-to-date product information, including documentation, on Oracle products can be found at the Web sites http://www.oracle.com and http://technet.oracle.com.

Extensible indexing in Oracle8i is described by Srinivasan et al. [2000b], while Srinivasan et al. [2000a] describe index organized tables in Oracle8i. Banerjee et al. [2000] describe XML support in Oracle8i. Bello et al. [1998] describe materialized views in Oracle. Antoshenkov [1995] describes the byte-aligned bitmap compression technique used in Oracle; see also Johnson [1999b].

The Oracle Parallel Server is described by Bamford et al. [1998]. Recovery in Oracle is described by Joshi et al. [1998] and Lahiri et al. [2001]. Messaging and queuing in Oracle are described by Gawlick [1998].

CHAPTER 9

Object-Relational Databases

Persistent programming languages add persistence and other database features to existing programming languages by using an existing object-oriented type system. In contrast, *object-relational data models* extend the relational data model by providing a richer type system including complex data types and object orientation. Relational query languages, in particular SQL, need to be correspondingly extended to deal with the richer type system. Such extensions attempt to preserve the relational foundations—in particular, the declarative access to data—while extending the modeling power. Object-relational database systems (that is, database systems based on the object-relation model) provide a convenient migration path for users of relational databases who wish to use object-oriented features.

We first present the motivation for the nested relational model, which allows relations that are not in first normal form, and allows direct representation of hierarchical structures. We then show how to extend SQL by adding a variety of object-relational features. Our discussion is based on the SQL:1999 standard.

Finally, we discuss differences between persistent programming languages and object-relational systems, and mention criteria for choosing between them.

9.1 Nested Relations

In Chapter 7, we defined *first normal form* (1NF), which requires that all attributes have *atomic domains*. Recall that a domain is *atomic* if elements of the domain are considered to be indivisible units.

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

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

Figure 9.1 Non-1NF books relation, *books*.

between the user's intuitive notion of an object and the database system's notion of a data item.

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

We illustrate nested relations by an example from a library. Suppose we store for each book the following information:

- Book title
- Set of authors
- Publisher
- Set of keywords

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

- **Authors**. A book may have a set of authors. Nevertheless, we may want to find all books of which Jones was one of the authors. Thus, we are interested in a subpart of the domain element "set of authors."
- **Keywords**. If we store a set of keywords for a book, we expect to be able to retrieve all books whose keywords include one or more keywords. Thus, we view the domain of the set of keywords as nonatomic.
- **Publisher**. Unlike *keywords* and *authors*, *publisher* does not have a set-valued domain. However, we may view *publisher* as consisting of the subfields *name* and *branch*. This view makes the domain of *publisher* nonatomic.

Figure 9.1 shows an example relation, *books*. The *books* relation can be represented in 1NF, as in Figure 9.2. Since we must have atomic domains in 1NF, yet want access to individual authors and to individual keywords, we need one tuple for each (keyword, author) pair. The *publisher* attribute is replaced in the 1NF version by two attributes: one for each subfield of *publisher*.

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

Figure 9.2 *flat-books*, a 1NF version of non-1NF relation *books*.

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

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

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

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

Figure 9.3 shows the projection of the relation *flat-books* of Figure 9.2 onto the preceding decomposition.

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

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

9.2 Complex Types

Nested relations are just one example of extensions to the basic relational model; other nonatomic data types, such as nested records, have also proved useful. The object-oriented data model has caused a need for features such as inheritance and references to objects. With complex type systems and object orientation, we can represent E-R model concepts, such as identity of entities, multivalued attributes, and generalization and specialization directly, without a complex translation to the relational model.

title	*author*
Compilers	Smith
Compilers	Jones
Networks	Jones
Networks	Frick

authors

title	*keyword*
Compilers	parsing
Compilers	analysis
Networks	Internet
Networks	Web

keywords

title	*pub-name*	*pub-branch*
Compilers	McGraw-Hill	New York
Networks	Oxford	London

books4

Figure 9.3 4NF version of the relation *flat-books* of Figure 9.2.

In this section, we describe extensions to SQL to allow complex types, including nested relations, and object-oriented features. Our presentation is based on the SQL:1999 standard, but we also outline features that are not currently in the standard but may be introduced in future versions of SQL standards.

9.2.1 Collection and Large Object Types

Consider this fragment of code.

```
create table books (
        ...
        keyword-set setof(varchar(20))
        ...
)
```

This table definition differs from table definitions in ordinary relational databases, since it allows attributes that are **sets**, thereby permitting multivalued attributes of E-R diagrams to be represented directly.

Sets are an instance of **collection type**s. Other instances of collection types include **array**s and **multiset**s (that is, unordered collections, where an element may occur multiple times). The following attribute definitions illustrate the declaration of an array:

author-array **varchar**(20) **array** [10]

Here, *author-array* is an array of up to 10 author names. We can access elements of an array by specifying the array index, for example *author-array*[1].

Arrays are the only collection type supported by SQL:1999; the syntax used is as in the preceding declaration. SQL:1999 does *not* support unordered sets or multisets, although they may appear in future versions of SQL.[1]

Many current-generation database applications need to store attributes that can be large (of the order of many kilobytes), such as a photograph of a person, or very large (of the order of many megabytes or even gigabytes), such as a high-resolution medical image or video clip. SQL:1999 therefore provides new large-object data types for character data (**clob**) and binary data (**blob**). The letters "lob" in these data types stand for "Large OBject". For example, we may declare attributes

book-review **clob**(10KB)
image **blob**(10MB)
movie **blob**(2GB))

Large objects are typically used in external applications, and it makes little sense to retrieve them in their entirety by SQL. Instead, an application would usually retrieve a "locator" for a large object and then use the locator to manipulate the object from the host language. For instance, JDBC permits the programmer to fetch a large object in small pieces, rather than all at once, much like fetching data from an operating system file.

9.2.2 Structured Types

Structured types can be declared and used in SQL:1999 as in the following example:

```
create type Publisher as
      (name varchar(20),
       branch varchar(20))
create type Book as
      (title varchar(20),
       author-array varchar(20) array [10],
       pub-date date,
       publisher Publisher,
       keyword-set setof(varchar(20)))
create table books of Book
```

The first statement defines a type called *Publisher*, which has two components: a name and a branch. The second statement defines a structured type *Book*, which contains a *title*, an *author-array*, which is an array of authors, a publication date, a publisher (of type *Publisher*), and a set of keywords. (The declaration of *keyword-set* as a set uses our extended syntax, and is not supported by the SQL:1999 standard.) The types illustrated above are called **structured type**s in SQL:1999.

1. The Oracle 8 database system supports nested relations, but uses a syntax different from that in this chapter.

Finally, a table *books* containing tuples of type *Book* is created. The table is similar to the nested relation *books* in Figure 9.1, except we have decided to create an array of author names instead of a set of author names. The array permits us to record the order of author names.

Structured types allow composite attributes of E-R diagrams to be represented directly. Unnamed **row type**s can also be used in SQL:1999 to define composite attributes. For instance, we could have defined an attribute *publisher1* as

```
publisher1 row (name varchar(20),
                branch varchar(20))
```

instead of creating a named type *Publisher*.

We can of course create tables without creating an intermediate type for the table. For example, the table *books* could also be defined as follows:

```
create table books
    (title varchar(20),
     author-array varchar(20) array[10],
     pub-date date,
     publisher Publisher,
     keyword-set setof(varchar(20)))
```

With the above declaration, there is no explicit type for rows of the table. [2]

A structured type can have **method**s defined on it. We declare methods as part of the type definition of a structured type:

```
create type Employee as (
    name varchar(20),
    salary integer )
method giveraise (percent integer)
```

We create the method body separately:

```
create method giveraise (percent integer) for Employee
begin
    set self.salary = self.salary + (self.salary * percent) / 100;
end
```

The variable **self** refers to the structured type instance on which the method is invoked. The body of the method can contain procedural statements, which we shall study in Section 9.6.

2. In Oracle PL/SQL, given a table *t*, *t*%**rowtype** denotes the type of the rows of the table. Similarly, *t.a*%**type** denotes the type of attribute *a* of table *t*.

9.2.3 Creation of Values of Complex Types

In SQL:1999 **constructor function**s are used to create values of structured types. A function with the same name as a structured type is a constructor function for the structured type. For instance, we could declare a constructor for the type *Publisher* like this:

```
create function Publisher (n varchar(20), b varchar(20))
returns Publisher
begin
    set name = n;
    set branch = b;
end
```

We can then use *Publisher*('McGraw-Hill', 'New York') to create a value of the type *Publisher*.

SQL:1999 also supports functions other than constructors, as we shall see in Section 9.6; the names of such functions must be different from the name of any structured type.

Note that in SQL:1999, unlike in object-oriented databases, a constructor creates a value of the type, not an object of the type. That is, the value the constructor creates has no object identity. In SQL:1999 objects correspond to tuples of a relation, and are created by inserting a tuple in a relation.

By default every structured type has a constructor with no arguments, which sets the attributes to their default values. Any other constructors have to be created explicitly. There can be more than one constructor for the same structured type; although they have the same name, they must be distinguishable by the number of arguments and types of their arguments.

An array of values can be created in SQL:1999 in this way:

array['Silberschatz', 'Korth', 'Sudarshan']

We can construct a row value by listing its attributes within parentheses. For instance, if we declare an attribute *publisher1* as a row type (as in Section 9.2.2), we can construct this value for it:

('McGraw-Hill', 'New York')

without using a constructor.

We create set-valued attributes, such as *keyword-set*, by enumerating their elements within parentheses following the keyword **set**. We can create multiset values just like set values, by replacing **set** by **multiset**.[3]

Thus, we can create a tuple of the type defined by the *books* relation as:

('Compilers', **array**['Smith', 'Jones'], *Publisher*('McGraw-Hill', 'New York'),
set('parsing', 'analysis'))

3. Although sets and multisets are not part of the SQL:1999 standard, the other constructs shown in this section are part of the standard. Future versions of SQL are likely to support sets and multisets.

Here we have created a value for the attribute *Publisher* by invoking a *constructor* function for *Publisher* with appropriate arguments.

If we want to insert the preceding tuple into the relation *books*, we could execute the statement

```
insert into books
values
('Compilers', array['Smith', 'Jones'], Publisher('McGraw-Hill', 'New York'),
                                                    set('parsing', 'analysis'))
```

9.3 Inheritance

Inheritance can be at the level of types, or at the level of tables. We first consider inheritance of types, then inheritance at the level of tables.

9.3.1 Type Inheritance

Suppose that we have the following type definition for people:

```
create type Person
    (name varchar(20),
     address varchar(20))
```

We may want to store extra information in the database about people who are students, and about people who are teachers. Since students and teachers are also people, we can use inheritance to define the student and teacher types in SQL:1999:

```
create type Student
    under Person
    (degree varchar(20),
     department varchar(20))
create type Teacher
    under Person
    (salary integer,
     department varchar(20))
```

Both *Student* and *Teacher* inherit the attributes of *Person*—namely, *name* and *address*. *Student* and *Teacher* are said to be subtypes of *Person*, and *Person* is a supertype of *Student*, as well as of *Teacher*.

Methods of a structured type are inherited by its subtypes, just as attributes are. However, a subtype can redefine the effect of a method by declaring the method again, using **overriding method** in place of **method** in the method declaration.

Now suppose that we want to store information about teaching assistants, who are simultaneously students and teachers, perhaps even in different departments. We can do this by using **multiple inheritance**, which we studied in Chapter 8. The SQL:1999 standard does not support multiple inheritance. However, draft versions of the SQL:1999 standard provided for multiple inheritance, and although the final

SQL:1999 omitted it, future versions of the SQL standard may introduce it. We base our discussion on the draft versions of the SQL:1999 standard.

For instance, if our type system supports multiple inheritance, we can define a type for teaching assistant as follows:

create type *TeachingAssistant*
under *Student*, *Teacher*

TeachingAssistant would inherit all the attributes of *Student* and *Teacher*. There is a problem, however, since the attributes *name*, *address*, and *department* are present in *Student*, as well as in *Teacher*.

The attributes *name* and *address* are actually inherited from a common source, *Person*. So there is no conflict caused by inheriting them from *Student* as well as *Teacher*. However, the attribute *department* is defined separately in *Student* and *Teacher*. In fact, a teaching assistant may be a student of one department and a teacher in another department. To avoid a conflict between the two occurrences of *department*, we can rename them by using an **as** clause, as in this definition of the type *TeachingAssistant*:

create type *TeachingAssistant*
under *Student* **with** (*department* **as** *student-dept*),
Teacher **with** (*department* **as** *teacher-dept*)

We note that SQL:1999 supports only single inheritance— that is, a type can inherit from only a single type; the syntax used is as in our earlier examples. Multiple inheritance as in the *TeachingAssistant* example is *not* supported in SQL:1999. The SQL:1999 standard also requires an extra field at the end of the type definition, whose value is either **final** or **not final**. The keyword **final** says that subtypes may not be created from the given type, while **not final** says that subtypes may be created.

In SQL as in most other languages, a value of a structured type must have exactly one "most-specific type." That is, each value must be associated with one specific type, called its **most-specific type**, when it is created. By means of inheritance, it is also associated with each of the supertypes of its most specific type. For example, suppose that an entity has the type *Person*, as well as the type *Student*. Then, the most-specific type of the entity is *Student*, since *Student* is a subtype of *Person*. However, an entity cannot have the type *Student*, as well as the type *Teacher*, unless it has a type, such as *TeachingAssistant*, that is a subtype of *Teacher*, as well as of *Student*.

9.3.2 Table Inheritance

Subtables in SQL:1999 correspond to the E-R notion of specialization/generalization. For instance, suppose we define the *people* table as follows:

create table *people* **of** *Person*

We can then define tables *students* and *teachers* as **subtable**s of *people*, as follows:

```
create table students of Student
    under people
create table teachers of Teacher
    under people
```

The types of the subtables must be subtypes of the type of the parent table. Thereby, every attribute present in *people* is also present in the subtables.

Further, when we declare *students* and *teachers* as subtables of *people*, every tuple present in *students* or *teachers* becomes also implicitly present in *people*. Thus, if a query uses the table *people*, it will find not only tuples directly inserted into that table, but also tuples inserted into its subtables, namely *students* and *teachers*. However, only those attributes that are present in *people* can be accessed.

Multiple inheritance is possible with tables, just as it is possible with types. (We note, however, that multiple inheritance of tables is not supported by SQL:1999.) For example, we can create a table of type *TeachingAssistant*:

```
create table teaching-assistants
of TeachingAssistant
    under students, teachers
```

As a result of the declaration, every tuple present in the *teaching-assistants* table is also implicitly present in the *teachers* and in the *students* table, and in turn in the *people* table.

SQL:1999 permits us to find tuples that are in *people* but not in its subtables by using "**only** *people*" in place of *people* in a query.

There are some consistency requirements for subtables. Before we state the constraints, we need a definition: We say that tuples in a subtable **corresponds** to tuples in a parent table if they have the same values for all inherited attributes. Thus, corresponding tuples represent the same entity.

The consistency requirements for subtables are:

1. Each tuple of the supertable can correspond to at most one tuple in each of its immediate subtables.
2. SQL:1999 has an additional constraint that all the tuples corresponding to each other must be derived from one tuple (inserted into one table).

For example, without the first condition, we could have two tuples in *students* (or *teachers*) that correspond to the same person.

The second condition rules out a tuple in *people* corresponding to both a tuple in *students* and a tuple in *teachers*, unless all these tuples are implicitly present because a tuple was inserted in a table *teaching-assistants*, which is a subtable of both *teachers* and *students*.

Since SQL:1999 does not support multiple inheritance, the second condition actually prevents a person from being both a teacher and a student. The same problem would arise if the subtable *teaching-assistants* is absent, even if multiple inheritance were supported. Obviously it would be useful to model a situation where a person

can be a teacher and a student, even if a common subtable *teaching-assistants* is not present. Thus, it can be useful to remove the second consistency constraint. We return to this issue in Section 9.3.3.

Subtables can be stored in an efficient manner without replication of all inherited fields, in one of two ways:

- Each table stores the primary key (which may be inherited from a parent table) and the attributes defined locally. Inherited attributes (other than the primary key) do not need to be stored, and can be derived by means of a join with the supertable, based on the primary key.
- Each table stores all inherited and locally defined attributes. When a tuple is inserted, it is stored only in the table in which it is inserted, and its presence is inferred in each of the supertables. Access to all attributes of a tuple is faster, since a join is not required. However, in case the second consistency constraint is absent—that is, an entity can be represented in two subtables without being present in a common subtable of both—this representation can result in replication of information.

9.3.3 Overlapping Subtables

Inheritance of types should be used with care. A university database may have many subtypes of *Person*, such as *Student*, *Teacher*, *FootballPlayer*, *ForeignCitizen*, and so on. *Student* may itself have subtypes such as *UndergraduateStudent*, *GraduateStudent*, and *PartTimeStudent*. Clearly, a person can belong to several of these categories at once. As Chapter 8 mentions, each of these categories is sometimes called a *role*.

For each entity to have exactly one most-specific type, we would have to create a subtype for every possible combination of the supertypes. In the preceding example, we would have subtypes such as *ForeignUndergraduateStudent*, *ForeignGraduateStudentFootballPlayer*, and so on. Unfortunately, we would end up with an enormous number of subtypes of *Person*.

A better approach in the context of database systems is to allow an object to have multiple types, without having a most-specific type. Object-relational systems can model such a feature by using inheritance at the level of tables, rather than of types, and allowing an entity to exist in more than one table at once.

For example, suppose we again have the type *Person*, with subtypes *Student* and *Teacher*, and the corresponding table *people*, with subtables *teachers* and *students*. We can then have a tuple in *teachers* and a tuple in *students* corresponding to the same tuple in *people*.

There is no need to have a type *TeachingAssistant* that is a subtype of both *Student* and *Teacher*. We need not create a type *TeachingAssistant* unless we wish to store extra attributes or redefine methods in a manner specific to people who are both students and teachers.

We note, however, that SQL:1999 prohibits such a situation, because of consistency requirement 2 from Section 9.3.2. Since SQL:1999 also does not support multiple inheritance, we cannot use inheritance to model a situation where a person can be both a student and a teacher. We can of course create separate tables to represent the

information without using inheritance. We would have to add appropriate referential integrity constraints to ensure that students and teachers are also represented in the *people* table.

9.4 Reference Types

Object-oriented languages provide the ability to refer to objects. An attribute of a type can be a reference to an object of a specified type. For example, in SQL:1999 we can define a type *Department* with a field *name* and a field *head* which is a reference to the type *Person*, and a table *departments* of type *Department*, as follows:

```
create type Department (
        name varchar(20),
        head ref(Person) scope people
)
create table departments of Department
```

Here, the reference is restricted to tuples of the table *people*. The restriction of the **scope** of a reference to tuples of a table is mandatory in SQL:1999, and it makes references behave like foreign keys.

We can omit the declaration **scope** *people* from the type declaration and instead make an addition to the **create table** statement:

```
create table departments of Department
        (head with options scope people)
```

In order to initialize a reference attribute, we need to get the identifier of the tuple that is to be referenced. We can get the identifier value of a tuple by means of a query. Thus, to create a tuple with the reference value, we may first create the tuple with a null reference and then set the reference separately:

```
insert into departments
        values ('CS', null)
update departments
        set head = (select ref(p)
                    from people as p
                    where name = 'John')
where name = 'CS'
```

This syntax for accessing the identifier of a tuple is based on the Oracle syntax. SQL:1999 adopts a different approach, one where the referenced table must have an attribute that stores the identifier of the tuple. We declare this attribute, called the **self-referential attribute**, by adding a **ref is** clause to the **create table** statement:

```
create table people of Person
ref is oid system generated
```

Here, *oid* is an attribute name, not a keyword. The subquery above would then use

select *p.oid*

instead of **select ref**(*p*).

An alternative to system-generated identifiers is to allow users to generate identifiers. The type of the self-referential attribute must be specified as part of the type definition of the referenced table, and the table definition must specify that the reference is **user generated**:

```
create type Person
      (name varchar(20),
       address varchar(20))
      ref using varchar(20)
create table people of Person
      ref is oid user generated
```

When inserting a tuple in *people*, we must provide a value for the identifier:

```
insert into people values
      ('01284567', 'John', '23 Coyote Run')
```

No other tuple for *people* or its supertables or subtables can have the same identifier. We can then use the identifier value when inserting a tuple into *departments*, without the need for a separate query to retrieve the identifier:

```
insert into departments
      values ('CS', '01284567')
```

It is even possible to use an existing primary key value as the identifier, by including the **ref from** clause in the type definition:

```
create type Person
      (name varchar(20) primary key,
       address varchar(20))
      ref from(name)
create table people of Person
      ref is oid derived
```

Note that the table definition must specify that the reference is derived, and must still specify a self-referential attribute name. When inserting a tuple for *departments*, we can then use

```
insert into departments
      values ('CS', 'John')
```

9.5 Querying with Complex Types

In this section, we present extensions of the SQL query language to deal with complex types. Let us start with a simple example: Find the title and the name of the publisher of each book. This query carries out the task:

select *title*, *publisher.name*
from *books*

Notice that the field *name* of the composite attribute *publisher* is referred to by a dot notation.

9.5.1 Path Expressions

References are dereferenced in SQL:1999 by the $->$ symbol. Consider the *departments* table defined earlier. We can use this query to find the names and addresses of the heads of all departments:

select *head*$->$*name*, *head*$->$*address*
from *departments*

An expression such as "*head*$->$*name*" is called a **path expression**.

Since *head* is a reference to a tuple in the *people* table, the attribute *name* in the preceding query is the *name* attribute of the tuple from the *people* table. References can be used to hide join operations; in the preceding example, without the references, the *head* field of *department* would be declared a foreign key of the table *people*. To find the name and address of the head of a department, we would require an explicit join of the relations *departments* and *people*. The use of references simplifies the query considerably.

9.5.2 Collection-Valued Attributes

We now consider how to handle collection-valued attributes. Arrays are the only collection type supported by SQL:1999, but we use the same syntax for relation-valued attributes also. An expression evaluating to a collection can appear anywhere that a relation name may appear, such as in a **from** clause, as the following paragraphs illustrate. We use the table *books* which we defined earlier.

If we want to find all books that have the word "database" as one of their keywords, we can use this query:

select *title*
from *books*
where 'database' **in** (**unnest**(*keyword-set*))

Note that we have used **unnest**(*keyword-set*) in a position where SQL without nested relations would have required a **select-from-where** subexpression.

If we know that a particular book has three authors, we could write:

```
select author-array[1], author-array[2], author-array[3]
from books
where title = 'Database System Concepts'
```

Now, suppose that we want a relation containing pairs of the form "title, author-name" for each book and each author of the book. We can use this query:

```
select B.title, A.name
from books as B, unnest(B.author-array) as A
```

Since the *author-array* attribute of *books* is a collection-valued field, it can be used in a **from** clause, where a relation is expected.

9.5.3 Nesting and Unnesting

The transformation of a nested relation into a form with fewer (or no) relation-valued attributes is called **unnesting**. The *books* relation has two attributes, *author-array* and *keyword-set*, that are collections, and two attributes, *title* and *publisher*, that are not. Suppose that we want to convert the relation into a single flat relation, with no nested relations or structured types as attributes. We can use the following query to carry out the task:

```
select title, A as author, publisher.name as pub-name, publisher.branch
        as pub-branch, K as keyword
from books as B, unnest(B.author-array) as A, unnest (B.keyword-set) as K
```

The variable *B* in the from clause is declared to range over *books*. The variable *A* is declared to range over the authors in *author-array* for the book *B*, and *K* is declared to range over the keywords in the *keyword-set* of the book *B*. Figure 9.1 (in Section 9.1) shows an instance *books* relation, and Figure 9.2 shows the 1NF relation that is the result of the preceding query.

The reverse process of transforming a 1NF relation into a nested relation is called **nesting**. Nesting can be carried out by an extension of grouping in SQL. In the normal use of grouping in SQL, a temporary multiset relation is (logically) created for each group, and an aggregate function is applied on the temporary relation. By returning the multiset instead of applying the aggregate function, we can create a nested relation. Suppose that we are given a 1NF relation *flat-books*, as in Figure 9.2. The following query nests the relation on the attribute *keyword*:

```
select title, author, Publisher(pub-name, pub-branch) as publisher,
        set(keyword) as keyword-set
from flat-books
groupby title, author, publisher
```

The result of the query on the *books* relation from Figure 9.2 appears in Figure 9.4. If we want to nest the author attribute as well, and thereby to convert the 1NF table

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

Figure 9.4 A partially nested version of the *flat-books* relation.

flat-books in Figure 9.2 back to the nested table *books* in Figure 9.1, we can use the query:

select *title*, **set**(*author*) **as** *author-set*, *Publisher*(*pub-name, pub-branch*) **as** *publisher*,
set(*keyword*) **as** *keyword-set*
from *flat-books*
groupby *title, publisher*

Another approach to creating nested relations is to use subqueries in the **select** clause. The following query, which performs the same task as the previous query, illustrates this approach.

select *title*,
(**select** *author*
from *flat-books* **as** *M*
where *M.title* = *O.title*) **as** *author-set*,
Publisher(*pub-name, pub-branch*) **as** *publisher*,
(**select** *keyword*
from *flat-books* **as** *N*
where *N.title* = *O.title*) **as** *keyword-set*,
from *flat-books* **as** *O*

The system executes the nested subqueries in the **select** clause for each tuple generated by the **from** and *where* clauses of the outer query. Observe that the attribute *O.title* from the outer query is used in the nested queries, to ensure that only the correct sets of authors and keywords are generated for each title. An advantage of this approach is that an **orderby** clause can be used in the nested query, to generate results in a desired order. An array or a list could be constructed from the result of the nested query. Without such an ordering, arrays and lists would not be uniquely determined.

We note that while unnesting of array-valued attributes can be carried out in SQL:1999 as shown above, the reverse process of nesting is not supported in SQL:1999. The extensions we have shown for nesting illustrate features from some proposals for extending SQL, but are not part of any standard currently.

9.6 Functions and Procedures

SQL:1999 allows the definition of functions, procedures, and methods. These can be defined either by the procedural component of SQL:1999, or by an external programming language such as Java, C, or C++. We look at definitions in SQL:1999 first, and then see how to use definitions in external languages. Several database systems support their own procedural languages, such as PL/SQL in Oracle and TransactSQL in Microsoft SQLServer. These resemble the procedural part of SQL:1999, but there are differences in syntax and semantics; see the respective system manuals for further details.

9.6.1 SQL Functions and Procedures

Suppose that we want a function that, given the title of a book, returns the count of the number of authors, using the 4NF schema. We can define the function this way:

```
create function author-count(title varchar(20))
    returns integer
    begin
    declare a-count integer;
        select count(author) into a-count
        from authors
        where authors.title = title
    return a-count;
    end
```

This function can be used in a query that returns the titles of all books that have more than one author:

```
select title
from books4
where author-count(title) > 1
```

Functions are particularly useful with specialized data types such as images and geometric objects. For instance, a polygon data type used in a map database may have an associated function that checks if two polygons overlap, and an image data type may have associated functions to compare two images for similarity. Functions may be written in an external language such as C, as we see in Section 9.6.2. Some database systems also support functions that return relations, that is, multisets of tuples, although such functions are not supported by SQL:1999.

Methods, which we saw in Section 9.2.2, can be viewed as functions associated with structured types. They have an implicit first parameter called **self**, which is set to the structured type value on which the method is invoked. Thus, the body of the method can refer to an attribute a of the value by using **self**.a. These attributes can also be updated by the method.

SQL:1999 also supports procedures. The *author-count* function could instead be written as a procedure:

```
create procedure author-count-proc(in title varchar(20), out a-count integer)
    begin
        select count(author) into a-count
        from authors
        where authors.title = title
    end
```

Procedures can be invoked either from an SQL procedure or from embedded SQL by the **call** statement:

```
declare a-count integer;
call author-count-proc('Database Systems Concepts', a-count);
```

SQL:1999 permits more than one procedure of the same name, so long as the number of arguments of the procedures with the same name is different. The name, along with the number of arguments, is used to identify the procedure. SQL:1999 also permits more than one function with the same name, so long as the different functions with the same name either have different numbers of arguments, or for functions with the same number of arguments, differ in the type of at least one argument.

9.6.2 External Language Routines

SQL:1999 allows us to define functions in a programming language such as C or C++. Functions defined in this fashion can be more efficient than functions defined in SQL, and computations that cannot be carried out in SQL can be executed by these functions. An example of the use of such functions would be to perform a complex arithmetic computation on the data in a tuple.

External procedures and functions can be specified in this way:

```
create procedure author-count-proc( in title varchar(20), out count integer)
language C
external name '/usr/avi/bin/author-count-proc'

create function author-count (title varchar(20))
returns integer
language C
external name '/usr/avi/bin/author-count'
```

The external language procedures need to deal with null values and exceptions. They must therefore have several extra parameters: an **sqlstate** value to indicate failure/success status, a parameter to store the return value of the function, and indicator variables for each parameter/function result to indicate if the value is null. An extra line **parameter style general** added to the declaration above indicates that the external procedures/functions take only the arguments shown and do not deal with null values or exceptions.

Functions defined in a programming language and compiled outside the database system may be loaded and executed with the database system code. However, do-

ing so carries the risk that a bug in the program can corrupt the database internal structures, and can bypass the access-control functionality of the database system. Database systems that are concerned more about efficient performance than about security may execute procedures in such a fashion.

Database systems that are concerned about security would typically execute such code as part of a separate process, communicate the parameter values to it, and fetch results back, via interprocess communication.

If the code is written in a language such as Java, there is a third possibility: executing the code in a "*sandbox*" within the database process itself. The sandbox prevents the Java code from carrying out any reads or updates directly on the database.

9.6.3 Procedural Constructs

SQL:1999 supports a variety of procedural constructs, which gives it almost all the power of a general purpose programming language. The part of the SQL:1999 standard that deals with these constructs is called the **Persistent Storage Module (PSM)**.

A compound statement is of the form **begin** ... **end**, and it may contain multiple SQL statements between the **begin** and the **end**. Local variables can be declared within a compound statement, as we have seen in Section 9.6.1.

SQL:1999 supports while statements and repeat statements by this syntax:

```
declare n integer default 0;
while n < 10 do
    set n = n + 1;
end while
repeat
    set n = n − 1;
until n = 0
end repeat
```

This code does not do anything useful; it is simply meant to show the syntax of while and repeat loops. We will see more meaningful uses later.

There is also a **for** loop, which permits iteration over all results of a query:

```
declare n integer default 0;
for r as
        select balance from account
        where branch-name = 'Perryridge'
do
    set n = n+ r.balance
end for
```

The program implicitly opens a cursor when the **for** loop begins execution and uses it to fetch the values one row at a time into the **for** loop variable (*r*, in the above example). It is possible to give a name to the cursor, by inserting the text *cn* **cursor for** just after the keyword **as**, where *cn* is the name we wish to give to the cursor. The cursor

name can be used to perform update/delete operations on the tuple being pointed to by the cursor. The statement **leave** can be used to exit the loop, while **iterate** starts on the next tuple, from the beginning of the loop, skipping the remaining statements.

The conditional statements supported by SQL:1999 include if-then-else statements statements by using this syntax:

if *r.balance* < 1000
  then set $l = l +$ *r.balance*
elseif *r.balance* < 5000
  then set $m = m +$ *r.balance*
else set $h = h +$ *r.balance*
end if

This code assumes that l, m, and h are integer variables, and r is a row variable. If we replace the line "**set** $n = n +$ *r.balance*" in the **for** loop of the preceding paragraph by the **if-then-else** code, the loop would compute the total balances of accounts that fall under the low, medium, and high balance categories respectively.

SQL:1999 also supports a case statement similar to the C/C++ language case statement (in addition to case expressions, which we saw in Chapter 4).

Finally, SQL:1999 includes the concept of signaling **exception conditions**, and declaring **handlers** that can handle the exception, as in this code:

declare *out-of-stock* **condition**
declare exit handler for *out-of-stock*
begin
. . .
end

The statements between the **begin** and the **end** can raise an exception by executing **signal** *out-of-stock*. The handler says that if the condition arises, the action to be taken is to exit the enclosing **begin end** statement. Alternative actions would be **continue**, which continues execution from the next statement following the one that raised the exception. In addition to explicitly defined conditions, there are also predefined conditions such as **sqlexception**, **sqlwarning**, and **not found**.

Figure 9.5 provides a larger example of the use of SQL:1999 procedural constructs. The procedure *findEmpl* computes the set of all direct and indirect employees of a given manager (specified by the parameter *mgr*), and stores the resulting employee names in a relation called *empl*, which is assumed to exist already. The relation *manager*(*empname, mgrname*), specifying who works directly for which manager, is assumed to be available. The set of all direct/indirect employees is basically the transitive closure of the relation *manager*. We saw how to express such a query by recursion in Chapter 5 (Section 5.2.6).

The procedure uses two temporary tables, *newemp* and *temp*. The procedure inserts all employees who directly work for *mgr* into *newemp* before the **repeat** loop. The **repeat** loop first adds all employees in *newemp* to *empl*. Next, it computes employees who work for those in *newemp*, except those who have already been found to be

```
create procedure findEmpl(in mgr char(10))
– – Finds all employees who work directly or indirectly for mgr
– – and adds them to the relation empl(name).
– – The relation manager(empname, mgrname) specifies who directly
– – works for whom.
begin
    create temporary table newemp (name char(10));
    create temporary table temp (name char(10));
    insert into newemp
        select empname
        from manager
        where mgrname = mgr
    repeat
        insert into empl
            select name
            from newemp;

        insert into temp
            (select manager.empname
                from newemp, manager
                where newemp.empname = manager.mgrname;
            )
            except (
                select empname
                from empl
            );
        delete from newemp;
        insert into newemp
            select *
            from temp;
        delete from temp;

    until not exists (select * from newemp)
    end repeat;
end
```

Figure 9.5 Finding all employees of a manager.

employees of *mgr*, and stores them in the temporary table *temp*. Finally, it replaces the contents of *newemp* by the contents of *temp*. The **repeat** loop terminates when it finds no new (indirect) employees.

We note that the use of the **except** clause in the procedure ensures that the procedure works even in the (abnormal) case where there is a cycle of management. For example, if *a* works for *b*, *b* works for *c*, and *c* works for *a*, there is a cycle.

While cycles may be unrealistic in management control, cycles are possible in other applications. For instance, suppose we have a relation *flights*(*to*, *from*) that says which

cities can be reached from which other cities by a direct flight. We can modify the *findEmpl* procedure to find all cities that are reachable by a sequence of one or more flights from a given city. All we have to do is to replace *manager* by *flight* and replace attribute names correspondingly. In this situation there can be cycles of reachability, but the procedure would work correctly since it would eliminate cities that have already been seen.

9.7 Object-Oriented versus Object-Relational

We have now studied object-oriented databases built around persistent programming languages, as well as object-relational databases, which are object-oriented databases built on top of the relation model. Database systems of both types are on the market, and a database designer needs to choose the kind of system that is appropriate to the needs of the application.

Persistent extensions to programming languages and object-relational systems target different markets. The declarative nature and limited power (compared to a programming language) of the SQL language provides good protection of data from programming errors, and makes high-level optimizations, such as reducing I/O, relatively easy. (We cover optimization of relational expressions in Chapter 13.) Object-relational systems aim at making data modeling and querying easier by using complex data types. Typical applications include storage and querying of complex data, including multimedia data.

A declarative language such as SQL, however, imposes a significant performance penalty for certain kinds of applications that run primarily in main memory, and that perform a large number of accesses to the database. Persistent programming languages target such applications that have high performance requirements. They provide low-overhead access to persistent data, and eliminate the need for data translation if the data are to be manipulated by a programming language. However, they are more susceptible to data corruption by programming errors, and usually do not have a powerful querying capability. Typical applications include CAD databases.

We can summarize the strengths of the various kinds of database systems in this way:

- **Relational systems**: simple data types, powerful query languages, high protection
- **Persistent-programming-language–based OODBs**: complex data types, integration with programming language, high performance
- **Object-relational systems**: complex data types, powerful query languages, high protection

These descriptions hold in general, but keep in mind that some database systems blur the boundaries. For example, some object-oriented database systems built around a persistent programming language are implemented on top of a relational database system. Such systems may provide lower performance than object-oriented database systems built directly on a storage system, but provide some of the stronger protection guarantees of relational systems.

Many object-relational database systems are built on top of existing relational database systems. To do so, the complex data types supported by object-relational systems need to be translated to the simpler type system of relational databases.

To understand how the translation is done, we need only look at how some features of the E-R model are translated into relations. For instance, multivalued attributes in the E-R model correspond to set-valued attributes in the object-relational model. Composite attributes roughly correspond to structured types. ISA hierarchies in the E-R model correspond to table inheritance in the object-relational model. The techniques for converting E-R model features to tables, which we saw in Section 2.9, can be used, with some extensions, to translate object-relational data to relational data.

9.8 Summary

- The object-relational data model extends the relational data model by providing a richer type system including collection types, and object orientation.
- Object orientation provides inheritance with subtypes and subtables, as well as object (tuple) references.
- Collection types include nested relations, sets, multisets, and arrays, and the object-relational model permits attributes of a table to be collections.
- The SQL:1999 standard extends the SQL data definition and query language to deal with the new data types and with object orientation.
- We saw a variety of features of the extended data-definition language, as well as the query language, and in particular support for collection-valued attributes, inheritance, and tuple references. Such extensions attempt to preserve the relational foundations—in particular, the declarative access to data—while extending the modeling power.
- Object-relational database systems (that is, database systems based on the object-relation model) provide a convenient migration path for users of relational databases who wish to use object-oriented features.
- We have also outlined the procedural extensions provided by SQL:1999.
- We discussed differences between persistent programming languages and object-relational systems, and mention criteria for choosing between them.

Review Terms

- Nested relations
- Nested relational model
- Complex types
- Collection types
- Large object types
- Sets
- Arrays
- Multisets
- Character large object (clob)
- Binary large object (blob)

- Structured types
- Methods
- Row types
- Constructors
- Inheritance
  - Single inheritance
  - Multiple inheritance
- Type inheritance
- Most-specific type
- Table inheritance
- Subtable
- Overlapping subtables
- Reference types
- Scope of a reference
- Self-referential attribute
- Path expressions
- Nesting and unnesting
- SQL functions and procedures
- Procedural constructs
- Exceptions
- Handlers
- External language routines

Exercises

9.1 Consider the database schema

Emp = (ename, **setof***(Children),* **setof***(Skills))*
Children = (name, Birthday)
Birthday = (day, month, year)
Skills = (type, **setof***(Exams))*
Exams = (year, city)

Assume that attributes of type **setof***(Children)*, **setof***(Skills)*, and **setof***(Exams)*, have attribute names *ChildrenSet*, *SkillsSet*, and *ExamsSet*, respectively. Suppose that the database contains a relation *emp (Emp)*. Write the following queries in SQL:1999 (with the extensions described in this chapter).

a. Find the names of all employees who have a child who has a birthday in March.
b. Find those employees who took an examination for the skill type "typing" in the city "Dayton".
c. List all skill types in the relation *emp*.

9.2 Redesign the database of Exercise 9.1 into first normal form and fourth normal form. List any functional or multivalued dependencies that you assume. Also list all referential-integrity constraints that should be present in the first- and fourth-normal-form schemas.

9.3 Consider the schemas for the table *people*, and the tables *students* and *teachers*, which were created under *people*, in Section 9.3. Give a relational schema in third normal form that represents the same information. Recall the constraints on subtables, and give all constraints that must be imposed on the relational schema so that every database instance of the relational schema can also be represented by an instance of the schema with inheritance.

9.4 A car-rental company maintains a vehicle database for all vehicles in its current fleet. For all vehicles, it includes the vehicle identification number, license number, manufacturer, model, date of purchase, and color. Special data are included for certain types of vehicles:

- Trucks: cargo capacity
- Sports cars: horsepower, renter age requirement
- Vans: number of passengers
- Off-road vehicles: ground clearance, drivetrain (four- or two-wheel drive)

Construct an SQL:1999 schema definition for this database. Use inheritance where appropriate.

9.5 Explain the distinction between a type x and a reference type **ref**(x). Under what circumstances would you choose to use a reference type?

9.6 Consider the E-R diagram in Figure 2.11, which contains composite, multivalued and derived attributes.

a. Give an SQL:1999 schema definition corresponding to the E-R diagram. Use an array to represent the multivalued attribute, and appropriate SQL:1999 constructs to represent the other attribute types.
b. Give constructors for each of the structured types defined above.

9.7 Give an SQL:1999 schema definition of the E-R diagram in Figure 2.17, which contains specializations.

9.8 Consider the relational schema shown in Figure 3.39.

a. Give a schema definition in SQL:1999 corresponding to the relational schema, but using references to express foreign-key relationships.
b. Write each of the queries given in Exercise 3.10 on the above schema, using SQL:1999.

9.9 Consider an employee database with two relations

employee (*employee-name*, *street*, *city*)
works (*employee-name*, *company-name*, *salary*)

where the primary keys are underlined. Write a query to find companies whose employees earn a higher salary, on average, than the average salary at First Bank Corporation.

a. Using SQL:1999 functions as appropriate.
b. Without using SQL:1999 functions.

9.10 Rewrite the query in Section 9.6.1 that returns the titles of all books that have more than one author, using the **with** clause in place of the function.

9.11 Compare the use of embedded SQL with the use in SQL of functions defined in a general-purpose programming language. Under what circumstances would you use each of these features?

9.12 Suppose that you have been hired as a consultant to choose a database system for your client's application. For each of the following applications, state what type of database system (relational, persistent-programming-language–based OODB, object relational; do not specify a commercial product) you would recommend. Justify your recommendation.

a. A computer-aided design system for a manufacturer of airplanes
b. A system to track contributions made to candidates for public office
c. An information system to support the making of movies

Bibliographical Notes

The nested relational model was introduced in Makinouchi [1977] and Jaeschke and Schek [1982]. Various algebraic query languages are presented in Fischer and Thomas [1983], Zaniolo [1983], Ozsoyoglu et al. [1987], Gucht [1987], and Roth et al. [1988]. The management of null values in nested relations is discussed in Roth et al. [1989]. Design and normalization issues are discussed in Ozsoyoglu and Yuan [1987], Roth and Korth [1987], and Mok et al. [1996]. A collection of papers on nested relations appears in

Several object-oriented extensions to SQL have been proposed. POSTGRES (Stonebraker and Rowe [1986] and Stonebraker [1986a]) was an early implementation of an object-relational system. Illustra was the commercial object-relational system that is the successor of POSTGRES (Illustra was later acquired by Informix, which itself was recently acquired by IBM). The Iris database system from Hewlett-Packard (Fishman et al. [1990] and Wilkinson et al. [1990]) provides object-oriented extensions on top of a relational database system. The O_2 query language described in Bancilhon et al. [1989] is an object-oriented extension of SQL implemented in the O_2 object-oriented database system (Deux [1991]). UniSQL is described in UniSQL [1991]. XSQL is an object-oriented extension of SQL proposed by Kifer et al. [1992].

SQL:1999 was the product of an extensive (and long-delayed) standardization effort, which originally started off as adding object-oriented features to SQL and ended up adding many more features, such as control flow, as we have seen. The official standard documents are available (for a fee) from http://webstore.ansi.org. However, standards documents are very hard to read, and are best left to SQL:1999 implementers. Books on SQL:1999 were still in press at the time of writing this book, see the Web site of the book for current information.

Tools

The Informix database system provides support for many object-relational features. Oracle introduced several object-relational features in Oracle 8.0. Both these systems provided object-relational features before the SQL:1999 standard was finalized, and have some features that are not part of SQL:1999. IBM DB2 supports many of the SQL:1999 features.

CHAPTER 10

XML

Unlike most of the technologies presented in the preceding chapters, the **Extensible Markup Language** (**XML**) was not originally conceived as a database technology. In fact, like the *Hyper-Text Markup Language* (HTML) on which the World Wide Web is based, XML has its roots in document management, and is derived from a language for structuring large documents known as the *Standard Generalized Markup Language* (SGML). However, unlike SGML and HTML, XML can represent database data, as well as many other kinds of structured data used in business applications. It is particularly useful as a data format when an application must communicate with another application, or integrate information from several other applications. When XML is used in these contexts, many database issues arise, including how to organize, manipulate, and query the XML data. In this chapter, we introduce XML and discuss both the management of XML data with database techniques and the exchange of data formatted as XML documents.

10.1 Background

To understand XML, it is important to understand its roots as a document markup language. The term **markup** refers to anything in a document that is not intended to be part of the printed output. For example, a writer creating text that will eventually be typeset in a magazine may want to make notes about how the typesetting should be done. It would be important to type these notes in a way so that they could be distinguished from the actual content, so that a note like "do not break this paragraph" does not end up printed in the magazine. In electronic document processing, a **markup language** is a formal description of what part of the document is content, what part is markup, and what the markup means.

Just as database systems evolved from physical file processing to provide a separate logical view, markup languages evolved from specifying instructions for how to

print parts of the document to specify the *function* of the content. For instance, with functional markup, text representing section headings (for this section, the words "Background") would be marked up as being a section heading, instead of being marked up as text to be printed in large size, bold font. Such functional markup allowed the document to be formatted differently in different situations. It also helps different parts of a large document, or different pages in a large Web site to be formatted in a uniform manner. Functional markup also helps automate extraction of key parts of documents.

For the family of markup languages that includes HTML, SGML, and XML the markup takes the form of **tags** enclosed in angle-brackets, <>. Tags are used in pairs, with <tag> and </tag> delimiting the beginning and the end of the portion of the document to which the tag refers. For example, the title of a document might be marked up as follows.

```
<title>Database System Concepts</title>
```

Unlike HTML, XML does not prescribe the set of tags allowed, and the set may be specialized as needed. This feature is the key to XML's major role in data representation and exchange, whereas HTML is used primarily for document formatting.

For example, in our running banking application, account and customer information can be represented as part of an XML document as in Figure 10.1. Observe the use of tags such as account and account-number. These tags provide context for each value and allow the semantics of the value to be identified.

Compared to storage of data in a database, the XML representation may be inefficient, since tag names are repeated throughout the document. However, in spite of this disadvantage, an XML representation has significant advantages when it is used to exchange data, for example, as part of a message:

- First, the presence of the tags makes the message **self-documenting**; that is, a schema need not be consulted to understand the meaning of the text. We can readily read the fragment above, for example.
- Second, the format of the document is not rigid. For example, if some sender adds additional information, such as a tag last-accessed noting the last date on which an account was accessed, the recipient of the XML data may simply ignore the tag. The ability to recognize and ignore unexpected tags allows the format of the data to evolve over time, without invalidating existing applications.
- Finally, since the XML format is widely accepted, a wide variety of tools are available to assist in its processing, including browser software and database tools.

Just as SQL is the dominant *language* for querying relational data, XML is becoming the dominant *format* for data exchange.

```
<bank>
    <account>
        <account-number> A-101 </account-number>
        <branch-name> Downtown </branch-name>
        <balance> 500 </balance>
    </account>
    <account>
        <account-number> A-102 </account-number>
        <branch-name> Perryridge </branch-name>
        <balance> 400 </balance>
    </account>
    <account>
        <account-number> A-201 </account-number>
        <branch-name> Brighton </branch-name>
        <balance> 900 </balance>
    </account>
    <customer>
        <customer-name> Johnson </customer-name>
        <customer-street> Alma </customer-street>
        <customer-city> Palo Alto </customer-city>
    </customer>
    <customer>
        <customer-name> Hayes </customer-name>
        <customer-street> Main </customer-street>
        <customer-city> Harrison </customer-city>
    </customer>
    <depositor>
        <account-number> A-101 </account-number>
        <customer-name> Johnson </customer-name>
    </depositor>
    <depositor>
        <account-number> A-201 </account-number>
        <customer-name> Johnson </customer-name>
    </depositor>
    <depositor>
        <account-number> A-102 </account-number>
        <customer-name> Hayes </customer-name>
    </depositor>
</bank>
```

Figure 10.1 XML representation of bank information.

10.2 Structure of XML Data

The fundamental construct in an XML document is the **element**. An element is simply a pair of matching start- and end-tags, and all the text that appears between them.

XML documents must have a single **root** element that encompasses all other elements in the document. In the example in Figure 10.1, the <bank> element forms the root element. Further, elements in an XML document must **nest** properly. For instance,

<account> . . . <balance> . . . </balance> . . . </account>

is properly nested, whereas

<account> . . . <balance> . . . </account> . . . </balance>

is not properly nested.

While proper nesting is an intuitive property, we may define it more formally. Text is said to appear **in the context of** an element if it appears between the start-tag and end-tag of that element. Tags are properly nested if every start-tag has a unique matching end-tag that is in the context of the same parent element.

Note that text may be mixed with the subelements of an element, as in Figure 10.2. As with several other features of XML, this freedom makes more sense in a document-processing context than in a data-processing context, and is not particularly useful for representing more structured data such as database content in XML.

The ability to nest elements within other elements provides an alternative way to represent information. Figure 10.3 shows a representation of the bank information from Figure 10.1, but with account elements nested within customer elements. The nested representation makes it easy to find all accounts of a customer, although it would store account elements redundantly if they are owned by multiple customers.

Nested representations are widely used in XML data interchange applications to avoid joins. For instance, a shipping application would store the full address of sender and receiver redundantly on a shipping document associated with each shipment, whereas a normalized representation may require a join of shipping records with a *company-address* relation to get address information.

In addition to elements, XML specifies the notion of an **attribute**. For instance, the type of an account can represented as an attribute, as in Figure 10.4. The attributes of

```
. . .
    <account>
        This account is seldom used any more.
        <account-number> A-102 </account-number>
        <branch-name> Perryridge </branch-name>
        <balance> 400 </balance>
    </account>
. . .
```

Figure 10.2 Mixture of text with subelements.

```
<bank-1>
    <customer>
        <customer-name> Johnson </customer-name>
        <customer-street> Alma </customer-street>
        <customer-city> Palo Alto </customer-city>
        <account>
            <account-number> A-101 </account-number>
            <branch-name> Downtown </branch-name>
            <balance> 500 </balance>
        </account>
        <account>
            <account-number> A-201 </account-number>
            <branch-name> Brighton </branch-name>
            <balance> 900 </balance>
        </account>
    </customer>
    <customer>
        <customer-name> Hayes </customer-name>
        <customer-street> Main </customer-street>
        <customer-city> Harrison </customer-city>
        <account>
            <account-number> A-102 </account-number>
            <branch-name> Perryridge </branch-name>
            <balance> 400 </balance>
        </account>
    </customer>
</bank-1>
```

Figure 10.3 Nested XML representation of bank information.

an element appear as *name=value* pairs before the closing "$>$" of a tag. Attributes are strings, and do not contain markup. Furthermore, attributes can appear only once in a given tag, unlike subelements, which may be repeated.

Note that in a document construction context, the distinction between subelement and attribute is important—an attribute is implicitly text that does not appear in the printed or displayed document. However, in database and data exchange applications of XML, this distinction is less relevant, and the choice of representing data as an attribute or a subelement is frequently arbitrary.

One final syntactic note is that an element of the form <element></element>, which contains no subelements or text, can be abbreviated as <element/>; abbreviated elements may, however, contain attributes.

Since XML documents are designed to be exchanged between applications, a **namespace** mechanism has been introduced to allow organizations to specify globally unique names to be used as element tags in documents. The idea of a namespace is to prepend each tag or attribute with a universal resource identifier (for example, a Web address) Thus, for example, if First Bank wanted to ensure that XML documents

```
...
    <account acct-type= "checking">
        <account-number> A-102 </account-number>
        <branch-name> Perryridge </branch-name>
        <balance> 400 </balance>
    </account>
...
```

Figure 10.4 Use of attributes.

it created would not duplicate tags used by any business partner's XML documents, it can prepend a unique identifier with a colon to each tag name. The bank may use a Web URL such as

http://www.FirstBank.com

as a unique identifier. Using long unique identifiers in every tag would be rather inconvenient, so the namespace standard provides a way to define an abbreviation for identifiers.

In Figure 10.5, the root element (bank) has an attribute xmlns:FB, which declares that FB is defined as an abbreviation for the URL given above. The abbreviation can then be used in various element tags, as illustrated in the figure.

A document can have more than one namespace, declared as part of the root element. Different elements can then be associated with different namespaces. A *default namespace* can be defined, by using the attribute xmlns instead of xmlns:FB in the root element. Elements without an explicit namespace prefix would then belong to the default namespace.

Sometimes we need to store values containing tags without having the tags interpreted as XML tags. So that we can do so, XML allows this construct:

<![CDATA[<account> ···</account>]]>

Because it is enclosed within CDATA, the text <account> is treated as normal text data, not as a tag. The term CDATA stands for character data.

```
<bank xmlns:FB="http://www.FirstBank.com">
    ...
    <FB:branch>
        <FB:branchname> Downtown </FB:branchname>
        <FB:branchcity> Brooklyn </FB:branchcity>
    </FB:branch>
    ...
</bank>
```

Figure 10.5 Unique tag names through the use of namespaces.

10.3 XML Document Schema

Databases have schemas, which are used to constrain what information can be stored in the database and to constrain the data types of the stored information. In contrast, by default, XML documents can be created without any associated schema: An element may then have any subelement or attribute. While such freedom may occasionally be acceptable given the self-describing nature of the data format, it is not generally useful when XML documents must be processesed automatically as part of an application, or even when large amounts of related data are to be formatted in XML.

Here, we describe the document-oriented schema mechanism included as part of the XML standard, the *Document Type Definition*, as well as the more recently defined *XMLSchema*.

10.3.1 Document Type Definition

The **document type definition** (**DTD**) is an optional part of an XML document. The main purpose of a DTD is much like that of a schema: to constrain and type the information present in the document. However, the DTD does not in fact constrain types in the sense of basic types like integer or string. Instead, it only constrains the appearance of subelements and attributes within an element. The DTD is primarily a list of rules for what pattern of subelements appear within an element. Figure 10.6 shows a part of an example DTD for a bank information document; the XML document in Figure 10.1 conforms to this DTD.

Each declaration is in the form of a regular expression for the subelements of an element. Thus, in the DTD in Figure 10.6, a bank element consists of one or more account, customer, or depositor elements; the | operator specifies "or" while the + operator specifies "one or more." Although not shown here, the ∗ operator is used to specify "zero or more," while the ? operator is used to specify an optional element (that is, "zero or one").

```
<!DOCTYPE bank [
    <!ELEMENT bank ( (account—customer—depositor)+)>
    <!ELEMENT account ( account-number branch-name balance )>
    <!ELEMENT customer ( customer-name customer-street customer-city )>
    <!ELEMENT depositor ( customer-name account-number )>
    <!ELEMENT account-number ( #PCDATA )>
    <!ELEMENT branch-name ( #PCDATA )>
    <!ELEMENT balance( #PCDATA )>
    <!ELEMENT customer-name( #PCDATA )>
    <!ELEMENT customer-street( #PCDATA )>
    <!ELEMENT customer-city( #PCDATA )>
] >
```

Figure 10.6 Example of a DTD.

The account element is defined to contain subelements account-number, branch-name and balance (in that order). Similarly, customer and depositor have the attributes in their schema defined as subelements.

Finally, the elements account-number, branch-name, balance, customer-name, customer-street, and customer-city are all declared to be of type #PCDATA. The keyword #PCDATA indicates text data; it derives its name, historically, from "parsed character data." Two other special type declarations are empty, which says that the element has no contents, and any, which says that there is no constraint on the subelements of the element; that is, any elements, even those not mentioned in the DTD, can occur as subelements of the element. The absence of a declaration for an element is equivalent to explicitly declaring the type as any.

The allowable attributes for each element are also declared in the DTD. Unlike subelements, no order is imposed on attributes. Attributes may specified to be of type CDATA, ID, IDREF, or IDREFS; the type CDATA simply says that the attribute contains character data, while the other three are not so simple; they are explained in more detail shortly. For instance, the following line from a DTD specifies that element account has an attribute of type acct-type, with default value checking.

```
<!ATTLIST account acct-type CDATA "checking" >
```

Attributes must have a type declaration and a default declaration. The default declaration can consist of a default value for the attribute or #REQUIRED, meaning that a value must be specified for the attribute in each element, or #IMPLIED, meaning that no default value has been provided. If an attribute has a default value, for every element that does not specify a value for the attribute, the default value is filled in automatically when the XML document is read

An attribute of type ID provides a unique identifier for the element; a value that occurs in an ID attribute of an element must not occur in any other element in the same document. At most one attribute of an element is permitted to be of type ID.

```
<!DOCTYPE bank-2 [
    <!ELEMENT account ( branch, balance )>
    <!ATTLIST account
        account-number ID #REQUIRED
        owners IDREFS #REQUIRED >
    <!ELEMENT customer ( customer-name, customer-street, customer-city )>
    <!ATTLIST customer
        customer-id ID #REQUIRED
        accounts IDREFS #REQUIRED >
    ··· declarations for branch, balance, customer-name,
        customer-street and customer-city ···
] >
```

Figure 10.7 DTD with ID and IDREF attribute types.

An attribute of type IDREF is a reference to an element; the attribute must contain a value that appears in the ID attribute of some element in the document. The type IDREFS allows a list of references, separated by spaces.

Figure 10.7 shows an example DTD in which customer account relationships are represented by ID and IDREFS attributes, instead of depositor records. The account elements use account-number as their identifier attribute; to do so, account-number has been made an attribute of account instead of a subelement. The customer elements have a new identifier attribute called customer-id. Additionally, each customer element contains an attribute accounts, of type IDREFS, which is a list of identifiers of accounts that are owned by the customer. Each account element has an attribute owners, of type IDREFS, which is a list of owners of the account.

Figure 10.8 shows an example XML document based on the DTD in Figure 10.7. Note that we use a different set of accounts and customers from our earlier example, in order to illustrate the IDREFS feature better.

The ID and IDREF attributes serve the same role as reference mechanisms in object-oriented and object-relational databases, permitting the construction of complex data relationships.

```
<bank-2>
    <account account-number="A-401" owners="C100 C102">
        <branch-name> Downtown </branch-name>
        <balance> 500 </balance>
    </account>
    <account account-number="A-402" owners="C102 C101">
        <branch-name> Perryridge </branch-name>
        <balance> 900 </balance>
    </account>
    <customer customer-id="C100" accounts="A-401">
        <customer-name>Joe</customer-name>
        <customer-street> Monroe </customer-street>
        <customer-city> Madison </customer-city>
    </customer>
    <customer customer-id="C101" accounts="A-402">
        <customer-name>Lisa</customer-name>
        <customer-street> Mountain </customer-street>
        <customer-city> Murray Hill </customer-city>
    </customer>
    <customer customer-id="C102" accounts="A-401 A-402">
        <customer-name>Mary</customer-name>
        <customer-street> Erin </customer-street>
        <customer-city> Newark </customer-city>
    </customer>
</bank-2>
```

Figure 10.8 XML data with ID and IDREF attributes.

Document type definitions are strongly connected to the document formatting heritage of XML. Because of this, they are unsuitable in many ways for serving as the type structure of XML for data processing applications. Nevertheless, a tremendous number of data exchange formats are being defined in terms of DTDs, since they were part of the original standard. Here are some of the limitations of DTDs as a schema mechanism.

- Individual text elements and attributes cannot be further typed. For instance, the element balance cannot be constrained to be a positive number. The lack of such constraints is problematic for data processing and exchange applications, which must then contain code to verify the types of elements and attributes.
- It is difficult to use the DTD mechanism to specify unordered sets of subelements. Order is seldom important for data exchange (unlike document layout, where it is crucial). While the combination of alternation (the | operation) and the $*$ operation as in Figure 10.6 permits the specification of unordered collections of tags, it is much more difficult to specify that each tag may only appear once.
- There is a lack of typing in IDs and IDREFs. Thus, there is no way to specify the type of element to which an IDREF or IDREFS attribute should refer. As a result, the DTD in Figure 10.7 does not prevent the "owners" attribute of an account element from referring to other accounts, even though this makes no sense.

10.3.2 XML Schema

An effort to redress many of these DTD deficiencies resulted in a more sophisticated schema language, **XMLSchema**. We present here an example of XMLSchema, and list some areas in which it improves DTDs, without giving full details of XMLSchema's syntax.

Figure 10.9 shows how the DTD in Figure 10.6 can be represented by XMLSchema. The first element is the root element bank, whose type is declared later. The example then defines the types of elements account, customer, and depositor. Observe the use of types xsd:string and xsd:decimal to constrain the types of data elements. Finally the example defines the type BankType as containing zero or more occurrences of each of account, customer and depositor. XMLSchema can define the minimum and maximum number of occurrences of subelements by using minOccurs and maxOccurs. The default for both minimum and maximum occurrences is 1, so these have to be explicity specified to allow zero or more accounts, deposits, and customers.

Among the benefits that XMLSchema offers over DTDs are these:

- It allows user-defined types to be created.
- It allows the text that appears in elements to be constrained to specific types, such as numeric types in specific formats or even more complicated types such as lists or union.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="bank" type="BankType" />
<xsd:element name="account">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="account-number" type="xsd:string"/>
            <xsd:element name="branch-name" type="xsd:string"/>
            <xsd:element name="balance" type="xsd:decimal"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="customer">
            <xsd:element name="customer-number" type="xsd:string"/>
            <xsd:element name="customer-street" type="xsd:string"/>
            <xsd:element name="customer-city" type="xsd:string"/>
</xsd:element>
<xsd:element name="depositor">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="customer-name" type="xsd:string"/>
            <xsd:element name="account-number" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:complexType name="BankType">
    <xsd:sequence>
        <xsd:element ref="account" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="customer" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="depositor" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Figure 10.9 XMLSchema version of DTD from Figure 10.6.

- It allows types to be restricted to create specialized types, for instance by specifying minimum and maximum values.
- It allows complex types to be extended by using a form of inheritance.
- It is a superset of DTDs.
- It allows uniqueness and foreign key constraints.
- It is integrated with namespaces to allow different parts of a document to conform to different schema.
- It is itself specified by XML syntax, as Figure 10.9 shows.

However, the price paid for these features is that XMLSchema is significantly more complicated than DTDs.

10.4 Querying and Transformation

Given the increasing number of applications that use XML to exchange, mediate, and store data, tools for effective management of XML data are becoming increasingly important. In particular, tools for querying and transformation of XML data are essential to extract information from large bodies of XML data, and to convert data between different representations (schemas) in XML. Just as the output of a relational query is a relation, the output of an XML query can be an XML document. As a result, querying and transformation can be combined into a single tool.

Several languages provide increasing degrees of querying and transformation capabilities:

- XPath is a language for path expressions, and is actually a building block for the remaining two query languages.
- XSLT was designed to be a transformation language, as part of the XSL style sheet system, which is used to control the formatting of XML data into HTML or other print or display languages. Although designed for formatting, XSLT can generate XML as output, and can express many interesting queries. Furthermore, it is currently the most widely available language for manipulating XML data.
- XQuery has been proposed as a standard for querying of XML data. XQuery combines features from many of the earlier proposals for querying XML, in particular the language Quilt.

A **tree model** of XML data is used in all these languages. An XML document is modeled as a **tree**, with **nodes** corresponding to elements and attributes. Element nodes can have children nodes, which can be subelements or attributes of the element. Correspondingly, each node (whether attribute or element), other than the root element, has a parent node, which is an element. The order of elements and attributes in the XML document is modeled by the ordering of children of nodes of the tree. The terms parent, child, ancestor, descendant, and siblings are interpreted in the tree model of XML data.

The text content of an element can be modeled as a text node child of the element. Elements containing text broken up by intervening subelements can have multiple text node children. For instance, an element containing "this is a <bold> wonderful </bold> book" would have a subelement child corresponding to the element bold and two text node children corresponding to "this is a" and "book". Since such structures are not commonly used in database data, we shall assume that elements do not contain both text and subelements.

10.4.1 XPath

XPath addresses parts of an XML document by means of path expressions. The language can be viewed as an extension of the simple path expressions in object-oriented and object-relational databases (See Section 9.5.1).

A **path expression** in XPath is a sequence of location steps separated by "/" (instead of the "." operator that separates steps in SQL:1999). The result of a path expression is a set of values. For instance, on the document in Figure 10.8, the XPath expression

```
/bank-2/customer/name
```

would return these elements:

```
<name>Joe</name>
<name>Lisa</name>
<name>Mary</name>
```

The expression

```
/bank-2/customer/name/text()
```

would return the same names, but without the enclosing tags.

Like a directory hierarchy, the initial '/' indicates the root of the document. (Note that this is an abstract root "above" <bank-2> that is the document tag.) Path expressions are evaluated from left to right. As a path expression is evaluated, the result of the path at any point consists of a set of nodes from the document.

When an element name, such as customer, appears before the next '/', it refers to all elements of the specified name that are children of elements in the current element set. Since multiple children can have the same name, the number of nodes in the node set can increase or decrease with each step. Attribute values may also be accessed, using the "@" symbol. For instance, /bank-2/account/@account-number returns a set of all values of account-number attributes of account elements. By default, IDREF links are not followed; we shall see how to deal with IDREFs later.

XPath supports a number of other features:

- Selection predicates may follow any step in a path, and are contained in square brackets. For example,

  ```
  /bank-2/account[balance > 400]
  ```

  returns account elements with a balance value greater than 400, while

  ```
  /bank-2/account[balance > 400]/@account-number
  ```

  returns the account numbers of those accounts.

  We can test the existence of a subelement by listing it without any comparison operation; for instance, if we removed just "> 400" from the above, the

expression would return account numbers of all accounts that have a balance subelement, regardless of its value.

- XPath provides several functions that can be used as part of predicates, including testing the position of the current node in the sibling order and counting the number of nodes matched. For example, the path expression

  /bank-2/account/[customer/count()> 2]

  returns accounts with more than 2 customers. Boolean connectives and and or can be used in predicates, while the function not(. . .) can be used for negation.

- The function id("foo") returns the node (if any) with an attribute of type ID and value "foo". The function id can even be applied on sets of references, or even strings containing multiple references separated by blanks, such as IDREFS. For instance, the path

  /bank-2/account/id(@owner)

  returns all customers referred to from the owners attribute of account elements.

- The | operator allows expression results to be unioned. For example, if the DTD of bank-2 also contained elements for loans, with attribute borrower of type IDREFS identifying loan borrower, the expression

  /bank-2/account/id(@owner) | /bank-2/loan/id(@borrower)

  gives customers with either accounts or loans. However, the | operator cannot be nested inside other operators.

- An XPath expression can skip multiple levels of nodes by using "//". For instance, the expression /bank-2//name finds any name element *anywhere* under the /bank-2 element, regardless of the element in which it is contained. This example illustrates the ability to find required data without full knowledge of the schema.

- Each step in the path need not select from the children of the nodes in the current node set. In fact, this is just one of several directions along which a step in the path may proceed, such as parents, siblings, ancestors and descendants. We omit details, but note that "//", described above, is a short form for specifying "all descendants," while ".." specifies the parent.

10.4.2 XSLT

A **style sheet** is a representation of formatting options for a document, usually stored outside the document itself, so that formatting is separate from content. For example, a style sheet for HTML might specify the font to be used on all headers, and thus

```
<xsl:template match="/bank-2/customer">
    <customer>
    <xsl:value-of select="customer-name"/>
    </customer>
</xsl:template>
<xsl:template match="."/>
```

Figure 10.10 Using XSLT to wrap results in new XML elements.

replace a large number of font declarations in the HTML page. The **XML Stylesheet Language (XSL)** was originally designed for generating HTML from XML, and is thus a logical extension of HTML style sheets. The language includes a general-purpose transformation mechanism, called **XSL Transformations (XSLT)**, which can be used to transform one XML document into another XML document, or to other formats such as HTML.[1] XSLT transformations are quite powerful, and in fact XSLT can even act as a query language.

XSLT transformations are expressed as a series of recursive rules, called **templates**. In their basic form, templates allow selection of nodes in an XML tree by an XPath expression. However, templates can also generate new XML content, so that selection and content generation can be mixed in natural and powerful ways. While XSLT can be used as a query language, its syntax and semantics are quite dissimilar from those of SQL.

A simple template for XSLT consists of a **match** part and a **select** part. Consider this XSLT code:

```
<xsl:template match="/bank-2/customer">
    <xsl:value-of select="customer-name"/>
</xsl:template>
<xsl:template match="."/>
```

The xsl:template match statement contains an XPath expression that selects one or more nodes. The first template matches customer elements that occur as children of the bank-2 root element. The xsl:value-of statement enclosed in the match statement outputs values from the nodes in the result of the XPath expression. The first template outputs the value of the customer-name subelement; note that the value does not contain the element tag.

Note that the second template matches all nodes. This is required because the default behavior of XSLT on subtrees of the input document that do not match any template is to copy the subtrees to the output document.

XSLT copies any tag that is not in the xsl namespace unchanged to the output. Figure 10.10 shows how to use this feature to make each customer name from our example appear as a subelement of a "<customer>" element, by placing the xsl:value-of statement between <customer> and </customer>.

1. The XSL standard now consists of XSLT and a standard for specifying formatting features such as fonts, page margins, and tables. Formatting is not relevant from a database perspective, so we do not cover it here.

```
<xsl:template match="/bank">
    <customers>
    <xsl:apply-templates/>
    </customers>
</xsl:template>
<xsl:template match="/customer">
    <customer>
    <xsl:value-of select="customer-name"/>
    </customer>
</xsl:template>
<xsl:template match="."/>
```

Figure 10.11 Applying rules recursively.

Structural recursion is a key part of XSLT. Recall that elements and subelements naturally form a tree structure. The idea of structural recursion is this: When a template matches an element in the tree structure, XSLT can use structural recursion to apply template rules recursively on subtrees, instead of just outputting a value. It applies rules recursively by the xsl:apply-templates directive, which appears inside other templates.

For example, the results of our previous query can be placed in a surrounding <customers> element by the addition of a rule using xsl:apply-templates, as in Figure 10.11 The new rule matches the outer "bank" tag, and constructs a result document by applying all other templates to the subtrees appearing within the bank element, but wrapping the results in the given <customers> </customers> element. Without recursion forced by the <xsl:apply-templates/> clause, the template would output <customers> </customers>, and then apply the other templates on the subelements.

In fact, the structural recursion is critical to constructing well-formed XML documents, since XML documents must have a single top-level element containing all other elements in the document.

XSLT provides a feature called **keys**, which permit lookup of elements by using values of subelements or attributes; the goals are similar to that of the id() function in XPath, but permits attributes other than the ID attributes to be used. Keys are defined by an xsl:key directive, which has three parts, for example:

```
<xsl:key name="acctno" match="account" use="account-number"/>
```

The name attribute is used to distinguish different keys. The match attribute specifies which nodes the key applies to. Finally, the use attribute specifies the expression to be used as the value of the key. Note that the expression need not be unique to an element; that is, more than one element may have the same expression value. In the example, the key named acctno specifies that the account-number subelement of account should be used as a key for that account.

Keys can be subsequently used in templates as part of any pattern through the key function. This function takes the name of the key and a value, and returns the

```
<xsl:key name="acctno" match="account"use="account-number"/>
<xsl:key name="custno" match="customer" use="customer-name"/>
<xsl:template match="depositor">
    <cust-acct>
    <xsl:value-of select=key("custno", "customer-name")/>
    <xsl:value-of select=key("acctno", "account-number")/>
    </cust-acct>
</xsl:template>
<xsl:template match="."/>
```

Figure 10.12 Joins in XSLT.

set of nodes that match that value. Thus, the XML node for account "A-401" can be referenced as key("acctno", "A-401").

Keys can be used to implement some types of joins, as in Figure 10.12. The code in the figure can be applied to XML data in the format in Figure 10.1. Here, the key function joins the depositor elements with matching customer and account elements. The result of the query consists of pairs of customer and account elements enclosed within cust-acct elements.

XSLT allows nodes to be sorted. A simple example shows how xsl:sort would be used in our style sheet to return customer elements sorted by name:

```
<xsl:template match="/bank">
    <xsl:apply-templates select="customer">
    <xsl:sort select="customer-name"/>
    </xsl:apply-templates>
</xsl:template>
<xsl:template match="customer">
    <customer>
        <xsl:value-of select="customer-name"/>
        <xsl:value-of select="customer-street"/>
        <xsl:value-of select="customer-city"/>
    </customer>
</xsl:template>
<xsl:template match="."/>
```

Here, the xsl:apply-template has a select attribute, which constrains it to be applied only on customer subelements. The xsl:sort directive within the xsl:apply-template element causes nodes to be sorted *before* they are processed by the next set of templates. Options exist to allow sorting on multiple subelements/attributes, by numeric value, and in descending order.

10.4.3 XQuery

The World Wide Web Consortium (W3C) is developing XQuery, a query language for XML. Our discusssion here is based on a draft of the language standard, so the final standard may differ; however we expect the main features we cover here will

not change substantially. The XQuery language derives from an XML query language called Quilt; most of the XQuery features we outline here are part of Quilt. Quilt itself includes features from earlier languages such as XPath, discussed in Section 10.4.1, and two other XML query languages, XQL and XML-QL.

Unlike XSLT, XQuery does not represent queries in XML. Instead, they appear more like SQL queries, and are organized into "FLWR" (pronounced "flower") expressions comprising four sections: **for**, **let**, **where**, and **return**. The **for** section gives a series of variables that range over the results of XPath expressions. When more than one variable is specified, the results include the Cartesian product of the possible values the variables can take, making the **for** clause similar in spirit to the **from** clause of an SQL query. The **let** clause simply allows complicated expressions to be assigned to variable names for simplicity of representation. The **where** section, like the SQL **where** clause, performs additional tests on the joined tuples from the **for** section. Finally, the **return** section allows the construction of results in XML.

A simple FLWR expression that returns the account numbers for checking accounts is based on the XML document of Figure 10.8, which uses ID and IDREFS:

```
for $x in /bank-2/account
let $acctno := $x/@account-number
where $x/balance > 400
return <account-number> $acctno </account-number>
```

Since this query is simple, the **let** clause is not essential, and the variable $acctno in the **return** clause could be replaced with $x/@account-number. Note further that, since the **for** clause uses XPath expressions, selections may occur within the XPath expression. Thus, an equivalent query may have only **for** and **return** clauses:

```
for $x in /bank-2/account[balance > 400]
return <account-number> $x/@account-number </account-number>
```

However, the **let** clause simplifies complex queries.

Path expressions in XQuery may return a multiset, with repeated nodes. The function distinct applied on a multiset, returns a set without duplication. The distinct function can be used even within a **for** clause. XQuery also provides aggregate functions such as sum and count that can be applied on collections such as sets and multisets. While XQuery does not provide a **group by** construct, aggregate queries can be written by using nested FLWR constructs in place of grouping; we leave details as an exercise for you. Note also that variables assigned by **let** clauses may be set- or multiset-valued, if the path expression on the right-hand side returns a set or multiset value.

Joins are specified in XQuery much as they are in SQL. The join of depositor, account and customer elements in Figure 10.1, which we wrote in XSLT in Section 10.4.2, can be written in XQuery this way:

```
for $b in /bank/account,
      $c in /bank/customer,
      $d in /bank/depositor
where $a/account-number = $d/account-number
      and $c/customer-name = $d/customer-name
return <cust-acct> $c $a </cust-acct>
```

The same query can be expressed with the selections specified as XPath selections:

```
for $a in /bank/account,
      $c in /bank/customer,
      $d in /bank/depositor[account-number = $a/account-number
            and customer-name = $c/customer-name]
return <cust-acct> $c $a</cust-acct>
```

XQuery FLWR expressions can be nested in the **return** clause, in order to generate element nestings that do not appear in the source document. This feature is similar to nested subqueries in the **from** clause of SQL queries in Section 9.5.3.

For instance, the XML structure shown in Figure 10.3, with account elements nested within customer elements, can be generated from the structure in Figure 10.1 by this query:

```
<bank-1>
      for $c in /bank/customer
      return
            <customer>
                  $c/*
                  for $d in /bank/depositor[customer-name = $c/customer-name],
                        $a in /bank/account[account-number=$d/account-number]
                  return $a
            </customer>
</bank-1>
```

The query also introduces the syntax $c/*, which refers to all the children of the node, which is bound to the variable $c. Similarly, $c/text() gives the text content of an element, without the tags.

Path expressions in XQuery are based on path expressions in XPath, but XQuery provides some extensions (which may eventually be added to XPath itself). One of the useful syntax extensions is the operator ->, which can be used to dereference IDREFs, just like the function id(). The operator can be applied on a value of type IDREFS to get a set of elements. It can be used, for example, to find all the accounts associated with a customer, with the ID/IDREFS representation of bank information. We leave details to the reader.

Results can be sorted in XQuery if a **sortby** clause is included at the end of any expression; the clause specifies how the instances of that expression should be sorted. For instance, this query outputs all customer elements sorted by the name subelement:

```
        for $c in /bank/customer,
        return <customer> $c/* </customer> sortby(name)
```

To sort in descending order, we can use **sortby**(name **descending**).

Sorting can be done at multiple levels of nesting. For instance, we can get a nested representation of bank information sorted in customer name order, with accounts of each customer sorted by account number, as follows.

```
<bank-1>
    for $c in /bank/customer
    return
        <customer>
            $c/*
            for $d in /bank/depositor[customer-name = $c/customer-name],
                $a in /bank/account[account-number=$d/account-number]
            return <account> $a/* </account> sortby(account-number)
        </customer> sortby(customer-name)
</bank-1>
```

XQuery provides a variety of built-in functions, and supports user-defined functions. For instance, the built-in function document(name) returns the root of a named document; the root can then be used in a path expression to access the contents of the document. Users can define functions as illustrated by this function, which returns a list of all balances of a customer with a specified name:

```
function balances(xsd:string $c) returns list(xsd:numeric) {
    for $d in /bank/depositor[customer-name = $c],
        $a in /bank/account[account-number=$d/account-number]
    return $a/balance
}
```

XQuery uses the type system of XMLSchema. XQuery also provides functions to convert between types. For instance, number(x) converts a string to a number.

XQuery offers a variety of other features, such as if-then-else clauses, which can be used within **return** clauses, and existential and universal quantification, which can be used in predicates in **where** clauses. For example, existential quantification can be expressed using **some** $e **in** path **satisfies** P where path is a path expression, and P is a predicate which can use $e. Universal quantification can be expressed by using **every** in place of **some**.

10.5 The Application Program Interface

With the wide acceptance of XML as a data representation and exchange format, software tools are widely available for manipulation of XML data. In fact, there are two standard models for programmatic manipulation of XML, each available for use with a wide variety of popular programming languages.

One of the standard APIs for manipulating XML is the *document object model* (DOM), which treats XML content as a tree, with each element represented by a node, called a DOMNode. Programs may access parts of the document in a navigational fashion, beginning with the root.

DOM libraries are available for most common programming langauges and are even present in Web browsers, where it may be used to manipulate the document displayed to the user. We outline here some of the interfaces and methods in the Java API for DOM, to give a flavor of DOM. The Java DOM API provides an interface called Node, and interfaces Element and Attribute, which inherit from the Node interface. The Node interface provides methods such as getParentNode(), getFirstChild(), and getNextSibling(), to navigate the DOM tree, starting with the root node. Subelements of an element can be accessed by name getElementsByTagName(name), which returns a list of all child elements with a specified tag name; individual members of the list can be accessed by the method item(i), which returns the ith element in the list. Attribute values of an element can be accessed by name, using the method getAttribute(name). The text value of an element is modeled as a Text node, which is a child of the element node; an element node with no subelements has only one such child node. The method getData() on the Text node returns the text contents. DOM also provides a variety of functions for updating the document by adding and deleting attribute and element children of a node, setting node values, and so on.

Many more details are required for writing an actual DOM program; see the bibliographical notes for references to further information.

DOM can be used to access XML data stored in databases, and an XML database can be built using DOM as its primary interface for accessing and modifying data. However, the DOM interface does not support any form of declarative querying.

The second programming interface we discuss, the *Simple API for XML* (SAX) is an *event* model, designed to provide a common interface between parsers and applications. This API is built on the notion of *event handlers*, which consists of user-specified functions associated with parsing events. Parsing events correspond to the recognition of parts of a document; for example, an event is generated when the start-tag is found for an element, and another event is generated when the end-tag is found. The pieces of a document are always encountered in order from start to finish. SAX is not appropriate for database applications.

10.6 Storage of XML Data

Many applications require storage of XML data. One way to store XML data is to convert it to relational representation, and store it in a relational database. There are several alternatives for storing XML data, briefly outlined here.

10.6.1 Relational Databases

Since relational databases are widely used in existing applications, there is a great benefit to be had in storing XML data in relational databases, so that the data can be accessed from existing applications.

Converting XML data to relational form is usually straightforward if the data were generated from a relational schema in the first place, and XML was used merely as a data exchange format for relational data. However, there are many applications where the XML data is not generated from a relational schema, and translating the data to relational form for storage may not be straightforward. In particular, nested elements and elements that recur (corresponding to set valued attributes) complicate storage of XML data in relational format. Several alternative approaches are available:

- **Store as string.** A simple way to store XML data in a relational database is to store each child element of the top-level element as a string in a separate tuple in the database. For instance, the XML data in Figure 10.1 could be stored as a set of tuples in a relation *elements*(*data*), with the attribute *data* of each tuple storing one XML element (account, customer, or depositor) in string form.

  While the above representation is easy to use, the database system does not know the schema of the stored elements. As a result, it is not possible to query the data directly. In fact, it is not even possible to implement simple selections such as finding all account elements, or finding the account element with account number A-401, without scanning all tuples of the relation and examining the contents of the string stored in the tuple.

  A partial solution to this problem is to store different types of elements in different relations, and also store the values of some critical elements as attributes of the relation to enable indexing. For instance, in our example, the relations would be *account-elements*, *customer-elements*, and *depositor-elements*, each with an attribute *data*. Each relation may have extra attributes to store the values of some subelements, such as *account-number* or *customer-name*. Thus, a query that requires account elements with a specified account number can be answered efficiently with this representation. Such an approach depends on type information about XML data, such as the DTD of the data.

  Some database systems, such as Oracle 9, support **function indices**, which can help avoid replication of attributes between the XML string and relation attributes. Unlike normal indices, which are on attribute values, function indices can be built on the result of applying user-defined functions on tuples. For instance, a function index can be built on a user-defined function that returns the value of the account-number subelement of the XML string in a tuple. The index can then be used in the same way as an index on a *account-number* attribute.

  The above approaches have the drawback that a large part of the XML information is stored within strings. It is possible to store all the information in relations in one of several ways which we examine next.

- **Tree representation.** Arbitrary XML data can be modeled as a tree and stored using a pair of relations:

  nodes(*id*, *type*, *label*, *value*)
  child(*child-id*, *parent-id*)

Each element and attribute in the XML data is given a unique identifier. A tuple inserted in the *nodes* relation for each element and attribute with its identifier (*id*), its type (attribute or element), the name of the element or attribute (*label*), and the text value of the element or attribute (*value*). The relation *child* is used to record the parent element of each element and attribute. If order information of elements and attributes must be preserved, an extra attribute *position* can be added to the *child* relation to indicate the relative position of the child among the children of the parent. As an exercise, you can represent the XML data of Figure 10.1 by using this technique.

This representation has the advantage that all XML information can be represented directly in relational form, and many XML queries can be translated into relational queries and executed inside the database system. However, it has the drawback that each element gets broken up into many pieces, and a large number of joins are required to reassemble elements.

- **Map to relations.** In this approach, XML elements whose schema is known are mapped to relations and attributes. Elements whose schema is unknown are stored as strings, or as a tree representation.

  A relation is created for each element type whose schema is known. All attributes of these elements are stored as attributes of the relation. All subelements that occur at most once inside these element (as specified in the DTD) can also be represented as attributes of the relation; if the subelement can contain only text, the attribute stores the text value. Otherwise, the relation corresponding to the subelement stores the contents of the subelement, along with an identifier for the parent type and the attribute stores the identifier of the subelement. If the subelement has further nested subelements, the same procedure is applied to the subelement.

  If a subelement can occur multiple times in an element, the map-to-relations approach stores the contents of the subelements in the relation corresponding to the subelement. It gives both parent and subelement unique identifiers, and creates a separate relation, similar to the child relation we saw earlier in the tree representation, to identify which subelement occurs under which parent.

  Note that when we apply this appoach to the DTD of the data in Figure 10.1, we get back the original relational schema that we have used in earlier chapters. The bibliographical notes provide references to such hybrid approaches.

10.6.2 Nonrelational Data Stores

There are several alternatives for storing XML data in nonrelational data storage systems:

- **Store in flat files.** Since XML is primarily a file format, a natural storage mechanism is simply a flat file. This approach has many of the drawbacks, outlined in Chapter 1, of using file systems as the basis for database applications. In particular, it lacks data isolation, integrity checks, atomicity, concurrent access, and security. However, the wide availability of XML tools that work on

file data makes it relatively easy to access and query XML data stored in files. Thus, this storage format may be sufficient for some applications.

- **Store in an XML Database.** XML databases are databases that use XML as their basic data model. Early XML databases implemented the Document Object Model on a C++-based object-oriented database. This allows much of the object-oriented database infrastructure to be reused, while using a standard XML interface. The addition of an XML query language provides declarative querying. It is also possible to build XML databases as a layer on top of relational databases.

10.7 XML Applications

A central design goal for XML is to make it easier to communicate information, on the Web and between applications, by allowing the semantics of the data to be described with the data itself. Thus, while the large amount of XML data and its use in business applications will undoubtably require and benefit from database technologies, XML is foremost a means of communication. Two applications of XML for communication —exchange of data, and mediation of Web information resources—illustrate how XML achieves its goal of supporting data exchange and demonstrate how database technology and interaction are key in supporting exchange-based applications.

10.7.1 Exchange of Data

Standards are being developed for XML representation of data for a variety of specialized applications ranging from business applications such as banking and shipping to scientific applications such as chemistry and molecular biology. Some examples:

- The chemical industry needs information about chemicals, such as their molecular structure, and a variety of important properties such as boiling and melting points, calorific values, solubility in various solvents, and so on. *ChemML* is a standard for representing such information.
- In shipping, carriers of goods and customs and tax officials need shipment records containing detailed information about the goods being shipped, from whom and to where they were sent, to whom and to where they are being shipped, the monetary value of the goods, and so on.
- An online marketplace in which business can buy and sell goods (a so-called business-to-business B2B market) requires information such as product catalogs, including detailed product descriptions and price information, product inventories, offers to buy, and quotes for a proposed sale.

Using normalized relational schemas to model such complex data requirements results in a large number of relations, which is often hard for users to manage. The relations often have large numbers of attributes; explicit representation of attribute/-element names along with values in XML helps avoid confusion between attributes. Nested element representations help reduce the number of relations that must be

represented, as well as the number of joins required to get required information, at the possible cost of redundancy. For instance, in our bank example, listing customers with account elements nested within account elements, as in Figure 10.3, results in a format that is more natural for some applications, in particular for humans to read, than is the normalized representation in Figure 10.1.

When XML is used to exchange data between business applications, the data most often originate in relational databases. Data in relational databases must be *published*, that is, converted to XML form, for export to other applications. Incoming data must be *shredded*, that is, converted back from XML to normalized relation form and stored in a relational database. While application code can perform the publishing and shredding operations, the operations are so common that the conversions should be done automatically, without writing application code, where possible. Database vendors are therefore working to *XML-enable* their database products.

An XML-enabled database supports an automatic mapping from its internal model (relational, object-relational or object-oriented) to XML. These mappings may be simple or complex. A simple mapping might assign an element to every row of a table, and make each column in that row either an attribute or a subelement of the row's element. Such a mapping is straightforward to generate automatically. A more complicated mapping would allow nested structures to be created. Extensions of SQL with nested queries in the **select** clause have been developed to allow easy creation of nested XML output. Some database products also allow XML queries to access relational data by treating the XML form of relational data as a *virtual* XML document.

10.7.1.1 Data Mediation

Comparison shopping is an example of a mediation application, in which data about items, inventory, pricing, and shipping costs are extracted from a variety of Web sites offering a particular item for sale. The resulting aggregated information is significantly more valuable than the individual information offered by a single site.

A personal financial manager is a similar application in the context of banking. Consider a consumer with a variety of accounts to manage, such as bank accounts, savings accounts, and retirement accounts. Suppose that these accounts may be held at different institutions. Providing centralized management for all accounts of a customer is a major challenge. XML-based mediation addresses the problem by extracting an XML representation of account information from the respective Web sites of the financial institutions where the individual holds accounts. This information may be extracted easily if the institution exports it in a standard XML format, and undoubtedly some will. For those that do not, *wrapper* software is used to generate XML data from HTML Web pages returned by the Web site. Wrapper applications need constant maintenance, since they depend on formatting details of Web pages, which change often. Nevertheless, the value provided by mediation often justifies the effort required to develop and maintain wrappers.

Once the basic tools are available to extract information from each source, a *mediator* application is used to combine the extracted information under a single schema. This may require further transformation of the XML data from each site, since different sites may structure the same information differently. For instance, one of the

banks may export information in the format in Figure 10.1, while another may use the nested format in Figure 10.3. They may also use different names for the same information (for instance, acct-number and account-id), or may even use the same name for different information. The mediator must decide on a single schema that represents all required information, and must provide code to transform data between different representations. Such issues are discussed in more detail in Section 19.8, in the context of distributed databases. XML query languages such as XSLT and XQuery play an important role in the task of transformation between different XML representations.

10.8 Summary

- Like the Hyper-Text Markup Language, HTML, on which the Web is based, the Extensible Markup Language, XML, is a descendant of the Standard Generalized Markup Language (SGML). XML was originally intended for providing functional markup for Web documents, but has now become the defacto standard data format for data exchange between applications.
- XML documents contain elements, with matching starting and ending tags indicating the beginning and end of an element. Elements may have subelements nested within them, to any level of nesting. Elements may also have attributes. The choice between representing information as attributes and subelements is often arbitrary in the context of data representation.
- Elements may have an attribute of type ID that stores a unique identifier for the element. Elements may also store references to other elements using attributes of type IDREF. Attributes of type IDREFS can store a list of references.
- Documents may optionally have their schema specified by a Document Type Declaration, DTD. The DTD of a document specifies what elements may occur, how they may be nested, and what attributes each element may have.
- Although DTDs are widely used, they have several limitations. For instance, they do not provide a type system. XMLSchema is a new standard for specifying the schema of a document. While it provides more expressive power, including a powerful type system, it is also more complicated.
- XML data can be represented as tree structures, with nodes corresponding to elements and attributes. Nesting of elements is reflected by the parent-child structure of the tree representation.
- Path expressions can be used to traverse the XML tree structure, to locate required data. XPath is a standard language for path expressions, and allows required elements to be specified by a file-system-like path, and additionally allows selections and other features. XPath also forms part of other XML query languages.
- The XSLT language was originally designed as the transformation language for a style sheet facility, in other words, to apply formatting information to

XML documents. However, XSLT offers quite powerful querying and transformation features and is widely available, so it is used for quering XML data.

- XSLT programs contain a series of templates, each with a **match** part and a **select** part. Each element in the input XML data is matched against available templates, and the select part of the first matching template is applied to the element.

  Templates can be applied recursively, from within the body of another template, a procedure known as structural recursion. XSLT supports keys, which can be used to implement some types of joins. It also supports sorting and other querying facilities.

- The XQuery language, which is currently being standardized, is based on the Quilt query language. The XQuery language is similar to SQL, with **for**, **let**, **where**, and **return** clauses.

  However, it supports many extensions to deal with the tree nature of XML and to allow for the transformation of XML documents into other documents with a significantly different structure.

- XML data can be stored in any of several different ways. For example, XML data can be stored as strings in a relational database. Alternatively, relations can represent XML data as trees. As another alternative, XML data can be mapped to relations in the same way that E-R schemas are mapped to relational schemas.

  XML data may also be stored in file systems, or in XML-databases, which use XML as their internal representation.

- The ability to transform documents in languages such as XSLT and XQuery is a key to the use of XML in mediation applications, such as electronic business exchanges and the extraction and combination of Web data for use by a personal finance manager or comparison shopper.

Review Terms

- Extensible Markup Language (XML)
- Hyper-Text Markup Language (HTML)
- Standard Generalized Markup Language
- Markup language
- Tags
- Self-documenting
- Element
- Root element
- Nested elements
- Attribute
- Namespace
- Default namespace
- Schema definition
  - Document Type Definition (DTD)
  - XMLSchema
- ID
- IDREF and IDREFS
- Tree model of XML data

- Nodes
- Querying and transformation
- Path expressions
- XPath
- Style sheet
- XML Style sheet Language (XSL)
- XSL Transformations (XSLT)
  - Templates
    - Match
    - Select
  - Structural recursion
  - Keys
  - Sorting
- XQuery
  - FLWR expressions
    - **for**
    - **let**
    - **where**
    - **return**
  - Joins
  - Nested FLWR expression
  - Sorting
- XML API
- Document Object Model (DOM)
- Simple API for XML (SAX)
- Storage of XML data
  - In relational databases
    - Store as string
    - Tree representation
    - Map to relations
  - In nonrelational data stores
    - Files
    - XML-databases
- XML Applications
  - Exchange of data
    - Publish and shred
  - Data mediation
    - Wrapper software
- XML-Enabled database

Exercises

10.1 Give an alternative representation of bank information containing the same data as in Figure 10.1, but using attributes instead of subelements. Also give the DTD for this representation.

10.2 Show, by giving a DTD, how to represent the *books* nested-relation from Section 9.1, using XML.

10.3 Give the DTD for an XML representation of the following nested-relational schema

Emp = (ename, ChildrenSet **setof***(Children), SkillsSet* **setof***(Skills))*
Children = (name, Birthday)
Birthday = (day, month, year)
Skills = (type, ExamsSet **setof***(Exams))*
Exams = (year, city)

10.4 Write the following queries in XQuery, assuming the DTD from Exercise 10.3.

a. Find the names of all employees who have a child who has a birthday in March.

b. Find those employees who took an examination for the skill type "typing" in the city "Dayton".

c. List all skill types in *Emp*.

```
<!DOCTYPE bibliography [
    <!ELEMENT book (title, author+, year, publisher, place?)>
    <!ELEMENT article (title, author+, journal, year, number, volume, pages?)>
    <!ELEMENT author ( last-name, first-name) >
    <!ELEMENT title ( #PCDATA )>
    · · · similar PCDATA declarations for year, publisher, place, journal, year,
        number, volume, pages, last-name and first-name
] >
```

Figure 10.13 DTD for bibliographical data.

10.5 Write queries in XSLT and in XPath on the DTD of Exercise 10.3 to list all skill types in *Emp*.

10.6 Write a query in XQuery on the XML representation in Figure 10.1 to find the total balance, across all accounts, at each branch. (Hint: Use a nested query to get the effect of an SQL **group by**.)

10.7 Write a query in XQuery on the XML representation in Figure 10.1 to compute the left outer join of customer elements with account elements. (Hint: Use universal quantification.)

10.8 Give a query in XQuery to flip the nesting of data from Exercise 10.2. That is, at the outermost level of nesting the output must have elements corresponding to authors, and each such element must have nested within it items corresponding to all the books written by the author.

10.9 Give the DTD for an XML representation of the information in Figure 2.29. Create a separate element type to represent each relationship, but use ID and IDREF to implement primary and foreign keys.

10.10 Write queries in XSLT and XQuery to output customer elements with associated account elements nested within the customer elements, given the bank information representation using ID and IDREFS in Figure 10.8.

10.11 Give a relational schema to represent bibliographical information specified as per the DTD fragment in Figure 10.13. The relational schema must keep track of the order of author elements. You can assume that only books and articles appear as top level elements in XML documents.

10.12 Consider Exercise 10.11, and suppose that authors could also appear as top level elements. What change would have to be done to the relational schema.

10.13 Write queries in XQuery on the bibliography DTD fragment in Figure 10.13, to do the following.

a. Find all authors who have authored a book and an article in the same year.
b. Display books and articles sorted by year.
c. Display books with more than one author.

10.14 Show the tree representation of the XML data in Figure 10.1, and the representation of the tree using *nodes* and *child* relations described in Section 10.6.1.

10.15 Consider the following recursive DTD.

```
<!DOCTYPE parts [
    <!ELEMENT part (name, subpartinfo*)>
    <!ELEMENT subpartinfo (part, quantity)>
    <!ELEMENT name ( #PCDATA )>
    <!ELEMENT quantity ( #PCDATA )>
] >
```

a. Give a small example of data corresponding to the above DTD.

b. Show how to map this DTD to a relational schema. You can assume that part names are unique, that is, whereever a part appears, its subpart structure will be the same.

Bibliographical Notes

The XML Cover Pages site (www.oasis-open.org/cover/) contains a wealth of XML information, including tutorial introductions to XML, standards, publications, and software. The World Wide Web Consortium (W3C) acts as the standards body for Web-related standards, including basic XML and all the XML-related languages such as XPath, XSLT and XQuery. A large number of technical reports defining the XML related standards are available at www.w3c.org.

Fernandez et al. [2000] gives an algebra for XML. Quilt is described in Chamberlin et al. [2000]. Sahuguet [2001] describes a system, based on the Quilt language, for querying XML. Deutsch et al. [1999b] describes the XML-QL language. Integration of keyword querying into XML is outlined by Florescu et al. [2000]. Query optimization for XML is described in McHugh and Widom [1999]. Fernandez and Morishima [2001] describe efficient evaluation of XML queries in middleware systems. Other work on querying and manipulating XML data includes Chawathe [1999], Deutsch et al. [1999a], and Shanmugasundaram et al. [2000].

Florescu and Kossmann [1999], Kanne and Moerkotte [2000], and Shanmugasundaram et al. [1999] describe storage of XML data. Schning [2001] describes a database designed for XML. XML support in commercial databases is described in Banerjee et al. [2000], Cheng and Xu [2000] and Rys [2001]. See Chapters 25 through 27 for more information on XML support in commercial databases. The use of XML for data integration is described by Liu et al. [2000], Draper et al. [2001], Baru et al. [1999], and Carey et al. [2000].

Tools

A number of tools to deal with XML are available in the public domain. The site www.oasis-open.org/cover/ contains links to a variety of software tools for XML and XSL (including XSLT). Kweelt (available at http://db.cis.upenn.edu/Kweelt/) is a publicly available XML querying system based on the Quilt language.