10.5.3 Generator and Mutator Functions

In order to create data that conforms to a UDT, or to change components of objects with a UDT, we can use two kinds of methods that are created automatically, along with the observer methods, whenever a UDT is defined. These are:

1. A *generator method.* This method has the name of the type and no argument. It may be invoked without being applied to any object. That is, if T is a UDT, then $T()$ returns an object of type T, with no values in its various components.

2. *Mutator methods.* For each attribute x of UDT T, there is a mutator method $x(v)$. When applied to an object of type T, it changes the x attribute of that object to have value v. Notice that the mutator and observer method for an attribute each have the name of the attribute, but differ in that the mutator has an argument.

Example 10.24: We shall write a PSM procedure that takes as arguments a street, a city, and a name, and inserts into the relation `MovieStar` (of type `StarType` according to Example 10.17) an object constructed from these values, using calls to the proper generator and mutator functions. Recall from Example 10.14 that objects of `StarType` have a `name` component that is a character string, but an `address` component that is itself an object of type `AddressType`. The procedure `InsertStar` is shown in Fig. 10.22.

```
 1)  CREATE PROCEDURE InsertStar(
 2)      IN s CHAR(50),
 3)      IN c CHAR(20),
 4)      IN n CHAR(30)
     )
 5)  DECLARE newAddr AddressType;
 6)  DECLARE newStar StarType;

     BEGIN
 7)      SET newAddr = AddressType();
 8)      SET newStar = StarType();
 9)      newAddr.street(s);
10)      newAddr.city(c);
11)      newStar.name(n);
12)      newStar.address(newAddr);
13)      INSERT INTO MovieStar VALUES(newStar);
     END;
```

Figure 10.22: Creating and storing a `StarType` object

Lines (2) through (4) introduce the arguments s, c, and n, which will provide values for a street, city, and star name, respectively. Lines (5) and (6) declare two local variables. Each is of one of the UDT's involved in the type for objects that exist in the relation `MovieStar`. At lines (7) and (8) we create empty objects of each of these two types.

Lines (9) and (10) put real values in the object `newAddr`; these values are taken from the procedure arguments that provide a street and a city. Line (11) similarly installs the argument n as the value of the `name` component in the object `newStar`. Then line (12) takes the entire `newAddr` object and makes it the value of the `address` component in `newStar`. Finally, line (13) inserts the constructed object into relation `MovieStar`. Notice that, as always, a relation that has a UDT as its type has but a single component, even if that component has several attributes, such as `name` and `address` in this example.

To insert a star into `MovieStar`, we can call procedure `InsertStar`.

```
CALL InsertStar('345 Spruce St.', 'Glendale', 'Gwyneth Paltrow');
```

is an example. □

It is much simpler to insert objects into a relation with a UDT if your DBMS provides a generator function that takes values for the attributes of the UDT and returns a suitable object. For example, if we have functions `AddressType(s,c)` and `StarType(n,a)` that return objects of the indicated types, then we can make the insertion at the end of Example 10.24 with an `INSERT` statement of a familiar form:

```
INSERT INTO MovieStar VALUES(
    StarType('Gwyneth Paltrow',
        AddressType('345 Spruce St.', 'Glendale')));
```

10.5.4 Ordering Relationships on UDT's

Objects that are of some UDT are inherently abstract, in the sense that there is no way to compare two objects of the same UDT, either to test whether they are "equal" or whether one is less than another. Even two objects that have all components identical will not be considered equal unless we tell the system to regard them as equal. Similarly, there is no obvious way to sort the tuples of a relation that has a UDT unless we define a function that tells which of two objects of that UDT precedes the other.

Yet there are many SQL operations that require either an equality test or both an equality and a "less than" test. For instance, we cannot eliminate duplicates if we can't tell whether two tuples are equal. We cannot group by an attribute whose type is a UDT unless there is an equality test for that UDT. We cannot use an `ORDER BY` clause or a comparison like $<$ in a `WHERE` clause unless we can compare two elements.

To specify an ordering or comparison, SQL allows us to issue a `CREATE ORDERING` statement for any UDT. There are a number of forms this statement may take, and we shall only consider the two simplest options:

1. The statement

   `CREATE ORDERING FOR` T `EQUALS ONLY BY STATE;`

   says that two members of UDT T are considered equal if all of their corresponding components are equal. There is no $<$ defined on objects of UDT T.

2. The following statement

   `CREATE ORDERING FOR` T
   `ORDERING FULL BY RELATIVE WITH` F`;`

   says that any of the six comparisons (<, <=, >, >=, =, and <>) may be performed on objects of UDT T. To tell how objects x_1 and x_2 compare, we apply the function F to these objects. This function must be written so that $F(x_1, x_2) < 0$ whenever we want to conclude that $x_1 < x_2$; $F(x_1, x_2) = 0$ means that $x_1 = x_2$, and $F(x_1, x_2) > 0$ means that $x_1 > x_2$. If we replace "`ORDERING FULL`" with "`EQUALS ONLY`," then $F(x_1, x_2) = 0$ indicates that $x_1 = x_2$, while any other value of $F(x_1, x_2)$ means that $x_1 \neq x_2$. Comparison by $<$ is impossible in this case.

Example 10.25: Let us consider a possible ordering on the UDT `StarType` from Example 10.14. If we want only an equality on objects of this UDT, we could declare:

```
CREATE ORDERING FOR StarType EQUALS ONLY BY STATE;
```

That statement says that two objects of `StarType` are equal if and only if their names are equal as character strings, and their addresses are equal as objects of UDT `AddressType`.

The problem is that, unless we define an ordering for `AddressType`, an object of that type is not even equal to itself. Thus, we also need to create at least an equality test for `AddressType`. A simple way to do so is to declare that two `AddressType` objects are equal if and only if their streets and cities are each equal as strings. We could do so by:

```
CREATE ORDERING FOR AddressType EQUALS ONLY BY STATE;
```

Alternatively, we could define a complete ordering of `AddressType` objects. One reasonable ordering is to order addresses first by cities, alphabetically, and among addresses in the same city, by street address, alphabetically. To do so, we have to define a function, say `AddrLEG`, that takes two `AddressType` arguments and returns a negative, zero, or positive value to indicate that the first is less than, equal to, or greater than the second. We declare:

```
CREATE ORDERING FOR AddressType
ORDERING FULL BY RELATIVE WITH AddrLEG;
```

The function `AddrLEG` is shown in Fig. 10.23. Notice that if we reach line (7), it must be that the two `city` components are the same, so we compare the `street` components. Likewise, if we reach line (9), the only remaining possibility is that the cities are the same and the first street precedes the second alphabetically. □

```
1)  CREATE FUNCTION AddrLEG(
2)      x1 AddressType,
3)      x2 AddressType
4)  ) RETURNS INTEGER

5)  IF x1.city() < x2.city() THEN RETURN(-1)
6)  ELSEIF x1.city() > x2.city() THEN RETURN(1)
7)  ELSEIF x1.street() < x2.street() THEN RETURN(-1)
8)  ELSEIF x1.street() = x2.street() THEN RETURN(0)
9)  ELSE RETURN(1)
    END IF;
```

Figure 10.23: A comparison function for address objects

In practice, commercial DBMS's each have their own way of allowing the user to define comparisons for a UDT. In addition to the two approaches mentioned above, some of the capabilities offered are:

a) *Strict Object Equality.* Two objects are equal if and only if they are the same object.

b) *Method-Defined Equality.* A function is applied to two objects and returns true or false, depending on whether or not the two objects should be considered equal.

c) *Method-Defined Mapping.* A function is applied to one object and returns a real number. Objects are compared by comparing the real numbers returned.

10.5.5 Exercises for Section 10.5

Exercise 10.5.1: Use the `StarsIn` relation of Example 10.20 and the `Movies` and `MovieStar` relations accessible through `StarsIn` to write the following queries:

a) Find the names of the stars of *Dogma*.

! b) Find the titles and years of all movies in which at least one star lives in Malibu.

c) Find all the movies (objects of type `MovieType`) that starred Melanie Griffith.

! d) Find the movies (title and year) with at least five stars.

Exercise 10.5.2: Using your schema from Exercise 10.4.3, write the following queries. Don't forget to use references whenever appropriate.

a) Find the manufacturers of PC's with a hard disk larger than 60 gigabytes.

b) Find the manufacturers of laser printers.

! c) Produce a table giving for each model of laptop, the model of the laptop having the highest processor speed of any laptop made by the same manufacturer.

Exercise 10.5.3: Using your schema from Exercise 10.4.5, write the following queries. Don't forget to use references whenever appropriate and avoid joins (i.e., subqueries or more than one tuple variable in the `FROM` clause).

a) Find the ships with a displacement of more than 35,000 tons.

b) Find the battles in which at least one ship was sunk.

! c) Find the classes that had ships launched after 1930.

!! d) Find the battles in which at least one US ship was damaged.

Exercise 10.5.4: Assuming the function `AddrLEG` of Fig. 10.23 is available, write a suitable function to compare objects of type `StarType`, and declare your function to be the basis of the ordering of `StarType` objects.

! **Exercise 10.5.5:** Write a procedure to take a star name as argument and delete from `StarsIn` and `MovieStar` all tuples involving that star.

10.6 On-Line Analytic Processing

An important application of databases is examination of data for patterns or trends. This activity, called *OLAP* (standing for *On-Line Analytic Processing* and pronounced "oh-lap"), generally involves highly complex queries that use one or more aggregations. These queries are often termed *OLAP queries* or *decision-support queries*. Some examples will be given in Section 10.6.2. A typical example is for a company to search for those of its products that have markedly increasing or decreasing overall sales.

Decision-support queries typically examine very large amounts of data, even if the query results are small. In contrast, common database operations, such

as bank deposits or airline reservations, each touch only a tiny portion of the database; the latter type of operation is often referred to as *OLTP* (*On-Line Transaction Processing*, spoken "oh-ell-tee-pee").

A recent trend in DBMS's is to provide specialized support for OLAP queries. For example, systems often support a "data cube" in some way. We shall discuss the architecture of these systems in Section 10.7.

10.6.1 OLAP and Data Warehouses

It is common for OLAP applications to take place in a separate copy of the master database, called a *data warehouse*. Data from many separate databases may be integrated into the warehouse. In a common scenario, the warehouse is only updated overnight, while the analysts work on a frozen copy during the day. The warehouse data thus gets out of date by as much as 24 hours, which limits the timeliness of its answers to OLAP queries, but the delay is tolerable in many decision-support applications.

There are several reasons why data warehouses play an important role in OLAP applications. First, the warehouse may be necessary to organize and centralize data in a way that supports OLAP queries; the data may initially be scattered across many different databases. But often more important is the fact that OLAP queries, being complex and touching much of the data, take too much time to be executed in a transaction-processing system with high throughput requirements. Recall the discussion of serializable transactions in Section 6.6. Trying to run a long transaction that needed to touch much of the database serializably with other transactions would stall ordinary OLTP operations more than could be tolerated. For instance, recording new sales as they occur might not be permitted if there were a concurrent OLAP query computing average sales.

10.6.2 OLAP Applications

A common OLAP application uses a warehouse of sales data. Major store chains will accumulate terabytes of information representing every sale of every item at every store. Queries that aggregate sales into groups and identify significant groups can be of great use to the company in predicting future problems and opportunities.

Example 10.26: Suppose the Aardvark Automobile Co. builds a data warehouse to analyze sales of its cars. The schema for the warehouse might be:

```
Sales(serialNo, date, dealer, price)
Autos(serialNo, model, color)
Dealers(name, city, state, phone)
```

A typical decision-support query might examine sales on or after April 1, 2006 to see how the recent average price per vehicle varies by state. Such a query is shown in Fig. 10.24.

```
SELECT state, AVG(price)
FROM Sales, Dealers
WHERE Sales.dealer = Dealers.name AND
      date >= '2006-01-04'
GROUP BY state;
```

Figure 10.24: Find average sales price by state

Notice how the query of Fig. 10.24 touches much of the data of the database, as it classifies every recent `Sales` fact by the state of the dealer that sold it. In contrast, a typical OLTP query such as "find the price at which the auto with serial number 123 was sold," would touch only a single tuple of the data, provided there was an index on serial number. □

For another OLAP example, consider a credit-card company trying to decide whether applicants for a card are likely to be credit-worthy. The company creates a warehouse of all its current customers and their payment history. OLAP queries search for factors, such as age, income, home-ownership, and zip-code, that might help predict whether customers will pay their bills on time. Similarly, hospitals may use a warehouse of patient data — their admissions, tests administered, outcomes, diagnoses, treatments, and so on — to analyze for risks and select the best modes of treatment.

10.6.3 A Multidimensional View of OLAP Data

In typical OLAP applications there is a central relation or collection of data, called the *fact table*. A fact table represents events or objects of interest, such as sales in Example 10.26. Often, it helps to think of the objects in the fact table as arranged in a multidimensional space, or "cube." Figure 10.25 suggests three-dimensional data, represented by points within the cube; we have called the dimensions car, dealer, and date, to correspond to our earlier example of automobile sales. Thus, in Fig. 10.25 we could think of each point as a sale of a single automobile, while the dimensions represent properties of that sale.

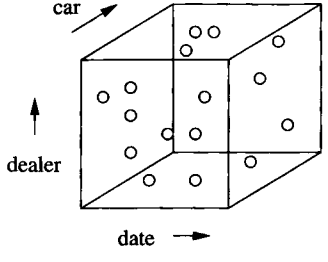

Figure 10.25: Data organized in a multidimensional space

A data space such as Fig. 10.25 will be referred to informally as a "data cube," or more precisely as a *raw-data cube* when we want to distinguish it from the more complex "data cube" of Section 10.7. The latter, which we shall refer to as a *formal* data cube when a distinction from the raw-data cube is needed, differs from the raw-data cube in two ways:

1. It includes aggregations of the data in all subsets of dimensions, as well as the data itself.

2. Points in the formal data cube may represent an initial aggregation of points in the raw-data cube. For instance, instead of the "car" dimension representing each individual car (as we suggested for the raw-data cube), that dimension might be aggregated by model only. There are points of a formal data cube that represent the total sales of all cars of a given model by a given dealer on a given day.

The distinctions between the raw-data cube and the formal data cube are reflected in the two broad directions that have been taken by specialized systems that support cube-structured data for OLAP:

1. *ROLAP*, or *Relational OLAP*. In this approach, data may be stored in relations with a specialized structure called a "star schema," described in Section 10.6.4. One of these relations is the "fact table," which contains the *raw*, or unaggregated, data, and corresponds to what we called the raw-data cube. Other relations give information about the values along each dimension. The query language, index structures, and other capabilities of the system may be tailored to the assumption that data is organized this way.

2. *MOLAP*, or *Multidimensional OLAP*. Here, a specialized structure, the formal "data cube" mentioned above, is used to hold the data, including its aggregates. Nonrelational operators may be implemented by the system to support OLAP queries on data in this structure.

10.6.4 Star Schemas

A *star schema* consists of the schema for the fact table, which links to several other relations, called "dimension tables." The fact table is at the center of the "star," whose points are the dimension tables. A fact table normally has several attributes that represent *dimensions*, and one or more *dependent* attributes that represent properties of interest for the point as a whole. For instance, dimensions for sales data might include the date of the sale, the place (store) of the sale, the type of item sold, the method of payment (e.g., cash or a credit card), and so on. The dependent attribute(s) might be the sales price, the cost of the item, or the tax, for instance.

Example 10.27: The `Sales` relation from Example 10.26

```
Sales(serialNo, date, dealer, price)
```

is a fact table. The dimensions are:

1. `serialNo`, representing the automobile sold, i.e., the position of the point in the space of possible automobiles.
2. `date`, representing the day of the sale, i.e., the position of the event in the time dimension.
3. `dealer`, representing the position of the event in the space of possible dealers.

The one dependent attribute is `price`, which is what OLAP queries to this database will typically request in an aggregation. However, queries asking for a count, rather than sum or average price would also make sense, e.g., "list the total number of sales for each dealer in the month of May, 2006." □

Supplementing the fact table are *dimension tables* describing the values along each dimension. Typically, each dimension attribute of the fact table is a foreign key, referencing the key of the corresponding dimension table, as suggested by Fig. 10.26. The attributes of the dimension tables also describe the possible groupings that would make sense in a SQL `GROUP BY` query. An example should make the ideas clearer.

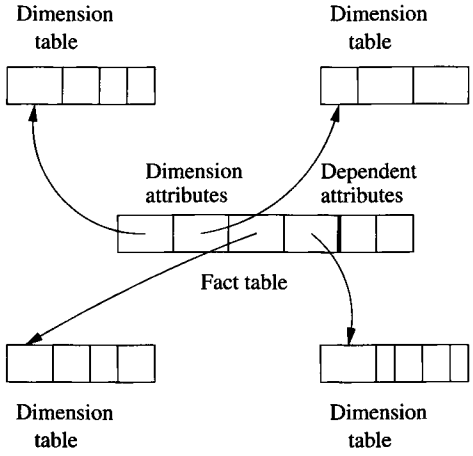

Figure 10.26: The dimension attributes in the fact table reference the keys of the dimension tables

Example 10.28: For the automobile data of Example 10.26, two of the three dimension tables might be:

```
Autos(serialNo, model, color)
Dealers(name, city, state, phone)
```

Attribute serialNo in the fact table Sales is a foreign key, referencing serialNo of dimension table Autos.[2] The attributes Autos.model and Autos.color give properties of a given auto. If we join the fact table Sales with the dimension table Autos, then the attributes model and color may be used for grouping sales in interesting ways. For instance, we can ask for a breakdown of sales by color, or a breakdown of sales of the Gobi model by month and dealer.

Similarly, attribute dealer of Sales is a foreign key, referencing name of the dimension table Dealers. If Sales and Dealers are joined, then we have additional options for grouping our data; e.g., we can ask for a breakdown of sales by state or by city, as well as by dealer.

One might wonder where the dimension table for time (the date attribute of Sales) is. Since time is a physical property, it does not make sense to store facts about time in a database, since we cannot change the answer to questions such as "in what year does the day July 5, 2007 appear?" However, since grouping by various time units, such as weeks, months, quarters, and years, is frequently desired by analysts, it helps to build into the database a notion of time, as if there were a time "dimension table" such as

```
Days(day, week, month, year)
```

A typical tuple of this imaginary "relation" would be $(5, 27, 7, 2007)$, representing July 5, 2007. The interpretation is that this day is the fifth day of the seventh month of the year 2007; it also happens to fall in the 27th full week of the year 2007. There is a certain amount of redundancy, since the week is calculable from the other three attributes. However, weeks are not exactly commensurate with months, so we cannot obtain a grouping by months from a grouping by weeks, or vice versa. Thus, it makes sense to imagine that both weeks and months are represented in this "dimension table." □

10.6.5 Slicing and Dicing

We can think of the points of the raw-data cube as partitioned along each dimension at some level of granularity. For example, in the time dimension, we might partition ("group by" in SQL terms) according to days, weeks, months, years, or not partition at all. For the cars dimension, we might partition by model, by color, by both model and color, or not partition. For dealers, we can partition by dealer, by city, by state, or not partition.

A choice of partition for each dimension "dices" the cube, as suggested by Fig. 10.27. The result is that the cube is divided into smaller cubes that represent groups of points whose statistics are aggregated by a query that performs this partitioning in its GROUP BY clause. Through the WHERE clause, a query also

[2]It happens that serialNo is also a key for the Sales relation, but there need not be an attribute that is both a key for the fact table and a foreign key for some dimension table.

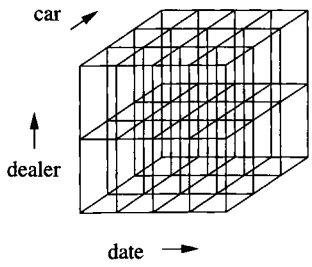

Figure 10.27: Dicing the cube by partitioning along each dimension

has the option of focusing on particular partitions along one or more dimensions (i.e., on a particular "slice" of the cube).

Example 10.29: Figure 10.28 suggests a query in which we ask for a slice in one dimension (the date), and dice in two other dimensions (car and dealer). The date is divided into four groups, perhaps the four years over which data has been accumulated. The shading in the diagram suggests that we are only interested in one of these years.

The cars are partitioned into three groups, perhaps sedans, SUV's, and convertibles, while the dealers are partitioned into two groups, perhaps the eastern and western regions. The result of the query is a table giving the total sales in six categories for the one year of interest. □

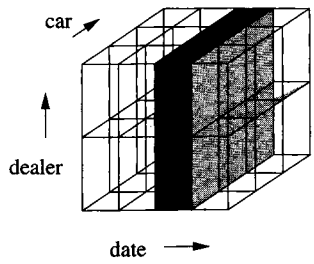

Figure 10.28: Selecting a slice of a diced cube

The general form of a so-called "slicing and dicing" query is thus:

```
SELECT <grouping attributes and aggregations>
FROM <fact table joined with some dimension tables>
WHERE <certain attributes are constant>
GROUP BY <grouping attributes>;
```

Example 10.30: Let us continue with our automobile example, but include the conceptual `Days` dimension table for time discussed in Example 10.28. If

Drill-Down and Roll-Up

Example 10.30 illustrates two common patterns in sequences of queries that slice-and-dice the data cube.

1. *Drill-down* is the process of partitioning more finely and/or focusing on specific values in certain dimensions. Each of the steps except the last in Example 10.30 is an instance of drill-down.
2. *Roll-up* is the process of partitioning more coarsely. The last step, where we grouped by years instead of months to eliminate the effect of randomness in the data, is an example of roll-up.

the Gobi isn't selling as well as we thought it would, we might try to find out which colors are not doing well. This query uses only the `Autos` dimension table and can be written in SQL as:

```
SELECT color, SUM(price)
FROM Sales NATURAL JOIN Autos
WHERE model = 'Gobi'
GROUP BY color;
```

This query dices by color and then slices by model, focusing on a particular model, the Gobi, and ignoring other data.

Suppose the query doesn't tell us much; each color produces about the same revenue. Since the query does not partition on time, we only see the total over all time for each color. We might suppose that the recent trend is for one or more colors to have weak sales. We may thus issue a revised query that also partitions time by month. This query is:

```
SELECT color, month, SUM(price)
FROM (Sales NATURAL JOIN Autos) JOIN Days ON date = day
WHERE model = 'Gobi'
GROUP BY color, month;
```

It is important to remember that the `Days` relation is not a conventional stored relation, although we may treat it as if it had the schema

```
Days(day, week, month, year)
```

The ability to use such a "relation" is one way that a system specialized to OLAP queries could differ from a conventional DBMS.

We might discover that red Gobis have not sold well recently. The next question we might ask is whether this problem exists at all dealers, or whether

only some dealers have had low sales of red Gobis. Thus, we further focus the query by looking at only red Gobis, and we partition along the dealer dimension as well. This query is:

```
SELECT dealer, month, SUM(price)
FROM (Sales NATURAL JOIN Autos) JOIN Days ON date = day
WHERE model = 'Gobi' AND color = 'red'
GROUP BY month, dealer;
```

At this point, we find that the sales per month for red Gobis are so small that we cannot observe any trends easily. Thus, we decide that it was a mistake to partition by month. A better idea would be to partition only by years, and look at only the last two years (2006 and 2007, in this hypothetical example). The final query is shown in Fig. 10.29. □

```
SELECT dealer, year, SUM(price)
FROM (Sales NATURAL JOIN Autos) JOIN Days ON date = day
WHERE model = 'Gobi' AND
      color = 'red' AND
      (year = 2006 OR year = 2007)
GROUP BY year, dealer;
```

Figure 10.29: Final slicing-and-dicing query about red Gobi sales

10.6.6 Exercises for Section 10.6

Exercise 10.6.1: An on-line seller of computers wishes to maintain data about orders. Customers can order their PC with any of several processors, a selected amount of main memory, any of several disk units, and any of several CD or DVD readers. The fact table for such a database might be:

```
Orders(cust, date, proc, memory, hd, od, quant, price)
```

We should understand attribute `cust` to be an ID that is the foreign key for a dimension table about customers, and understand attributes `proc`, `hd` (hard disk), and `od` (optical disk: CD or DVD, typically) analogously. For example, an `hd` ID might be elaborated in a dimension table giving the manufacturer of the disk and several disk characteristics. The `memory` attribute is simply an integer: the number of megabytes of memory ordered. The `quant` attribute is the number of machines of this type ordered by this customer, and the `price` attribute is the total cost of each machine ordered.

a) Which are dimension attributes, and which are dependent attributes?

b) For some of the dimension attributes, a dimension table is likely to be needed. Suggest appropriate schemas for these dimension tables.

! **Exercise 10.6.2:** Suppose that we want to examine the data of Exercise 10.6.1 to find trends and thus predict which components the company should order more of. Describe a series of drill-down and roll-up queries that could lead to the conclusion that customers are beginning to prefer a DVD drive to a CD drive.

10.7 Data Cubes

In this section, we shall consider the "formal" data cube and special operations on data presented in this form. Recall from Section 10.6.3 that the formal data cube (just "data cube" in this section) precomputes all possible aggregates in a systematic way. Surprisingly, the amount of extra storage needed is often tolerable, and as long as the warehoused data does not change, there is no penalty incurred trying to keep all the aggregates up-to-date.

In the data cube, it is normal for there to be some aggregation of the raw data of the fact table before it is entered into the data-cube and its further aggregates computed. For instance, in our cars example, the dimension we thought of as a serial number in the star schema might be replaced by the model of the car. Then, each point of the data cube becomes a description of a model, a dealer and a date, together with the sum of the sales for that model, on that date, by that dealer. We shall continue to call the points of the (formal) data cube a "fact table," even though the interpretation of the points may be slightly different from fact tables in a star schema built from a raw-data cube.

10.7.1 The Cube Operator

Given a fact table F, we can define an augmented table CUBE(F) that adds an additional value, denoted *, to each dimension. The * has the intuitive meaning "any," and it represents aggregation along the dimension in which it appears. Figure 10.30 suggests the process of adding a border to the cube in each dimension, to represent the * value and the aggregated values that it implies. In this figure we see three dimensions, with the lightest shading representing aggregates in one dimension, darker shading for aggregates over two dimensions, and the darkest cube in the corner for aggregation over all three dimensions. Notice that if the number of values along each dimension is reasonably large, then the "border" represents only a small addition to the volume of the cube (i.e., the number of tuples in the fact table). In that case, the size of the stored data CUBE(F) is not much greater than the size of F itself.

A tuple of the table CUBE(F) that has * in one or more dimensions will have for each dependent attribute the sum (or another aggregate function) of the values of that attribute in all the tuples that we can obtain by replacing

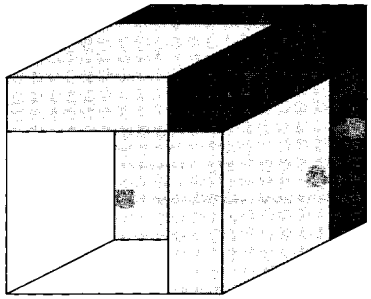

Figure 10.30: The cube operator augments a data cube with a border of aggregations in all combinations of dimensions

the *'s by real values. In effect, we build into the data the result of aggregating along any set of dimensions. Notice, however, that the CUBE operator does not support aggregation at intermediate levels of granularity based on values in the dimension tables. For instance, we may either leave data broken down by day (or whatever the finest granularity for time is), or we may aggregate time completely, but we cannot, with the CUBE operator alone, aggregate by weeks, months, or years.

Example 10.31: Let us reconsider the Aardvark database from Example 10.26 in the light of what the CUBE operator can give us. Recall the fact table from that example is

```
Sales(serialNo, date, dealer, price)
```

However, the dimension represented by `serialNo` is not well suited for the cube, since the serial number is a key for `Sales`. Thus, summing the price over all dates, or over all dealers, but keeping the serial number fixed has no effect; we would still get the "sum" for the one auto with that serial number. A more useful data cube would replace the serial number by the two attributes — model and color — to which the serial number connects `Sales` via the dimension table `Autos`. Notice that if we replace `serialNo` by `model` and `color`, then the cube no longer has a key among its dimensions. Thus, an entry of the cube would have the total sales price for all automobiles of a given model, with a given color, by a given dealer, on a given date.

There is another change that is useful for the data-cube implementation of the `Sales` fact table. Since the CUBE operator normally sums dependent variables, and we might want to get average prices for sales in some category, we need both the sum of the prices for each category of automobiles (a given model of a given color sold on a given day by a given dealer) and the total number of sales in that category. Thus, the relation `Sales` to which we apply the CUBE operator is

```
Sales(model, color, date, dealer, val, cnt)
```

The attribute `val` is intended to be the total price of all automobiles for the given model, color, date, and dealer, while `cnt` is the total number of automobiles in that category.

Now, let us consider the relation CUBE(`Sales`). A hypothetical tuple that would be in CUBE(`Sales`) is:

```
('Gobi', 'red', '2001-05-21', 'Friendly Fred', 45000, 2)
```

The interpretation is that on May 21, 2001, dealer Friendly Fred sold two red Gobis for a total of $45,000. In `Sales`, this tuple might appear as well, or there could be in `Sales` two tuples, each with a `cnt` of 1, whose `val`'s summed to 45,000.

The tuple

```
('Gobi', *, '2001-05-21', 'Friendly Fred', 152000, 7)
```

says that on May 21, 2001, Friendly Fred sold seven Gobis of all colors, for a total price of $152,000. Note that this tuple is in CUBE(`Sales`) but not in `Sales`.

Relation CUBE(`Sales`) also contains tuples that represent the aggregation over more than one attribute. For instance,

```
('Gobi', *, '2001-05-21', *, 2348000, 100)
```

says that on May 21, 2001, there were 100 Gobis sold by all the dealers, and the total price of those Gobis was $2,348,000.

```
('Gobi', *, *, *, 1339800000, 58000)
```

Says that over all time, dealers, and colors, 58,000 Gobis have been sold for a total price of $1,339,800,000. Lastly, the tuple

```
(*, *, *, *, 3521727000, 198000)
```

tells us that total sales of all Aardvark models in all colors, over all time at all dealers is 198,000 cars for a total price of $3,521,727,000. □

10.7.2 The Cube Operator in SQL

SQL gives us a way to apply the cube operator within queries. If we add the term `WITH CUBE` to a group-by clause, then we get not only the tuple for each group, but also the tuples that represent aggregation along one or more of the dimensions along which we have grouped. These tuples appear in the result with `NULL` where we have used *.

Example 10.32: We can construct a materialized view that is the data cube we called CUBE(Sales) in Example 10.31 by the following:

```
CREATE MATERIALIZED VIEW SalesCube AS
    SELECT model, color, date, dealer, SUM(val), SUM(cnt)
    FROM Sales
    GROUP BY model, color, date, dealer WITH CUBE;
```

The view SalesCube will then contain not only the tuples that are implied by the group-by operation, such as

```
('Gobi', 'red', '2001-05-21', 'Friendly Fred', 45000, 2)
```

but will also contain those tuples of CUBE(Sales) that are constructed by rolling up the dimensions listed in the GROUP BY. Some examples of such tuples would be:

```
('Gobi', NULL, '2001-05-21', 'Friendly Fred', 152000, 7)
('Gobi', NULL, '2001-05-21', NULL, 2348000, 100)
('Gobi', NULL, NULL, NULL, 1339800000, 58000)
(NULL, NULL, NULL, NULL, 3521727000, 198000)
```

Recall that NULL is used to indicate a rolled-up dimension, equivalent to the * we used in the abstract CUBE operator's result. □

A variant of the CUBE operator, called ROLLUP, produces the additional aggregated tuples only if they aggregate over a tail of the sequence of grouping attributes. We indicate this option by appending WITH ROLLUP to the group-by clause.

Example 10.33: We can get the part of the data cube for Sales that is constructed by the ROLLUP operator with:

```
CREATE MATERIALIZED VIEW SalesRollup AS
    SELECT model, color, date, dealer, SUM(val), SUM(cnt)
    FROM Sales
    GROUP BY model, color, date, dealer WITH ROLLUP;
```

The view SalesRollup will contain tuples

```
('Gobi', 'red', '2001-05-21', 'Friendly Fred', 45000, 2)
('Gobi', 'red', '2001-05-21', NULL, 3678000, 135)
('Gobi', 'red', NULL, NULL, 657100000, 34566)
('Gobi', NULL, NULL, NULL, 1339800000, 58000)
(NULL, NULL, NULL, NULL, 3521727000, 198000)
```

because these tuples represent aggregation along some dimension and all dimensions, if any, that follow it in the list of grouping attributes.

However, SalesRollup would not contain tuples such as

```
('Gobi', NULL, '2001-05-21', 'Friendly Fred', 152000, 7)
('Gobi', NULL, '2001-05-21', NULL, 2348000, 100)
```

These each have NULL in a dimension (color in both cases) but do not have NULL in one or more of the following dimension attributes. □

10.7.3 Exercises for Section 10.7

Exercise 10.7.1: What is the ratio of the size of CUBE(F) to the size of F if fact table F has the following characteristics?

a) F has ten dimension attributes, each with ten different values.

b) F has ten dimension attributes, each with two different values.

Exercise 10.7.2: Use the materialized view SalesCube from Example 10.32 to answer the following queries:

a) Find the total sales of blue cars for each dealer.

b) Find the total number of green Gobis sold by dealer "Smilin' Sally."

c) Find the average number of Gobis sold on each day of March, 2007 by each dealer.

! **Exercise 10.7.3:** What help, if any, would the rollup SalesRollup of Example 10.33 be for each of the queries of Exercise 10.7.2?

Exercise 10.7.4: In Exercise 10.6.1 we spoke of PC-order data organized as a fact table with dimension tables for attributes cust, proc, memory, hd, and od. That is, each tuple of the fact table Orders has an ID for each of these attributes, leading to information about the PC involved in the order. Write a SQL query that will produce the data cube for this fact table.

Exercise 10.7.5: Answer the following queries using the data cube from Exercise 10.7.4. If necessary, use dimension tables as well. You may invent suitable names and attributes for the dimension tables.

a) Find, for each processor speed, the total number of computers ordered in each month of the year 2007.

b) List for each type of hard disk (e.g., SCSI or IDE) and each processor type the number of computers ordered.

c) Find the average price of computers with 3.0 gigahertz processors for each month from Jan., 2005.

! **Exercise 10.7.6:** The cube tuples mentioned in Example 10.32 are not in the rollup of Example 10.33. Are there other rollups that would contain these tuples?

!! **Exercise 10.7.7:** If the fact table F to which we apply the CUBE operator is sparse (i.e., there are many fewer tuples in F than the product of the number of possible values along each dimension), then the ratio of the sizes of CUBE(F) and F can be very large. How large can it be?

10.8 Summary of Chapter 10

- ✦ *Privileges*: For security purposes, SQL systems allow many different kinds of privileges to be managed for database elements. These privileges include the right to select (read), insert, delete, or update relations, the right to reference relations (refer to them in a constraint), and the right to create triggers.

- ✦ *Grant Diagrams*: Privileges may be granted by owners to other users or to the general user PUBLIC. If granted with the grant option, then these privileges may be passed on to others. Privileges may also be revoked. The grant diagram is a useful way to remember enough about the history of grants and revocations to keep track of who has what privilege and from whom they obtained those privileges.

- ✦ *SQL Recursive Queries*: In SQL, one can define a relation recursively — that is, in terms of itself. Or, several relations can be defined to be mutually recursive.

- ✦ *Monotonicity*: Negations and aggregations involved in a SQL recursion must be monotone — inserting tuples in one relation does not cause tuples to be deleted from any relation, including itself. Intuitively, a relation may not be defined, directly or indirectly, in terms of a negation or aggregation of itself.

- ✦ *The Object-Relational Model*: An alternative to pure object-oriented database models like ODL is to extend the relational model to include the major features of object-orientation. These extensions include nested relations, i.e., complex types for attributes of a relation, including relations as types. Other extensions include methods defined for these types, and the ability of one tuple to refer to another through a reference type.

- ✦ *User-Defined Types in SQL*: Object-relational capabilities of SQL are centered around the UDT, or user-defined type. These types may be declared by listing their attributes and other information, as in table declarations. In addition, methods may be declared for UDT's.

- ✦ *Relations With a UDT as Type*: Instead of declaring the attributes of a relation, we may declare that relation to have a UDT. If we do so, then its tuples have one component, and this component is an object of the UDT.

- *Reference Types*: A type of an attribute can be a reference to a UDT. Such attributes essentially are pointers to objects of that UDT.

- *Object Identity for UDT's*: When we create a relation whose type is a UDT, we declare an attribute to serve as the "object-ID" of each tuple. This component is a reference to the tuple itself. Unlike in object-oriented systems, this "OID" column may be accessed by the user, although it is rarely meaningful.

- *Accessing components of a UDT*: SQL provides observer and mutator functions for each attribute of a UDT. These functions, respectively, return and change the value of that attribute when applied to any object of that UDT.

- *Ordering Functions for UDT's*: In order to compare objects, or to use SQL operations such as `DISTINCT`, `GROUP BY`, or `ORDER BY`, it is necessary for the implementer of a UDT to provide a function that tells whether two objects are equal or whether one precedes the other.

- *OLAP*: On-line analytic processing involves complex queries that touch all or much of the data, at the same time. Often, a separate database, called a data warehouse, is constructed to run such queries while the actual database is used for short-term transactions (OLTP, or on-line transaction processing).

- *ROLAP and MOLAP*: It is frequently useful, for OLAP queries, to think of the data as residing in a multidimensional space, with dimensions corresponding to independent aspects of the data represented. Systems that support such a view of data take either a relational point of view (ROLAP, or relational-OLAP systems), or use the specialized data-cube model (MOLAP, or multidimensional-OLAP systems).

- *Star Schemas*: In a star schema, each data element (e.g., a sale of an item) is represented in one relation, called the fact table, while information helping to interpret the values along each dimension (e.g., what kind of product is item 1234?) is stored in a dimension table for each dimension.

- *The Cube Operator*: A specialized operator called cube pre-aggregates the fact table along all subsets of dimensions. It may add little to the space needed by the fact table, and greatly increases the speed with which many OLAP queries can be answered.

- *Data Cubes in SQL*: We can turn the result of a query into a data cube by appending `WITH CUBE` to a group-by clause. We can also construct a portion of the cube by using `WITH ROLLUP` there.

10.9 References for Chapter 10

The ideas behind the SQL authorization mechanism originated in [4] and [1].

Material on object-relational features of SQL can be obtained as described in the bibliographic notes to Chapter 6.

The source of the SQL-99 proposal for recursion is [2]. This proposal, and its monotonicity requirement, built on foundations developed over many years, involving recursion and negation in Datalog; see [5].

The cube operator was proposed in [3].

1. R. Fagin, "On an authorization mechanism," *ACM Transactions on Database Systems* **3**:3, pp. 310–319, 1978.

2. S. J. Finkelstein, N. Mattos, I. S. Mumick, and H. Pirahesh, "Expressing recursive queries in SQL," ISO WG3 report X3H2–96–075, March, 1996.

3. J. N. Gray, A. Bosworth, A. Layman, and H. Pirahesh, "Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Proc. Intl. Conf. on Data Engineering* (1996), pp. 152–159.

4. P. P. Griffiths and B. W. Wade, "An authorization mechanism for a relational database system," *ACM Transactions on Database Systems* **1**:3, pp. 242–255, 1976.

5. J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Volume I*, Computer Science Press, New York, 1988.

Part III

Modeling and Programming for Semistructured Data

Chapter 11

The Semistructured-Data Model

We now turn to a different kind of data model. This model, called "semistructured," is distinguished by the fact that the schema is implied by the data, rather than being declared separately from the data as is the case for the relational model and all the other models we studied up to this point. After a general discussion of semistructured data, we turn to the most important manifestation of this idea: XML. We shall cover ways to describe XML data, in effect enforcing a schema for this "schemaless" data. These methods include DTD's (Document Type Definitions) and the language XML Schema.

11.1 Semistructured Data

The *semistructured-data* model plays a special role in database systems:

1. It serves as a model suitable for *integration* of databases, that is, for describing the data contained in two or more databases that contain similar data with different schemas.

2. It serves as the underlying model for notations such as XML, to be taken up in Section 11.2, that are being used to share information on the Web.

In this section, we shall introduce the basic ideas behind "semistructured data" and how it can represent information more flexibly than the other models we have met previously.

11.1.1 Motivation for the Semistructured-Data Model

The models we have seen so far — E/R, UML, relational, ODL — each start with a schema. The schema is a rigid framework into which data is placed. This

rigidity provides certain advantages. Especially, the relational model owes much of its success to the existence of efficient implementations. This efficiency comes from the fact that the data in a relational database must fit the schema, and the schema is known to the query processor. For instance, fixing the schema allows the data to be organized with data structures that support efficient answering of queries, as we discussed in Section 8.3.

On the other hand, interest in the semistructured-data model is motivated primarily by its flexibility. In particular, semistructured data is "schemaless." More precisely, the data is *self-describing*; it carries information about what its schema is, and that schema can vary arbitrarily, both over time and within a single database.

One might naturally wonder whether there is an advantage to creating a database without a schema, where one could enter data at will, and attach to the data whatever schema information you felt was appropriate for that data. There are actually some small-scale information systems, such as Lotus Notes, that take the self-describing-data approach. This flexibility may make query processing harder, but it offers significant advantages to users. For example, we can maintain a database of movies in the semistructured model and add new attributes like "would I like to see this movie?" as we wish. The attributes do not need to have a value for all movies, or even for more than one movie. Likewise, we can add relationships like "homage to," without having to change the schema or even represent the relationship in more than one pair of movies.

11.1.2 Semistructured Data Representation

A database of *semistructured data* is a collection of *nodes*. Each node is either a *leaf* or *interior*. Leaf nodes have associated data; the type of this data can be any atomic type, such as numbers and strings. Interior nodes have one or more arcs out. Each arc has a *label*, which indicates how the node at the head of the arc relates to the node at the tail. One interior node, called the *root*, has no arcs entering and represents the entire database. Every node must be reachable from the root, although the graph structure is not necessarily a tree.

Example 11.1: Figure 11.1 is an example of a semistructured database about stars and movies. We see a node at the top labeled *Root*; this node is the entry point to the data and may be thought of as representing all the information in the database. The central objects or entities — stars and movies in this case — are represented by nodes that are children of the root.

We also see many leaf nodes. At the far left is a leaf labeled `Carrie Fisher`, and at the far right is a leaf labeled 1977, for instance. There are also many interior nodes. Three particular nodes we have labeled *cf*, *mh*, and *sw*, standing for "Carrie Fisher," "Mark Hamill," and "Star Wars," respectively. These labels are not part of the model, and we placed them on these nodes only so we would have a way of referring to the nodes, which otherwise would be nameless, in the text. We may think of node *sw*, for instance, as representing the concept "Star

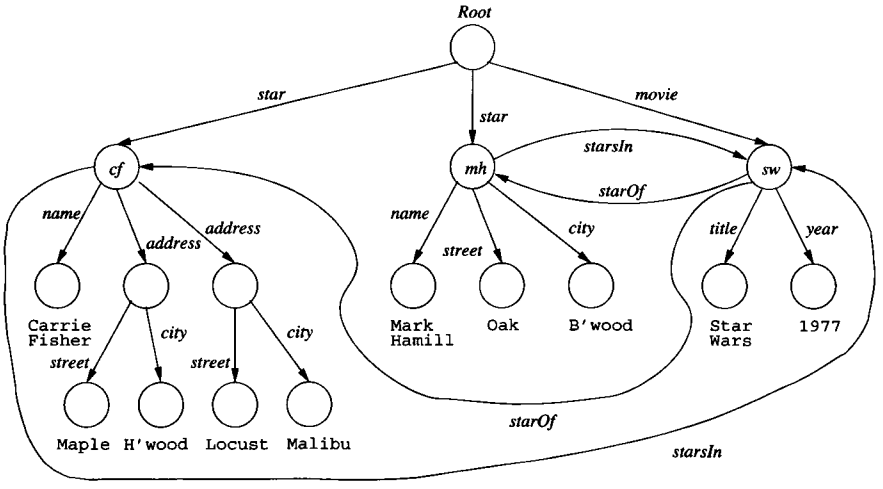

Figure 11.1: Semistructured data representing a movie and stars

Wars": the title and year of this movie, other information not shown, such as its length, and its stars, two of which are shown. □

A label L on the arc from node N to node M can play one of two roles:

1. It may be possible to think of N as representing an object or entity, while M represents one of its attributes. Then, L represents the name of the attribute.

2. We may be able to think of N and M as objects or entities and L as the name of a relationship from N to M.

Example 11.2: Consider Fig. 11.1 again. The node indicated by *cf* may be thought of as representing the `Star` object for Carrie Fisher. We see, leaving this node, an arc labeled *name*, which represents the attribute `name` and leads to a leaf node holding the correct name. We also see two arcs, each labeled *address*. These arcs lead to unnamed nodes which we may think of as representing two addresses of Carrie Fisher. There is no schema to tell us whether stars can have more than one address; we simply put two address nodes in the graph if we feel it is appropriate.

Notice in Fig. 11.1 how both nodes have out-arcs labeled *street* and *city*. Moreover, these arcs each lead to leaf nodes with the appropriate atomic values. We may think of *address* nodes as structs or objects with two fields, named *street* and *city*. However, in the semistructured model, it is entirely appropriate to add other components, e.g., zip, to some addresses, or to have one or both fields missing.

The other kind of arc also appears in Fig. 11.1. For instance, the node *cf* has an out-arc leading to the node *sw* and labeled *starsIn*. The node *mh* (for Mark Hamill) has a similar arc, and the node *sw* has arcs labeled *starOf* to both nodes *cf* and *mh*. These arcs represent the stars-in relationship between stars and movies. □

11.1.3 Information Integration Via Semistructured Data

The flexibility and self-describing nature of semistructured data has made it important in two applications. We shall discuss its use for data exchange in Section 11.2, but here we shall consider its use as a tool for information integration. As databases have proliferated, it has become a common requirement that data in two or more of them be accessible as if they were one database. For instance, companies may merge; each has its own personnel database, its own database of sales, inventory, product designs, and perhaps many other matters. If corresponding databases had the same schemas, then combining them would be simple; for instance, we could take the union of the tuples in two relations that had the same schema and played the same roles in the the two databases.

However, life is rarely that simple. Independently developed databases are unlikely to share a schema, even if they talk about the same things, such as personnel. For instance, one employee database may record spouse-name, another not. One may have a way to represent several addresses, phones, or emails for an employee, another database may allow only one of each. One may treat consultants as employees, another not. One database might be relational, another object-oriented.

To make matters more complex, databases tend over time to be used in so many different applications that it is impossible to turn them off and copy or translate their data into another database, even if we could figure out an efficient way to transform the data from one schema to another. This situation is often referred to as the *legacy-database problem*; once a database has been in existence for a while, it becomes impossible to disentangle it from the applications that grow up around it, so the database can never be decommissioned.

A possible solution to the legacy-database problem is suggested in Fig. 11.2. We show two legacy databases with an interface; there could be many legacy systems involved. The legacy systems are each unchanged, so they can support their usual applications.

For flexibility in integration, the interface supports semistructured data, and the user is allowed to query the interface using a query language that is suitable for such data. The semistructured data may be constructed by translating the data at the sources, using components called *wrappers* (or "adapters") that are each designed for the purpose of translating one source to semistructured data.

Alternatively, the semistructured data at the interface may not exist at all. Rather, the user queries the interface as if there were semistructured data, while the interface answers the query by posing queries to the sources, each referring to the schema found at that source.

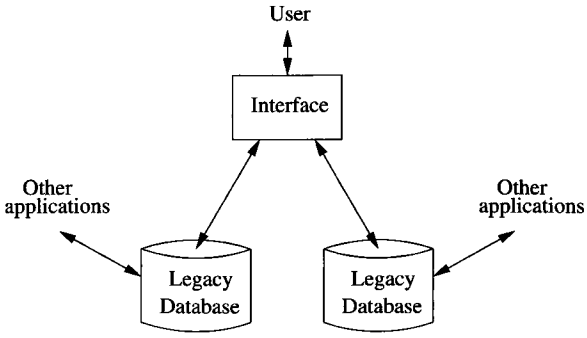

Figure 11.2: Integrating two legacy databases through an interface that supports semistructured data

Example 11.3: We can see in Fig. 11.1 a possible effect of information about stars being gathered from several sources. Notice that the address information for Carrie Fisher has an address concept, and the address is then broken into street and city. That situation corresponds roughly to data that had a nested-relation schema like `Stars(name, address(street, city))`.

On the other hand, the address information for Mark Hamill has no address concept at all, just street and city. This information may have come from a schema such as `Stars(name, street, city)` that can represent only one address for a star. Some of the other variations in schema that are not reflected in the tiny example of Fig. 11.1, but that could be present if movie information were obtained from several sources, include: optional film-type information, a director, a producer or producers, the owning studio, revenue, and information on where the movie is currently playing. □

11.1.4 Exercises for Section 11.1

Exercise 11.1.1: Since there is no schema to design in the semistructured-data model, we cannot ask you to design schemas to describe different situations. Rather, in the following exercises we shall ask you to suggest how particular data might be organized to reflect certain facts.

a) Add to Fig. 11.1 the facts that *Star Wars* was directed by George Lucas and produced by Gary Kurtz.

b) Add to Fig. 11.1 information about *Empire Strikes Back* and *Return of the Jedi*, including the facts that Carrie Fisher and Mark Hamill appeared in these movies.

c) Add to (b) information about the studio (Fox) for these movies and the address of the studio (Hollywood).

Exercise 11.1.2: Suggest how typical data about banks and customers, as in Exercise 4.1.1, could be represented in the semistructured model.

Exercise 11.1.3: Suggest how typical data about players, teams, and fans, as was described in Exercise 4.1.3, could be represented in the semistructured model.

Exercise 11.1.4: Suggest how typical data about a genealogy, as was described in Exercise 4.1.6, could be represented in the semistructured model.

! **Exercise 11.1.5:** UML and the semistructured-data model are both "graphical" in nature, in the sense that they use nodes, labels, and connections among nodes as the medium of expression. Yet there is an essential difference between the two models. What is it?

11.2 XML

XML (*Extensible Markup Language*) is a tag-based notation designed originally for "marking" documents, much like the familiar HTML. Nowadays, data with XML "markup" can be represented in many ways. However, in this section we shall refer to XML data as represented in one or more documents. While HTML's tags talk about the presentation of the information contained in documents — for instance, which portion is to be displayed in italics or what the entries of a list are — XML tags are intended to talk about the meanings of pieces of the document.

In this section we shall introduce the rudiments of XML. We shall see that it captures, in a linear form, the same structure as do the graphs of semistructured data introduced in Section 11.1. In particular, tags can play the same role as the labels on the arcs of a semistructured-data graph.

11.2.1 Semantic Tags

Tags in XML are text surrounded by triangular brackets, i.e., `<...>`, as in HTML. Also as in HTML, tags generally come in matching pairs, with an *opening tag* like `<Foo>` and a matched *closing* tag that is the same word with a slash, like `</Foo>`. Between a matching pair `<Foo>` and `</Foo>`, there can be text, including text with nested HTML tags, and any number of other nested matching pairs of XML tags. A pair of matching tags and everything that comes between them is called an *element*.

A single tag, with no matched closing tag, is also permitted in XML. In this form, the tag has a slash before the right bracket, for example, `<Foo/>`. Such a tag cannot have any other elements or text nested within it. It can, however, have attributes (see Section 11.2.4).

11.2.2 XML With and Without a Schema

XML is designed to be used in two somewhat different modes:

1. *Well-formed* XML allows you to invent your own tags, much like the arc-labels in semistructured data. This mode corresponds quite closely to semistructured data, in that there is no predefined schema, and each document is free to use whatever tags the author of the document wishes. Of course the nesting rule for tags must be obeyed, or the document is not well-formed.

2. *Valid* XML involves a "DTD," or "Document Type Definition" (see Section 11.3) that specifies the allowable tags and gives a grammar for how they may be nested. This form of XML is intermediate between the strict-schema models such as the relational model, and the completely schemaless world of semistructured data. As we shall see in Section 11.3, DTD's generally allow more flexibility in the data than does a conventional schema; DTD's often allow optional fields or missing fields, for instance.

11.2.3 Well-Formed XML

The minimal requirement for well-formed XML is that the document begin with a declaration that it is XML, and that it have a *root element* that is the entire body of the text. Thus, a well-formed XML document would have an outer structure like:

```
<? xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<SomeTag>
    ...
</SomeTag>
```

The first line indicates that the file is an XML document. The encoding UTF-8 (UTF = "Unicode Transformation Format") is a common choice of encoding for characters in documents, because it is compatible with ASCII and uses only one byte for the ASCII characters. The attribute `standalone = "yes"` indicates that there is no DTD for this document; i.e., it is well-formed XML. Notice that this initial declaration is delineated by special markers `<?...?>`. The root element for this document is labeled `<SomeTag>`.

Example 11.4: In Fig. 11.3 is an XML document that corresponds roughly to the data in Fig. 11.1. In particular, it corresponds to the tree-like portion of the semistructured data — the root and all the nodes and arcs except the "sideways" arcs among the nodes *cf*, *mh*, and *sw*. We shall see in Section 11.2.4 how those may be represented.

The root element is `StarMovieData`. Within this element, we see two elements, each beginning with the tag `<Star>` and ending with its matching

```
<? xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<StarMovieData>
    <Star>
        <Name>Carrie Fisher</Name>
        <Address>
            <Street>123 Maple St.</Street>
            <City>Hollywood</City>
        </Address>
        <Address>
            <Street>5 Locust Ln.</Street>
            <City>Malibu</City>
        </Address>
    </Star>
    <Star>
        <Name>Mark Hamill</Name>
        <Street>456 Oak Rd.</Street>
        <City>Brentwood</City>
    </Star>
    <Movie>
        <Title>Star Wars</Title>
        <Year>1977</Year>
    </Movie>
</StarMovieData>
```

Figure 11.3: An XML document about stars and movies

</Star>. Within each element is a subelement giving the name of the star. One element, for Carrie Fisher, also has two subelements, each giving the address of one of her homes. These elements are each delineated by an <Address> opening tag and its matched closing tag. The element for Mark Hamill has only subelements for one street and one city, and does not use an <Address> tag to group these. This distinction appeared as well in Fig. 11.1. We also see one element with opening tag <Movie> and its matched closing tag. This element has subelements for the title and year of the movie.

Notice that the document of Fig. 11.3 does not represent the relationship "stars-in" between stars and movies. We could indicate the movies of a star by including, within the element devoted to that star, the titles and years of their movies. Figure 11.4 is an example of this representation. □

11.2.4 Attributes

As in HTML, an XML element can have *attributes* (name-value pairs) within its opening tag. An attribute is an alternative way to represent a leaf node of semistructured data. Attributes, like tags, can represent labeled arcs in

```
<Star>
    <Name>Mark Hamill</Name>
    <Street>Oak</Street>
    <City>Brentwood</City>
    <Movie>
        <Title>Star Wars</Title>
        <Year>1977</Year>
    </Movie>
    <Movie>
        <Title>Empire Strikes Back</Title>
        <Year>1980</Year>
    </Movie>
</Star>
```

Figure 11.4: Nesting movies within stars

a semistructured-data graph. Attributes can also be used to represent the "sideways" arcs as in Fig. 11.1.

Example 11.5: The *title* or *year* children of the movie node labeled *sw* could be represented directly in the `<Movie>` element, rather than being represented by nested elements. That is, we could replace the `<Movie>` element of Fig. 11.3 by:

```
<Movie year = 1977><Title>Star Wars</Title></Movie>
```

We could even make both child nodes be attributes by:

```
<Movie title = "Star Wars" year = 1977></Movie>
```

or even:

```
<Movie title = "Star Wars" year = 1977 />
```

Notice that here we use a single tag without a matched closing tag, as indicated by the slash at the end. □

11.2.5 Attributes That Connect Elements

An important use for attributes is to represent connections in a semistructured data graph that do not form a tree. We shall see in Section 11.3.4 how to declare certain attributes to be identifiers for their elements. We shall also see how to declare that other attributes are references to these element identifiers. For the moment, let us just see an example of how these attributes could be used.

Example 11.6: Figure 11.5 can be interpreted as an exact representation in XML of the semistructured data graph of Fig. 11.1. However, in order to make the interpretation, we need to have enough schema information that we know the attribute `starID` is an identifier for the element in which it appears. That is, `cf` is the identifier of the first `<Star>` element (for Carrie Fisher) and `mh` is the identifier of the second `<Star>` element (for Mark Hamill). Likewise, we must establish that the attribute `movieID` within a `<Movie>` tag is an identifier for that element. Thus, `sw` is an identifier for the lone `<Movie>` element in Fig. 11.5.

```
<? xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<StarMovieData>
    <Star starID = "cf" starredIn = "sw">
        <Name>Carrie Fisher</Name>
        <Address>
            <Street>123 Maple St.</Street>
            <City>Hollywood</City>
        </Address>
        <Address>
            <Street>5 Locust Ln.</Street>
            <City>Malibu</City>
        </Address>
    </Star>
    <Star starID = "mh" starredIn = "sw">
        <Name>Mark Hamill</Name>
        <Street>456 Oak Rd.</Street>
        <City>Brentwood</City>
    </Star>
    <Movie movieID = "sw" starsOf = "cf", "mh">
        <Title>Star Wars</Title>
        <Year>1977</Year>
    </Movie>
</StarMovieData>
```

Figure 11.5: Adding stars-in information to our XML document

Moreover, the schema must also say that the attributes `starredIn` for `<Star>` elements and `starsOf` for `<Movie>` elements are references to one or more ID's. That is, the value `sw` for `starredIn` within each of the `<Movie>` elements says that both Carrie Fisher and Mark Hamill starred in *Star Wars*. Likewise, the list of ID's `cf` and `mh` that is the value of `starsOf` in the `<Movie>` element says that both these stars were stars of *Star Wars*. □

11.2.6 Namespaces

There are situations in which XML data involves tags that come from two or more different sources, and which may therefore have conflicting names. For example, we would not want to confuse an HTML tag used in text with an XML tag that represents the meaning of that text. In Section 11.4, we shall see how XML Schema requires tags from two separate vocabularies. To distinguish among different vocabularies for tags in the same document, we can use a *namespace* for a set of tags.

To say that an element's tag should be interpreted as part of a certain namespace, we can use the attribute `xmlns` in its opening tag. There is a special form used for this attribute:

`xmlns:`*name*`="`*URI*`"`

Within the element having this attribute, *name* can modify any tag to say the tag belongs to this namespace. That is, we can create *qualified* names of the form *name*:*tag*, where *name* is the name of the namespace to which the tag *tag* belongs.

The URI (Universal Resource Identifier) is typically a URL referring to a document that describes the meaning of the tags in the namespace. This description need not be formal; it could be an informal article about expectations. It could even be nothing at all, and still serve the purpose of distinguishing different tags that had the same name.

Example 11.7: Suppose we want to say that in element `StarMovieData` of Fig. 11.5 certain tags belong to the namespace defined in the document `infolab.stanford.edu/movies`. We could choose a name such as `md` for the namespace by using the opening tag:

```
<md:StarMovieData xmlns:md=
    "http://infolab.stanford.edu/movies">
```

Our intent is that `StarMovieData` itself is part of this namespace, so it gets the prefix `md:`, as does its closing tag `/md:StarMovieData`. Inside this element, we have the option of asserting that the tags of subelements belong to this namespace by prefixing their opening and closing tags with `md:`. □

11.2.7 XML and Databases

Information encoded in XML is not always intended to be stored in a database. It has become common for computers to share data across the Internet by passing messages in the form of XML elements. These messages live for a very short time, although they may have been generated using data from one database and wind up being stored as tuples of a database at the receiving end. For example, the XML data in Fig. 11.5 might be turned into some tuples

to insert into relations `MovieStar` and `StarsIn` of our running example movie database.

However, it is becoming increasingly common for XML to appear in roles traditionally reserved for relational databases. For example, we discussed in Section 11.1.3 how systems that integrate the data of an enterprise produce integrated views of many databases. XML is becoming an important option as the way to represent these views, as an alternative to views consisting of relations or classes of objects. The integrated views are then queried using one of the specialized XML query languages that we shall meet in Chapter 12.

When we store XML in a database, we must deal with the requirement that access to information must be efficient, especially for very large XML documents or very large collections of small documents.[1] A relational DBMS provides indexes and other tools for making access efficient, a subject we introduced in Section 8.3. There are two approaches to storing XML to provide some efficiency:

1. Store the XML data in a parsed form, and provide a library of tools to navigate the data in that form. Two common standards are called SAX (Simple API for XML) and DOM (Document Object Model).

2. Represent the documents and their elements as relations, and use a conventional, relational DBMS to store them.

In order to represent XML documents as relations, we should start by giving each document and each element of those documents a unique ID. For the document, the ID could be its URL or path in a file system. A possible relational database schema is:

```
DocRoot(docID, rootElementID)
SubElement(parentID, childID, position)
ElementAttribute(elementID, name, value)
ElementValue(elementID, value)
```

This schema is suitable for documents that obey the restriction that each element either contains only text or contains only subelements. Accommodating elements with *mixed content* of text and subelements is left as an exercise.

The first relation, `DocRoot` relates document ID's to the ID's of their root element. The second relation, `SubElement`, connects an element (the "parent") to each of its immediate subelements ("children"). The third attribute of `SubElement` gives the position of the child among all the children of the parent.

The third relation, `ElementAttribute` relates elements to their attributes; each tuple gives the name and value of one of the attributes of an element. Finally, `ElementValue` relates those elements that have no subelements to the text, if any, that is contained in that element.

[1] Recall that XML data need not take the form of documents (i.e., a header with a root element) at all. For example, XML data could be a stream of elements without headers. However, we shall continue to speak of "documents" as XML data.

There is a small matter that values of attributes and elements can have different types, e.g., integers or strings, while relational attributes each have a unique type. We could treat the two attributes named `value` as always being strings, and interpret those strings that were integers or another type properly as we processed the data. Or we could split each of the last two relations into as many relations as there are different types of data.

11.2.8 Exercises for Section 11.2

Exercise 11.2.1: Repeat Exercise 11.1.1 using XML.

Exercise 11.2.2: Show that any relation can be represented by an XML document. *Hint*: Create an element for each tuple with a subelement for each component of that tuple.

! **Exercise 11.2.3:** How would you represent an empty element (one that had neither text nor subelements) in the database schema of Section 11.2.7?

! **Exercise 11.2.4:** In Section 11.2.7 we gave a database schema for representing documents that do not have *mixed content* — elements that contain a mixture of text (`#PCDATA`) and subelements. Show how to modify the schema when elements can have mixed content.

11.3 Document Type Definitions

For a computer to process XML documents automatically, it is helpful for there to be something like a schema for the documents. It is useful to know what kinds of elements can appear in a collection of documents and how elements can be nested. The description of the schema is given by a grammar-like set of rules, called a *document type definition*, or DTD. It is intended that companies or communities wishing to share data will each create a DTD that describes the form(s) of the data they share, thus establishing a shared view of the semantics of their elements. For instance, there could be a DTD for describing protein structures, a DTD for describing the purchase and sale of auto parts, and so on.

11.3.1 The Form of a DTD

The gross structure of a DTD is:

```
<!DOCTYPE root-tag [
    <!ELEMENT element-name (components)>
        more elements
]>
```

The opening *root-tag* and its matched closing tag surround a document that conforms to the rules of this DTD. Element declarations, introduced by `!ELEMENT`, give the tag used to surround the portion of the document that represents the element, and also give a parenthesized list of "components." The latter are elements that may or must appear in the element being described. The exact requirements on components are indicated in a manner we shall see shortly.

There are two important special cases of components:

1. `(#PCDATA)` ("parsed character data") after an element name means that element has a value that is text, and it has no elements nested within. Parsed character data may be thought of as HTML text. It can have formatting information within it, and the special characters like < must be escaped, by `<` and similar HTML codes. For instance,

   ```
   <!ELEMENT Title (#PCDATA)>
   ```

   says that between `<Title>` and `</Title>` tags a character string can appear. However, any nested tags are not part of the XML; they could be HTML, for instance.

2. The keyword `EMPTY`, with no parentheses, indicates that the element is one of those that has no matched closing tag. It has no subelements, nor does it have text as a value. For instance,

   ```
   <!ELEMENT Foo EMPTY>
   ```

   says that the only way the tag `Foo` can appear is as `<Foo/>`.

Example 11.8: In Fig. 11.6 we see a DTD for stars.[2] The DTD name and root element is `Stars`. The first element definition says that inside the matching pair of tags `<Stars>...</Stars>` we shall find zero or more `Star` elements, each representing a single star. It is the * in `(Star*)` that says "zero or more," i.e., "any number of."

The second element, `Star`, is declared to consist of three kinds of subelements: `Name`, `Address`, and `Movies`. They must appear in this order, and each must be present. However, the + following `Address` says "one or more"; that is, there can be any number of addresses listed for a star, but there must be at least one. The `Name` element is then defined to be parsed character data. The fourth element says that an address element consists of subelements for a street and a city, in that order.

Then, the `Movies` element is defined to have zero or more elements of type `Movie` within it; again, the * says "any number of." A `Movie` element is defined to consist of title and year elements, each of which are simple text. Figure 11.7 is an example of a document that conforms to the DTD of Fig. 11.6. □

```
<!DOCTYPE Stars [
    <!ELEMENT Stars (Star*)>
    <!ELEMENT Star (Name, Address+, Movies)>
    <!ELEMENT Name (#PCDATA)>
    <!ELEMENT Address (Street, City)>
    <!ELEMENT Street (#PCDATA)>
    <!ELEMENT City (#PCDATA)>
    <!ELEMENT Movies (Movie*)>
    <!ELEMENT Movie (Title, Year)>
    <!ELEMENT Title (#PCDATA)>
    <!ELEMENT Year (#PCDATA)>
]>
```

Figure 11.6: A DTD for movie stars

The components of an element E are generally other elements. They must appear between the tags $<E>$ and $</E>$ in the order listed. However, there are several operators that control the number of times elements appear.

1. A * following an element means that the element may occur any number of times, including zero times.
2. A + following an element means that the element may occur one or more times.
3. A ? following an element means that the element may occur either zero times or one time, but no more.
4. We can connect a list of options by the "or" symbol | to indicate that exactly one option appears. For example, if <Movie> elements had <Genre> subelements, we might declare these by

   ```
   <!ELEMENT Genre (Comedy|Drama|SciFi|Teen)>
   ```

   to indicate that each <Genre> element has one of these four subelements.
5. Parentheses can be used to group components. For example, if we declared addresses to have the form

   ```
   <!ELEMENT Address Street, (City|Zip)>
   ```

   then <Address> elements would each have a <Street> subelement followed by either a <City> or <Zip> subelement, but not both.

[2]Note that the stars-and-movies XML document of Fig. 11.3 is not intended to conform to this DTD.

```
<Stars>
    <Star>
        <Name>Carrie Fisher</Name>
        <Address>
            <Street>123 Maple St.</Street>
            <City>Hollywood</City>
        </Address>
        <Address>
            <Street>5 Locust Ln.</Street>
            <City>Malibu</City>
        </Address>
        <Movies>
            <Movie>
                <Title>Star Wars</Title>
                <Year>1977</Year>
            </Movie>
            <Movie>
                <Title>Empire Strikes Back</Title>
                <Year>1980</Year>
            </Movie>
            <Movie>
                <Title>Return of the Jedi</Title>
                <Year>1983</Year>
            </Movie>
        </Movies>
    </Star>
    <Star>
        <Name>Mark Hamill</Name>
        <Address>
            <Street>456 Oak Rd.<Street>
            <City>Brentwood</City>
        </Address>
        <Movies>
            <Movie>
                <Title>Star Wars</Title>
                <Year>1977</Year>
            </Movie>
            <Movie>
                <Title>Empire Strikes Back</Title>
                <Year>1980</Year>
            </Movie>
            <Movie>
                <Title>Return of the Jedi</Title>
                <Year>1983</Year>
            </Movie>
        </Movies>
    </Star>
</Stars>
```

Figure 11.7: Example of a document following the DTD of Fig. 11.6

11.3.2 Using a DTD

If a document is intended to conform to a certain DTD, we can either:

a) Include the DTD itself as a preamble to the document, or

b) In the opening line, refer to the DTD, which must be stored separately in the file system accessible to the application that is processing the document.

Example 11.9: Here is how we might introduce the document of Fig. 11.7 to assert that it is intended to conform to the DTD of Fig. 11.6.

```
<?xml version = "1.0" encoding = "utf-8" standalone = "no"?>
<!DOCTYPE Stars SYSTEM "star.dtd">
```

The attribute `standalone = "no"` says that a DTD is being used. Recall we set this attribute to `"yes"` when we did not wish to specify a DTD for the document. The location from which the DTD can be obtained is given in the `!DOCTYPE` clause, where the keyword `SYSTEM` followed by a file name gives this location. □

11.3.3 Attribute Lists

A DTD also lets us specify which attributes an element may have, and what the types of these attributes are. A declaration of the form

`<!ATTLIST` *element-name attribute-name type* `>`

says that the named attribute can be an attribute of the named element, and that the type of this attribute is the indicated type. Several attributes can be defined in one `ATTLIST` statement, but it is not necessary to do so, and the `ATTLIST` statements can appear in any position in the DTD.

The most common type for attributes is `CDATA`. This type is essentially character-string data with special characters like < escaped as in `#PCDATA`. Notice that `CDATA` does not take a pound sign as `#PCDATA` does. Another option is an enumerated type, which is a list of possible strings, surrounded by parentheses and separated by |'s. Following the data type there can be a keyword `#REQUIRED` or `#IMPLIED`, which means that the attribute must be present, or is optional, respectively.

Example 11.10: Instead of having the title and year be subelements of a `<Movie>` element, we could make these be attributes instead. Figure 11.8 shows possible attribute-list declarations. Notice that `Movie` is now an empty element. We have given it three attributes: `title`, `year`, and `genre`. The first two are `CDATA`, while the `genre` has values from an enumerated type. Note that in the document, the values, such as *comedy*, appear with quotes. Thus,

```
<Movie title = "Star Wars" year = "1977" genre = "sciFi" />
```

is a possible movie element in a document that conforms to this DTD. □

```
<!ELEMENT Movie EMPTY>
    <!ATTLIST Movie
        title CDATA #REQUIRED
        year CDATA #REQUIRED
        genre (comedy | drama | sciFi | teen) #IMPLIED
    >
```

Figure 11.8: Data about movies will appear as attributes

```
<!DOCTYPE StarMovieData [
    <!ELEMENT StarMovieData (Star*, Movie*)>
    <!ELEMENT Star (Name, Address+)>
        <!ATTLIST Star
            starId ID #REQUIRED
            starredIn IDREFS #IMPLIED
        >
    <!ELEMENT Name (#PCDATA)>
    <!ELEMENT Address (Street, City)>
    <!ELEMENT Street (#PCDATA)>
    <!ELEMENT City (#PCDATA)>
    <!ELEMENT Movie (Title, Year)>
        <!ATTLIST Movie
            movieId ID #REQUIRED
            starsOf IDREFS #IMPLIED
        >
    <!ELEMENT Title (#PCDATA)>
    <!ELEMENT Year (#PCDATA)>
]>
```

Figure 11.9: A DTD for stars and movies, using ID's and IDREF's

11.3.4 Identifiers and References

Recall from Section 11.2.5 that certain attributes can be used as identifiers for elements. In a DTD, we give these attributes the type `ID`. Other attributes have values that are references to these element ID's; these attributes may be declared to have type `IDREF`. The value of an `IDREF` attribute must also be the value of some `ID` attribute of some element, so the `IDREF` is in effect a pointer to the `ID`. An alternative is to give an attribute the type `IDREFS`. In that case, the value of the attribute is a string consisting of a list of ID's, separated by whitespace. The effect is that an `IDREFS` attribute links its element to a set of elements — the elements identified by the ID's on the list.

Example 11.11: Figure 11.9 shows a DTD in which stars and movies are given equal status, and the ID-IDREFS correspondence is used to describe the many-many relationship between movies and stars that was suggested in the semistructured data of Fig. 11.1. The structure differs from that of the DTD in Fig. 11.6, in that stars and movies have equal status; both are subelements of the root element. That is, the name of the root element for this DTD is `StarMovieData`, and its elements are a sequence of stars followed by a sequence of movies.

A star no longer has a set of movies as subelements, as was the case for the DTD of Fig. 11.6. Rather, its only subelements are a name and address, and in the beginning `<Star>` tag we shall find an attribute `starredIn` of type `IDREFS`, whose value is a list of ID's for the movies of the star.

```
<? xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<StarMovieData>
    <Star starID = "cf" starredIn = "sw">
        <Name>Carrie Fisher</Name>
        <Address>
            <Street>123 Maple St.</Street>
            <City>Hollywood</City>
        </Address>
        <Address>
            <Street>5 Locust Ln.</Street>
            <City>Malibu</City>
        </Address>
    </Star>
    <Star starID = "mh" starredIn = "sw">
        <Name>Mark Hamill</Name>
        <Address>
            <Street>456 Oak Rd.</Street>
            <City>Brentwood</City>
        </Address>
    </Star>
    <Movie movieID = "sw" starsOf = "cf mh">
        <Title>Star Wars</Title>
        <Year>1977</Year>
    </Movie>
</StarMovieData>
```

Figure 11.10: Adding stars-in information to our XML document

A `<Star>` element also has an attribute `starId`. Since it is declared to be of type `ID`, the value of `starId` may be referenced by `<Movie>` elements to indicate the stars of the movie. That is, when we look at the attribute list for `Movie` in Fig. 11.9, we see that it has an attribute `movieId` of type `ID`; these

are the ID's that will appear on lists that are the values of `starredIn` elements. Symmetrically, the attribute `starsOf` of `Movie` is an `IDREFS`, a list of ID's for stars. □

11.3.5 Exercises for Section 11.3

Exercise 11.3.1: Add to the document of Fig. 11.10 the following facts:

a) Carrie Fisher and Mark Hamill also starred in *The Empire Strikes Back* (1980) and *Return of the Jedi* (1983).

b) Harrison Ford also starred in *Star Wars*, in the two movies mentioned in (a), and the movie *Firewall* (2006).

c) Carrie Fisher also starred in *Hannah and Her Sisters* (1985).

d) Matt Damon starred in *The Bourne Identity* (2002).

Exercise 11.3.2: Suggest how typical data about banks and customers, as was described in Exercise 4.1.1, could be represented as a DTD.

Exercise 11.3.3: Suggest how typical data about players, teams, and fans, as was described in Exercise 4.1.3, could be represented as a DTD.

Exercise 11.3.4: Suggest how typical data about a genealogy, as was described in Exercise 4.1.6, could be represented as a DTD.

! **Exercise 11.3.5:** Using your representation from Exercise 11.2.2, devise an algorithm that will take any relation schema (a relation name and a list of attribute names) and produce a DTD describing a document that represents that relation.

11.4 XML Schema

XML Schema is an alternative way to provide a schema for XML documents. It is more powerful than DTD's, giving the schema designer extra capabilities. For instance, XML Schema allows arbitrary restrictions on the number of occurrences of subelements. It allows us to declare types, such as integer or float, for simple elements, and it gives us the ability to declare keys and foreign keys.

11.4.1 The Form of an XML Schema

An XML Schema description of a schema is itself an XML document. It uses the namespace at the URL:

```
http://www.w3.org/2001/XMLSchema
```

that is provided by the World-Wide-Web Consortium. Each XML-Schema document thus has the form:

```
<? xml version = "1.0" encoding = "utf-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    ...
</xs:schema>
```

The first line indicates XML, and uses the special brackets <? amd ?>. The second line is the root tag for the document that is the schema. The attribute `xmlns` (XML namespace) makes the variable `xs` stand for the namespace for XML Schema that was mentioned above. It is this namespace that causes the tag `<xs:schema>` to be interpreted as `schema` in the namespace for XML Schema. As discussed in Section 11.2.6, qualifying each XML-Schema term we use with the prefix `xs:` will cause each such tag to be interpreted according to the rules for XML Schema. Between the opening `<xs:schema>` tag and its matched closing tag `</xs:schema>` will appear a schema. In what follows, we shall learn the most important tags from the XML-Schema namespace and what they mean.

11.4.2 Elements

An important component of schemas is the *element*, which is similar to an element definition in a DTD. In the discussion that follows, you should be alert to the fact that, because XML-Schema definitions are XML documents, these schemas are themselves composed of "elements." However, the elements of the schema itself, each of which has a tag that begins with `xs:`, are not the elements being defined by the schema.[3] The form of an element definition in XML Schema is:

```
<xs:element name = element name type = element type >
   constraints and/or structure information
</xs:element>
```

The element name is the chosen tag for these elements in the schema being defined. The type can be either a simple type or a complex type. Simple types include the common primitive types, such as `xs:integer`, `xs:string`, and `xs:boolean`. There can be no subelements for an element of a simple type.

Example 11.12: Here are title and year elements defined in XML Schema:

```
<xs:element name = "Title" type = "xs:string" />
<xs:element name = "Year" type = "xs:integer" />
```

[3]To further assist in the distinction between tags that are part of a schema definition and the tags of the schema being defined, we shall begin each of the latter with a capital letter.

Each of these <xs:element> elements is itself empty, so it can be closed by /> with no matched closing tag. The first defined element has name Title and is of string type. The second element is named Year and is of type integer. In documents (perhaps talking about movies) with <Title> and <Year> elements, these elements will not be empty, but rather will be followed by a string (the title) or integer (the year), and a matched closing tag, </Title> or </Year>, respectively. □

11.4.3 Complex Types

A *complex type* in XML Schema can have several forms, but the most common is a sequence of elements. These elements are required to occur in the sequence given, but the number of repetitions of each element can be controlled by attributes minOccurs and maxOccurs, that appear in the element definitions themselves. The meanings of these attributes are as expected; no fewer than minOccurs occurrences of each element may appear in the sequence, and no more than maxOccurs occurrences may appear. If there is more than one occurrence, they must all appear consecutively. The default, if one or both of these attributes are missing, is one occurrence. To say that there is no upper limit on occurrences, use the value "unbounded" for maxOccurs.

```
<xs:complexType name = type name >
   <xs:sequence>
      list of element definitions
   </xs:sequence>
</xs:complexType>
```

Figure 11.11: Defining a complex type that is a sequence of elements

The form of a definition for a complex-type that is a sequence of elements is shown in Fig. 11.11. The name for the complex type is optional, but is needed if we are going to use this complex type as the type of one or more elements of the schema being defined. An alternative is to place the complex-type definition between an opening <xs:element> tag and its matched closing tag, to make that complex type be the type of the element.

Example 11.13: Let us write a complete XML-Schema document that defines a very simple schema for movies. The root element for movie documents will be <Movies>, and the root will have zero or more <Movie> subelements. Each <Movie> element will have two subelements: a title and year, in that order. The XML-Schema document is shown in Fig. 11.12.

Lines (1) and (2) are a typical preamble to an XML-Schema definition. In lines (3) through (8), we define a complex type, whose name is movieType. This type consists of a sequence of two elements named Title and Year; they are the elements we saw in Example 11.12. The type definition itself does not

```
 1)  <? xml version = "1.0" encoding = "utf-8" ?>
 2)  <xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">

 3)     <xs:complexType name = "movieType">
 4)        <xs:sequence>
 5)           <xs:element name = "Title" type = "xs:string" />
 6)           <xs:element name = "Year" type = "xs:integer" />
 7)        </xs:sequence>
 8)     </xs:complexType>

 9)     <xs:element name = "Movies">
10)        <xs:complexType>
11)           <xs:sequence>
12)              <xs:element name = "Movie" type = "movieType"
                     minOccurs = "0" maxOccurs = "unbounded" />
13)           </xs:sequence>
14)        </xs:complexType>
15)     </xs:element>

16)  </xs:schema>
```

Figure 11.12: A schema for movies in XML Schema

create any elements, but notice how the name `movieType` is used in line (12) to make this type be the type of `Movie` elements.

Lines (9) through (15) define the element `Movies`. Although we could have created a complex type for this element, as we did for `Movie`, we have chosen to include the type in the element definition itself. Thus, we put no type attribute in line (9). Rather, between the opening `<xs:element>` tag at line (9) and its matched closing tag at line (15) appears a complex-type definition for the element `Movies`. This complex type has no name, but it is defined at line (11) to be a sequence. In this case, the sequence has only one kind of element, `Movie`, as indicated by line (12). This element is defined to have type `movieType` — the complex type we defined at lines (3) through (8). It is also defined to have between zero and infinity occurrences. Thus, the schema of Fig. 11.12 says the same thing as the DTD we show in Fig. 11.13. □

There are several other ways we can construct a complex type.

- In place of `xs:sequence` we could use `xs:all`, which means that each of the elements between the opening `<xs:all>` tag and its matched closing tag must occur, in any order, exactly once each.
- Alternatively, we could replace `xs:sequence` by `xs:choice`. Then, exactly one of the elements found between the opening `<xs:choice>` tag

```
<!DOCTYPE Movies [
    <!ELEMENT Movies (Movie*)>
    <!ELEMENT Movie (Title, Year)>
    <!ELEMENT Title (#PCDATA)>
    <!ELEMENT Year (#PCDATA)>
]>
```

Figure 11.13: A DTD for movies

and its matched closing tag will appear.

The elements inside a sequence or choice can have `minOccurs` and `maxOccurs` attributes to govern how many times they can appear. In the case of a choice, only one of the elements can appear at all, but it can appear more than once if it has a value of `maxOccurs` greater than 1. The rules for `xs:all` are different. It is not permitted to have a `maxOccurs` value other than 1, but `minOccurs` can be either 0 or 1. In the former case, the element might not appear at all.

11.4.4 Attributes

A complex type can have attributes. That is, when we define a complex type T, we can include instances of element `<xs:attribute>`. When we use T as the type of an element E, then E can have (or must have) an instance of this attribute. The form of an attribute definition is:

`<xs:attribute name =` *attribute name* `type =` *type name*
other information about the attribute `/>`

The "other information" may include information such as a default value and usage (required or optional — the latter is the default).

Example 11.14: The notation

```
<xs:attribute name = "year" type = "xs:integer"
    default = "0" />
```

defines `year` to be an attribute of type integer. We do not know of what element `year` is an attribute; it depends where the above definition is placed. The default value of `year` is 0, meaning that if an element without a value for attribute `year` occurs in a document, then the value of `year` is taken to be 0.

As another instance:

```
<xs:attribute name = "year" type = "xs:integer"
    use = "required" />
```

is another definition of the attribute year. However, setting use to required means that any element of the type being defined must have a value for attribute year. □

Attribute definitions are placed within a complex-type definition. In the next example, we rework Example 11.13 by making the type movieType have attributes for the title and year, rather than subelements for that information.

```
1)  <? xml version = "1.0" encoding = "utf-8" ?>
2)  <xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">

3)      <xs:complexType name = "movieType">
4)          <xs:attribute name = "title" type = "xs:string"
                use = "required" />
5)          <xs:attribute name = "year" type = "xs:integer"
                use = "required" />
6)      </xs:complexType>

7)      <xs:element name = "Movies">
8)          <xs:complexType>
9)              <xs:sequence>
10)                 <xs:element name = "Movie" type = "movieType"
                        minOccurs = "0" maxOccurs = "unbounded" />
11)             </xs:sequence>
12)         </xs:complexType>
13)     </xs:element>

14) </xs:schema>
```

Figure 11.14: Using attributes in place of simple elements

Example 11.15: Figure 11.14 shows the revised XML Schema definition. At lines (4) and (5), the attributes title and year are defined to be required attributes for elements of type movieType. When element Movie is given that type at line (10), we know that every <Movie> element must have values for title and year. Figure 11.15 shows the DTD resembling Fig. 11.14. □

11.4.5 Restricted Simple Types

It is possible to create a restricted version of a simple type such as integer or string by limiting the values the type can take. These types can then be used as the type of an attribute or element. We shall consider two kinds of restrictions here:

```
<!DOCTYPE Movies [
    <!ELEMENT Movies (Movie*)>
    <!ELEMENT Movie EMPTY>
        <!ATTLIST Movie
            title CDATA #REQUIRED
            year  CDATA #REQUIRED
        >
]>
```

Figure 11.15: DTD equivalent for Fig. 11.14

1. Restricting numerical values by using `minInclusive` to state the lower bound, `maxInclusive` to state the upper bound.[4]

2. Restricting values to an enumerated type.

The form of a range restriction is shown in Fig. 11.16. The restriction has a base, which may be a primitive type (e.g., `xs:string`) or another simple type.

```
<xs:simpleType name = type name >
  <xs:restriction base = base type >
    upper and/or lower bounds
  </xs:restriction>
</xs:simpleType>
```

Figure 11.16: Form of a range restriction

Example 11.16: Suppose we want to restrict the year of a movie to be no earlier than 1915. Instead of using `xs:integer` as the type for element `Year` in line (6) of Fig. 11.12 or for the attribute `year` in line (5) of Fig. 11.14, we could define a new simple type as in Fig. 11.17. The type `movieYearType` would then be used in place of `xs:integer` in the two lines cited above. □

Our second way to restrict a simple type is to provide an enumeration of values. The form of a single enumerated value is:

```
<xs:enumeration value = some value />
```

A restriction can consist of any number of these values.

[4]The "inclusive" means that the range of values includes the given bound. An alternative is to replace `Inclusive` by `Exclusive`, meaning that the stated bounds are just outside the permitted range.

```
<xs:simpleType name = "movieYearType">
    <xs:restriction base = "xs:integer">
        <xs:minInclusive value = "1915" />
    </xs:restriction>
<xs:simpleType>
```

Figure 11.17: A type that restricts integer values to be 1915 or greater

Example 11.17: Let us design a simple type suitable for the genre of movies. In our running example, we have supposed that there are only four possible genres: comedy, drama, sciFi, and teen. Figure 11.18 shows how to define a type `genreType` that could serve as the type for an element or attribute representing our genres of movies. □

```
<xs:simpleType name = "genreType">
    <xs:restriction base = "xs:string">
        <xs:enumeration value = "comedy" />
        <xs:enumeration value = "drama" />
        <xs:enumeration value = "sciFi" />
        <xs:enumeration value = "teen" />
    </xs:restriction>
<xs:simpleType>
```

Figure 11.18: A enumerated type in XML Schema

11.4.6 Keys in XML Schema

An element can have a key declaration, which says that when we look at a certain class C of elements, values of one or more given *fields* within those elements are unique. The concept of "field" is actually quite general, but the most common case is for a field to be either a subelement or an attribute. The class C of elements is defined by a "selector." Like fields, selectors can be complex, but the most common case is a sequence of one or more element names, each a subelement of the one before it. In terms of a tree of semistructured data, the class is all those nodes reachable from a given node by following a particular sequence of arc labels.

Example 11.18: Suppose we want to say, about the semistructured data in Fig. 11.1, that among all the nodes we can reach from the root by following a *star* label, what we find following a further *name* label leads us to a unique value. Then the "selector" would be *star* and the "field" would be *name*. The implication of asserting this key is that within the root element shown, there

cannot be two stars with the same name. If movies had names instead of titles, then the key assertion would not prevent a movie and a star from having the same name. Moreover, if there were actually many elements like the tree of Fig. 11.1 found in one document (e.g., each of the objects we called "Root" in that figure were actually a single movie and its stars), then different trees could have the same star name without violating the key constraint. □

The form of a key declaration is

```
<xs:key name = key name >
   <xs:selector xpath = path description >
   <xs:field xpath = path description >
</xs:key>
```

There can be more than one line with an `xs:field` element, in case several fields are needed to form the key. An alternative is to use the element `xs:unique` in place of `xs:key`. The difference is that if "key" is used, then the fields must exist for each element defined by the selector. However, if "unique" is used, then they might not exist, and the constraint is only that they are unique if they exist.

The selector path can be any sequence of elements, each of which is a subelement of the previous. The element names are separated by slashes. The field can be any subelement of the last element on the selector path, or it can be an attribute of that element. If it is an attribute, then it is preceded by the "at-sign." There are other options, and in fact, the selector and field can be any XPath expressions; we take up the XPath query language in Section 12.1.

Example 11.19: In Fig. 11.19 we see an elaboration of Fig. 11.12. We have added the element `Genre` to the definition of `movieType`, in order to have a nonkey subelement for a movie. Lines (3) through (10) define `genreType` as in Example 11.17. The `Genre` subelement of `movieType` is added at line (15).

The definition of the `Movies` element has been changed in lines (24) through (28) by the addition of a key. The name of the key is `movieKey`; this name will be used if it is referenced by a foreign key, as we shall discuss in Section 11.4.7. Otherwise, the name is irrelevant. The selector path is just `Movie`, and there are two fields, `Title` and `Year`. The meaning of this key declaration is that, within any `Movies` element, among all its `Movie` subelements, no two can have both the same title and the same year, nor can any of these values be missing. Note that because of the way `movieType` was defined at lines (13) and (14), with no values for `minOccurs` or `maxOccurs` for `Title` or `Year`, the defaults, 1, apply, and there must be exactly one occurrence of each. □

11.4.7 Foreign Keys in XML Schema

We can also declare that an element has, perhaps deeply nested within it, a field or fields that serve as a reference to the key for some other element. This

```
1)  <? xml version = "1.0" encoding = "utf-8" ?>
2)  <xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">

3)  <xs:simpleType name = "genreType">
4)     <xs:restriction base = "xs:string">
5)        <xs:enumeration value = "comedy" />
6)        <xs:enumeration value = "drama" />
7)        <xs:enumeration value = "sciFi" />
8)        <xs:enumeration value = "teen" />
9)     </xs:restriction>
10) <xs:simpleType>

11)    <xs:complexType name = "movieType">
12)       <xs:sequence>
13)          <xs:element name = "Title" type = "xs:string" />
14)          <xs:element name = "Year" type = "xs:integer" />
15)          <xs:element name = "Genre" type = "genreType"
                minOccurs = "0" maxOccurs = "1" />
16)       </xs:sequence>
17)    </xs:complexType>

18)    <xs:element name = "Movies">
19)       <xs:complexType>
20)          <xs:sequence>
21)             <xs:element name = "Movie" type = "movieType"
                    minOccurs = "0" maxOccurs = "unbounded" />
22)          </xs:sequence>
23)       </xs:complexType>
24)       <xs:key name = "movieKey">
25)          <xs:selector xpath = "Movie" />
26)          <xs:field xpath = "Title" />
27)          <xs:field xpath = "Year" />
28)       </xs:key>
29)    </xs:element>

30) </xs:schema>
```

Figure 11.19: A schema for movies in XML Schema

capability is similar to what we get with ID's and IDREF's in a DTD (see Section 11.3.4). However, the latter are untyped references, while references in XML Schema are to particular types of elements. The form of a foreign-key definition in XML Schema is:

```
<xs:keyref name = foreign-key name refer = key name >
   <xs:selector xpath = path description >
   <xs:field xpath = path description >
</xs:keyref>
```

The schema element is `xs:keyref`. The foreign-key itself has a name, and it refers to the name of some key or unique value. The selector and field(s) are as for keys.

Example 11.20: Figure 11.20 shows the definition of an element `<Stars>`. We have used the style of XML Schema where each complex type is defined within the element that uses it. Thus, we see at lines (4) through (6) that a `<Stars>` element consists of one or more `<Star>` subelements.

At lines (7) through (11), we see that each `<Star>` element has three kinds of subelements. There is exactly one `<Name>` and one `<Address>` subelement, and any number of `<StarredIn>` subelements. In lines (12) through (15), we find that a `<StarredIn>` element has no subelements, but it does have two attributes, `title` and `year`.

Lines (22) through (26) define a foreign key. In line (22) we see that the name of this foreign-key constraint is `movieRef` and that it refers to the key `movieKey` that was defined in Fig. 11.19. Notice that this foreign key is defined within the `<Stars>` definition. The selector is `Star/StarredIn`. That is, it says we should look at every `<StarredIn>` subelement of every `<Star>` subelement of a `<Stars>` element. From that `<StarredIn>` element, we extract the two fields `title` and `year`. The `@` indicates that these are attributes rather than subelements. The assertion made by this foreign-key constraint is that any title-year pair we find in this way will appear in some `<Movie>` element as the pair of values for its subelements `<Title>` and `<Year>`. □

11.4.8 Exercises for Section 11.4

Exercise 11.4.1: Give an example of a document that conforms to the XML Schema definition of Fig. 11.12 and an example of one that has all the elements mentioned, but does not conform to the definition.

Exercise 11.4.2: Rewrite Fig. 11.12 so that there is a named complex type for `Movies`, but no named type for `Movie`.

Exercise 11.4.3: Write the XML Schema definitions of Fig. 11.19 and 11.20 as a DTD.

```
1)  <? xml version = "1.0" encoding = "utf-8" ?>
2)  <xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">

3)  <xs:element name = "Stars">

4)     <xs:complexType>
5)        <xs:sequence>
6)           <xs:element name = "Star" minOccurs = "1"
                   maxOccurs = "unbounded">
7)              <xs:complexType>
8)                 <xs:sequence>
9)                    <xs:element name = "Name"
                         type = "xs;string" />
10)                   <xs:element name = "Address"
                         type = "xs:string" />
11)                   <xs:element name = "StarredIn"
                            minOccurs = "0"
                            maxOccurs = "unbounded">
12)                      <xs:complexType>
13)                         <xs:attribute name = "title"
                               type = "xs:string" />
14)                         <xs:attribute name = "year"
                               type = "xs:integer" />
15)                      </xs:complexType>
16)                   </xs:element>
17)                </xs:sequence>
18)             </xs:complexType>
19)          </xs:element>
20)       </xs:sequence>
21)    </xs:complexType>

22)    <xs:keyref name = "movieRef" refers = "movieKey">
23)       <xs:selector xpath = "Star/StarredIn" />
24)       <xs:field  xpath = "@title" />
25)       <xs:field  xpath = "@year" />
26)    </xs:keyref>

27) </xs:element>
```

Figure 11.20: Stars with a foreign key

11.5 Summary of Chapter 11

- ✦ *Semistructured Data*: In this model, data is represented by a graph. Nodes are like objects or values of attributes, and labeled arcs connect an object to both the values of its attributes and to other objects to which it is connected by a relationship.

- ✦ *XML*: The Extensible Markup Language is a World-Wide-Web Consortium standard that representes semistructured data linearly.

- ✦ *XML Elements*: Elements consist of an opening tag `<Foo>`, a matched closing tag `</Foo>`, and everything between them. What appears can be text, or it can be subelements, nested to any depth.

- ✦ *XML Attributes*: Tags can have attribute-value pairs within them. These attributes provide additional information about the element with which they are associated.

- ✦ *Document Type Definitions*: The DTD is a simple, grammatical form of defining elements and attributes of XML, thus providing a rudimentary schema for those XML documents that use the DTD. An element is defined to have a sequence of subelements, and these elements can be required to appear exactly once, at most once, at least once, or any number of times. An element can also be defined to have a list of required and/or optional attributes.

- ✦ *Identifiers and References in DTD's*: To represent graphs that are not trees, a DTD allows us to declare attributes of type `ID` and `IDREF(S)`. An element can thus be given an identifier, and that identifier can be referred to by other elements from which we would like to establish a link.

- ✦ *XML Schema*: This notation is another way to define a schema for certain XML documents. XML Schema definitions are themselves written in XML, using a set of tags in a namespace that is provided by the World-Wide-Web Consortium.

- ✦ *Simple Types in XML Schema*: The usual sorts of primitive types, such as integers and strings, are provided. Additional simple types can be defined by restricting a simple type, such as by providing a range for values or by giving an enumeration of permitted values.

- ✦ *Complex Types in XML Schema*: Structured types for elements may be defined to be sequences of elements, each with a minimum and maximum number of occurrences. Attributes of an element may also be defined in its complex type.

- ✦ *Keys and Foreign Keys in XML Schema*: A set of elements and/or attributes may be defined to have a unique value within the scope of some

enclosing element. Other sets of elements and/or attributes may be defined to have a value that appears as a key within some other kind of element.

11.6 References for Chapter 11

Semistructured data as a data model was first studied in [5] and [4]. LOREL, the prototypical query language for this model is described in [3]. Surveys of work on semistructured data include [1], [7], and the book [2].

XML is a standard developed by the World-Wide-Web Consortium. The home page for information about XML is [9]. References on DTD's and XML Schema are also found there. For XML parsers, the definition of DOM is in [8] and for SAX it is [6]. A useful place to go for quick tutorials on many of these subjects is [10].

1. S. Abiteboul, "Querying semi-structured data," *Proc. Intl. Conf. on Database Theory* (1997), Lecture Notes in Computer Science 1187 (F. Afrati and P. Kolaitis, eds.), Springer-Verlag, Berlin, pp. 1–18.

2. S. Abiteboul, D. Suciu, and P. Buneman, *Data on the Web: From Relations to Semistructured Data and XML*, Morgan-Kaufmann, San Francisco, 1999.

3. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Weiner, "The LOREL query language for semistructured data," In *J. Digital Libraries* 1:1, 1997.

4. P. Buneman, S. B. Davidson, and D. Suciu, "Programming constructs for unstructured data," *Proceedings of the Fifth International Workshop on Database Programming Languages*, Gubbio, Italy, Sept., 1995.

5. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom, "Object exchange across heterogeneous information sources," *IEEE Intl. Conf. on Data Engineering*, pp. 251–260, March 1995.

6. Sax Project, `http://www.saxproject.org/`

7. D. Suciu (ed.) Special issue on management of semistructured data, *SIGMOD Record* **26**:4 (1997).

8. World-Wide-Web Consortium, `http://www.w3.org/DOM/`

9. World-Wide-Web Consortium, `http://www.w3.org/XML/`

10. W3 Schools, `http://www.w3schools.com`

Chapter 12

Programming Languages for XML

We now turn to programming languages for semistructured data. All the widely used languages of this type apply to XML data, and might be used for semistructured data represented in other ways as well. In this chapter, we shall study three such languages. The first, XPath, is a simple language for describing sets of similar paths in a graph of semistructured data. XQuery is an extension of XPath that adopts something of the style of SQL. It allows iterations over sets, subqueries, and many other features that will be familiar from the study of SQL.

The third topic of this chapter is XSLT. This language was developed originally as a transformation language, capable of restructuring XML documents or turning them into printable (HTML) documents. However, its expressive power is actually quite similar to that of XQuery, and it is capable of producing XML results. Thus, it can serve as a query language for XML.

12.1 XPath

In this section, we introduce XPath. We begin with a discussion of the data model used in the most recent version of XPath, called XPath 2.0; this model is used in XQuery as well. This model plays a role analogous to the "bag of tuples of primitive-type components" that is used in the relational model as the value of a relation.

In later sections, we learn about XPath path expressions and their meaning. In general, these expressions allow us to move from elements of a document to some or all of their subelements. Using "axes," we are also able to move within documents in a variety of ways, and to obtain the attributes of elements.

12.1.1 The XPath Data Model

As in the relational model, XPath assumes that all values — those it produces and those constructed in intermediate steps — have the same general "shape." In the relational model, this "shape" is a bag of tuples. Tuples in a given bag all have the same number of components, and the components each have a primitive type, e.g., integer or string. In XPath, the analogous "shape" is *sequence of items*. An *item* is either:

1. A value of primitive type: integer, real, boolean, or string, for example.
2. A *node*. There are many kinds of nodes, but in our introduction, we shall only talk about three kinds:
   (a) *Documents*. These are files containing an XML document, perhaps denoted by their local path name or a URL.
   (b) *Elements*. These are XML elements, including their opening tags, their matched closing tag if there is one, and everything in between (i.e., below them in the tree of semistructured data that an XML document represents).
   (c) *Attributes*. These are found inside opening tags, as we discussed in several places in Chapter 11.

The items in a sequence need not be all of the same type, although often they will be.

Example 12.1: Figure 12.1 is a sequence of four items. The first is the integer 10; the second is a string, and the third is a real. These are all items of primitive type.

```
10

"ten"

10.0

<Number base = "8">
    <Digit>1</Digit>
    <Digit>2</Digit>
</Number>

@val="10"
```

Figure 12.1: A sequence of five items

The fourth item is a node, and this node's type is "element." Notice that the element has tag `Number` with an attribute and two subelements with tag `Digit`. The last item is an attribute node. □

12.1.2 Document Nodes

While the documents to which XPath is applied can come from various sources, it is common to apply XPath to documents that are files. We can make a document node from a file by applying the function:

doc(*file name*)

The named file should be an XML document. We can name a file either by giving its local name or a URL if it is remote. Thus, examples of document nodes include:

```
doc("movies.xml")
doc("/usr/sally/data/movies.xml")
doc("infolab.stanford.edu/~hector/movies.xml")
```

Every XPath query refers to a document. In many cases, this document will be apparent from the context. For example, recall our discussion of XML-Schema keys in Section 11.4.6. We used XPath expressions to denote the selector and field(s) for a key. In that context, the document was "whatever document the schema definition is being applied to."

12.1.3 Path Expressions

Typically, an XPath expression starts at the root of a document and gives a sequence of tags and slashes (/), say $/T_1/T_2/\cdots/T_n$. We evaluate this expression by starting with a sequence of items consisting of one node: the document. We then process each of $T_1, T_2, \ldots$ in turn. To process T_i, consider the sequence of items that results from processing the previous tags, if any. Examine those items, in order, and find for each all its subelements whose tag is T_i. Those items are appended to the output sequence, in the order in which they appear in the document.

As a special case, the root tag T_1 for the document is considered a "subelement" of the document node. Thus, the expression $/T_1$ produces a sequence of one item, which is an element node consisting of the entire contents of the document. The difference may appear subtle; before we applied the expression $/T_1$, we had a document node representing the file, and after applying $/T_1$ to that node we have an element node representing the text in the file.

Example 12.2: Suppose our document is a file containing the XML text of Fig. 11.5, which we reproduce here as Fig. 12.2. The path expression `/StarMovieData` produces the sequence of one element. This element has tag `<StarMovieData>`, of course, and it consists of everything in Fig. 12.2 except for line (1).

Now, consider the path expression

```
/StarMovieData/Star/Name
```

```
 1)  <? xml version="1.0" encoding="utf-8" standalone="yes" ?>
 2)  <StarMovieData>
 3)      <Star starID = "cf" starredIn = "sw">
 4)          <Name>Carrie Fisher</Name>
 5)          <Address>
 6)              <Street>123 Maple St.</Street>
 7)              <City>Hollywood</City>
 8)          </Address>
 9)          <Address>
10)              <Street>5 Locust Ln.</Street>
11)              <City>Malibu</City>
12)          </Address>
13)      </Star>
14)      <Star starID = "mh" starredIn = "sw">
15)          <Name>Mark Hamill</Name>
16)          <Street>456 Oak Rd.</Street>
17)          <City>Brentwood</City>
18)      </Star>
19)      <Movie movieID = "sw" starsOf = "cf", "mh">
20)          <Title>Star Wars</Title>
21)          <Year>1977</Year>
22)      </Movie>
23)  </StarMovieData>
```

Figure 12.2: An XML document for applying path expressions

When we apply the `StarMovieData` tag to the sequence consisting of the document, we get the sequence consisting of the root element, as discussed above. Next, we apply to this sequence the tag `Star`. There are two subelements of the `StarMovieData` element that have tag `Star`. These are lines (3) through (12) for star Carrie Fisher and lines (14) through (18) for star Mark Hamill. Thus, the result of the path expression `/StarMovieData/Star` is the sequence of these two elements, in that order.

Finally, we apply to this sequence the tag `Name`. The first element has one `Name` subelement, at line (4). The second element also has one `Name` subelement, at line (15). Thus, the sequence

```
<Name>Carrie Fisher</Name>
<Name>Mark Hamill</Name>
```

is the result of applying the path expression `/StarMovieData/Star/Name` to the document of Fig. 12.2. □

12.1.4 Relative Path Expressions

In several contexts, we shall use XPath expressions that are *relative* to the current node or sequence of nodes.

- In Section 11.4.6 we talked about selector and field values that were really XPath expressions relative to a node or sequence of nodes for which we were defining a key.
- In Example 12.2 we talked about applying the XPath expression `Star` to the element consisting of the entire document, or the expression `Name` to a sequence of `Star` elements.

Relative expressions do not start with a slash. Each such expression must be applied in some context, which will be clear from its use. The similarity to the way files and directories are designated in a UNIX file system is not accidental.

12.1.5 Attributes in Path Expressions

Path expressions allow us to find all the elements within a document that are reached from the root along a particular kind of path (a sequence of tags). Sometimes, we want to find not these elements but rather the values of an attribute of those elements. If so, we can end the path expression by an attribute name preceded by an at-sign. That is, the path-expression form is $/T_1/T_2/\cdots/T_n/@A$.

The result of this expression is computed by first applying the path expression $/T_1/T_2/\cdots/T_n$ to get a sequence of elements. We then look at the opening tag of each element, in turn, to find an attribute A. If there is one, then the value of that attribute is appended to the sequence that forms the result.

Example 12.3: The path expression

```
/StarMovieData/Star/@starID
```

applied to the document of Fig. 12.2 finds the two `Star` elements and looks into their opening tags at lines (3) and (14) to find the values of their `starID` attributes. Both elements have this attribute, so the result sequence is `"cf" "mh"`. □

12.1.6 Axes

So far, we have only navigated through semistructured-data graphs in two ways: from a node to its children or to an attribute. XPath in fact provides a large number of *axes*, which are modes of navigation. Two of these axes are *child* (the default axis) and *attribute*, for which `@` is really a shorthand. At each step in a path expression, we can prefix a tag or attribute name by an axis name and a double-colon. For example,

```
/StarMovieData/Star/@starID
```

is really shorthand for:

```
/child::StarMovieData/child::Star/attribute::starID
```

Some of the other axes are parent, ancestor (really a proper ancestor), descendant (a proper descendant), next-sibling (any sibling to the right), previous-sibling (any sibling to the left), self, and descendant-or-self. The latter has a shorthand // and takes us from a sequence of elements to those elements and all their subelements, at any level of nesting.

Example 12.4: It might look hard to find, in the document of Fig. 12.2, all the cities where stars live. The problem is that Mark Hamill's city is not nested within an `Address` element, so it is not reached along the same paths as Carrie Fisher's cities. However, the path expression

```
//City
```

finds all the `City` subelements, at any level of nesting, and returns them in the order in which they appear in the document. That is, the result of this path expression is the sequence:

```
<City>Hollywood</City>
<City>Malibu</City>
<City>Brentwood</City>
```

which we obtain from lines (7), (11), and (17), respectively.

We could also use the // axis within the path expression. For example, should the document contain city information that wasn't about stars (e.g., studios and their addresses), then we could restrict the paths that we consider to make sure that the city was a subelement of a `Star` element. For the given document, the path expression

```
/StarMovieData/Star//City
```

produces the same three `City` elements as a result. □

Some of the other axes have shorthands as well. For example, .. stands for parent, and . for self. We have already seen @ for attribute and / for child.

12.1.7 Context of Expressions

In order to understand the meaning of an axis like parent, we need to explore further the view of data in XPath. Results of expressions are sequences of elements or primitive values. However, XPath expressions and their results do not exist in isolation; if they did, it would not make sense to ask for the "parent" of an element. Rather, there is normally a context in which the expression is

evaluated. In all our examples, there is a single document from which elements are extracted. If we think of an element in the result of some XPath expression as a reference to the element in the document, then it makes sense to apply axes like parent, ancestor, or next-sibling to the element in the sequence.

For example, we mentioned in Section 11.4.6 that keys in XML Schema are defined by a pair of XPath expressions. Key constraints apply to XML documents that obey the schema that includes the constraint. Each such document provides the context for the XPath expressions in the schema itself. Thus, it is permitted to use all the XPath axes in these expressions.

12.1.8 Wildcards

Instead of specifying a tag along every step of a path, we can use a * to say "any tag." Likewise, instead of specifying an attribute, @* says "any attribute."

Example 12.5: Consider the path expression

```
/StarMovieData/*/@*
```

applied to the document of Fig. 12.2. First, `/StarMovieData/*` takes us to every subelement of the root element. There are three: two stars and a movie. Thus, the result of this path expression is the sequence of elements in lines (3) through (13), (14) through (18), and (19) through (22).

However, the expression asks for the values of all the attributes of these elements. We therefore look for attributes among the outermost tags of each of these elements, and return their values in the order in which they appear in the document. Thus, the sequence

```
"cf" "sw" "mh" "sw" "sw" "cf" "mh"
```

is the result of the XPath query.

A subtle point is that the value of the `starsOf` attribute in line (19) is itself a sequence of items — strings `"cf"` and `"mh"`. XPath expands sequences that are part of other sequences, so all items are at the "top level," as we showed above. That is, a sequence of items is not itself an item. □

12.1.9 Conditions in Path Expressions

As we evaluate a path expression, we can restrict ourselves to follow only a subset of the paths whose tags match the tags in the expression. To do so, we follow a tag by a condition, surrounded by square brackets. This condition can be anything that has a boolean value. Values can be compared by comparison operators such as = or >=. "Not equal" is represented as in C, by !=. A compound condition can be constructed by connecting comparisons with operators or or and.

The values compared can be path expressions, in which case we are comparing the sequences returned by the expressions. Comparisons have an implied

"there exists" sense; two sequences are related if any pair of items, one from each sequence, are related by the given comparison operator. An example should make this concept clear.

Example 12.6: The following path expression:

```
/StarMovieData/Star[//City = "Malibu"]/Name
```

returns the names of the movie stars who have at least one home in Malibu. To begin, the path expression `/StarMovieData/Star` returns a sequence of all the `Star` elements. For each of these elements, we need to evaluate the truth of the condition `//City = "Malibu"`. Here, `//City` is a path expression, but it, like any path expression in a condition, is evaluated relative to the element to which the condition is applied. That is, we interpret the expression assuming that the element were the entire document to which the path expression is applied.

We start with the element for Carrie Fisher, lines (3) through (13) of Fig. 12.2. The expression `//City` causes us to look for all subelements, nested zero or more levels deep, that have a `City` tag. There are two, at lines (7) and (11). The result of the path expression `//City` applied to the Carrie-Fisher element is thus the sequence:

```
<City>Hollywood</City>
<City>Malibu</City>
```

Each item in this sequence is compared with the value `"Malibu"`. An element whose type is a primitive value such as a string can be equated to that string, so the second item passes the test. As a result, the entire `Star` element of lines (3) through (13) satisfies the condition.

When we apply the condition to the second item, lines (14) through (18) for Mark Hamill, we find a `City` subelement, but its value does not match `"Malibu"` and this element fails the condition. Thus, only the Carrie-Fisher element is in the result of the path expression

```
/StarMovieData/Star[//City = "Malibu"]
```

We have still to finish the XPath query by applying to this sequence of one element the continuation of the path expression, `/Name`. At this stage, we search for a `Name` subelement of the Carrie-Fisher element and find it at line (4). Consequently, the query result is the sequence of one element, `<Name>Carrie Fisher</Name>`. □

Several other useful forms of condition are:

- An integer $[i]$ by itself is true only when applied the ith child of its parent.
- A tag $[T]$ by itself is true only for elements that have one or more subelements with tag T.

- Similarly, an attribute [A] by itself is true only for elements that have a value for the attribute A.

Example 12.7: Figure 12.3 is a variant of our running movie example, in which we have grouped all the movies with a common title as one `Movie` element, with subelements that have tag `Version`. The title is an attribute of the movie, and the year is an attribute of the version. Versions have `Star` subelements. Consider the XPath query, applied to this document:

```
/Movies/Movie/Version[1]/@year
```

It asks for the year in which the first version of each movie was made, and the result is the sequence `"1933" "1984"`.

```
 1)  <? xml version="1.0" encoding="utf-8" standalone="yes" ?>
 2)  <Movies>
 3)      <Movie title = "King Kong">
 4)          <Version year = "1933">
 5)              <Star>Fay Wray</Star>
 6)          </Version>
 7)          <Version year = "1976">
 8)              <Star>Jeff Bridges</Star>
 9)              <Star>Jessica Lange</Star>
10)          </Version>
11)          <Version year = "2005" />
12)      </Movie>
13)      <Movie title = "Footloose">
14)          <Version year = "1984">
15)              <Star>Kevin Bacon</Star>
16)              <Star>John Lithgow</Star>
17)              <Star>Sarah Jessica Parker</Star>
18)          </Version>
19)      </Movie>
20)  </Movies>
```

Figure 12.3: An XML document for applying path expressions

In more detail, there are four `Version` elements that match the path

```
/Movies/Movie/Version
```

These are at lines (4) through (6), (7) through (10), line (11), and lines (14) through (18), respectively. Of these, the first and last are the first children of their respective parents. The `year` attributes for these versions are 1933 and 1984, respectively. □

Example 12.8: The XPath query:

```
/Movies/Movie/Version[Star]
```

applied to the document of Fig. 12.3 returns three `Version` elements. The condition `[Star]` is interpreted as "has at least one `Star` subelement." That condition is true for the `Version` elements of lines (4) through (6), (7) through (10), and (14) through (18); it is false for the element of line (11). □

```
<Products>
    <Maker name = "A">
        <PC model = "1001" price = "2114">
            <Speed>2.66</Speed>
            <RAM>1024</RAM>
            <HardDisk>250</HardDisk>
        </PC>
        <PC model = "1002" price = "995">
            <Speed>2.10</Speed>
            <RAM>512</RAM>
            <HardDisk>250</HardDisk>
        </PC>
        <Laptop model = "2004" price = "1150">
            <Speed>2.00</Speed>
            <RAM>512</RAM>
            <HardDisk>60</HardDisk>
            <Screen>13.3</Screen>
        </Laptop>
        <Laptop model = "2005" price = "2500">
            <Speed>2.16</Speed>
            <RAM>1024</RAM>
            <HardDisk>120</HardDisk>
            <Screen>17.0</Screen>
        </Laptop>
    </Maker>
```

Figure 12.4: XML document with product data — beginning

12.1.10 Exercises for Section 12.1

Exercise 12.1.1: Figures 12.4 and 12.5 are the beginning and end, respectively, of an XML document that contains some of the data from our running products exercise. Write the following XPath queries. What is the result of each?

```
    <Maker name = "E">
        <PC model = "1011" price = "959">
            <Speed>1.86</Speed>
            <RAM>2048</RAM>
            <HardDisk>160</HardDisk>
        </PC>
        <PC model = "1012" price = "649">
            <Speed>2.80</Speed>
            <RAM>1024</RAM>
            <HardDisk>160</HardDisk>
        </PC>
        <Laptop model = "2001" price = "3673">
            <Speed>2.00</Speed>
            <RAM>2048</RAM>
            <HardDisk>240</HardDisk>
            <Screen>20.1</Screen>
        </Laptop>
        <Printer model = "3002" price = "239">
            <Color>false</Color>
            <Type>laser</Type>
        </Printer>
    <Maker name = "H">
        <Printer model = "3006" price = "100">
            <Color>true</Color>
            <Type>ink-jet</Type>
        </Printer>
        <Printer model = "3007" price = "200">
            <Color>true</Color>
            <Type>laser</Type>
        </Printer>
    </Maker>
</Products>
```

Figure 12.5: XML document with product data — end

a) Find the amount of RAM on each PC.

b) Find the price of each product of any kind.

c) Find all the printer elements.

! d) Find the makers of laser printers.

! e) Find the makers of PC's and/or laptops.

f) Find the model numbers of PC's with a hard disk of at least 200 gigabytes.

!! g) Find the makers of at least two PC's.

Exercise 12.1.2: The document of Fig. 12.6 contains data similar to that used in our running battleships exercise. In this document, data about ships is nested within their class element, and information about battles appears inside each ship element. Write the following queries in XPath. What is the result of each?

a) Find the names of all ships.

b) Find all the `Class` elements for classes with a displacement larger than 35000.

c) Find all the `Ship` elements for ships that were launched before 1917.

d) Find the names of the ships that were sunk.

! e) Find the years in which ships having the same name as their class were launched.

! f) Find the names of all ships that were in battles.

!! g) Find the `Ship` elements for all ships that fought in two or more battles.

12.2 XQuery

XQuery is an extension of XPath that has become a standard for high-level querying of databases containing data in XML form. This section will introduce some of the important capabilities of XQuery.

```
<Ships>
    <Class name = "Kongo" type = "bc" country = "Japan"
            numGuns = "8" bore = "14" displacement = "32000">
        <Ship name = "Kongo" launched = "1913" />
        <Ship name = "Hiei" launched = "1914" />
        <Ship name = "Kirishima" launched = "1915">
            <Battle outcome = "sunk">Guadalcanal</Battle>
        </Ship>
        <Ship name = "Haruna" launched = "1915" />
    </Class>
    <Class name = "North Carolina" type = "bb" country = "USA"
            numGuns = "9" bore = "16" displacement = "37000">
        <Ship name = "North Carolina" launched = "1941" />
        <Ship name = "Washington" launched = "1941">
            <Battle outcome = "ok">Guadalcanal</Battle>
        </Ship>
    </Class>
    <Class name = "Tennessee" type = "bb" country = "USA"
            numGuns = "12" bore = "14" displacement = "32000">
        <Ship name = "Tennessee" launched = "1920">
            <Battle outcome = "ok">Surigao Strait</Battle>
        </Ship>
        <Ship name = "California" launched = "1921">
            <Battle outcome = "ok">Surigao Strait</Battle>
    </Class>
    <Class name = "King George V" type = "bb"
            country = "Great Britain"
            numGuns = "10" bore = "14" displacement = "32000">
        <Ship name = "King George V" launched = "1940" />
        <Ship name = "Prince of Wales" launched = "1941">
            <Battle outcome = "damaged">Denmark Strait</Battle>
            <Battle outcome = "sunk">Malaya</Battle>
        </Ship>
        <Ship name = "Duke of York" launched = "1941">
            <Battle outcome = "ok">North Cape</Battle>
        </Ship>
        <Ship name = "Howe" launched = "1942" />
        <Ship name = "Anson" launched = "1942" />
    </Class>
</Ships>
```

Figure 12.6: XML document containing battleship data

Case Sensitivity of XQuery

XQuery is case sensitive. Thus, keywords such as `let` or `for` need to be written in lower case, just like keywords in C or Java.

12.2.1 XQuery Basics

XQuery uses the same model for values that we introduced for XPath in Section 12.1.1. That is, all values produced by XQuery expressions are sequences of items. Items are either primitive values or nodes of various types, including elements, attributes, and documents. Elements in a sequence are assumed to exist in the context of some document, as discussed in Section 12.1.7.

XQuery is a *functional language*, which implies that any XQuery expression can be used in any place that an expression is expected. This property is a very strong one. SQL, for example, allows subqueries in many places; but SQL does not permit, for example, any subquery to be any operand of any comparison in a where-clause. The functional property is a double-edged sword. It requires every operator of XQuery to make sense when applied to lists of more than one item, leading to some unexpected consequences.

To start, every XPath expression is an XQuery expression. There is, however, much more to XQuery, including FLWR (pronounced "flower") expressions, which are in some sense analogous to SQL select-from-where expressions.

12.2.2 FLWR Expressions

Beyond XPath expressions, the most important form of XQuery expression involves clauses of four types, called for-, let-, where-, and return- (FLWR) clauses.[1] We shall introduce each type of clause in turn. However, we should be aware that there are options in the order and occurrences of these clauses.

1. The query begins with zero or more for- and let-clauses. There can be more than one of each kind, and they can be interlaced in any order, e.g., for, for, let, for, let.
2. Then comes an optional where-clause.
3. Finally, there is exactly one return-clause.

Example 12.9: Perhaps the simplest FLWR expression is:

```
return <Greeting>Hello World</Greeting>
```

It examines no data, and produces a value that is a simple XML element. □

[1]There is also an order-by clause that we shall introduce in Section 12.2.10. For that reason, FLWR is a less common acronym for the principal form of XQuery query than is FLWOR.

Let Clauses

The simple form of a let-clause is:

`let` *variable* `:=` *expression*

The intent of this clause is that the expression is evaluated and assigned to the variable for the remainder of the FLWR expression. Variables in XQuery must begin with a dollar-sign. Notice that the assignment symbol is `:=`, not an equal sign (which is used, as in XPath, in comparisons). More generally, a comma-separated list of assignments to variables can appear where we have shown one.

Example 12.10: One use of let-clauses is to assign a variable to refer to one of the documents whose data is used by the query. For example, if we want to query a document in file `stars.xml`, we can start our query with:

```
let $stars := doc("stars.xml")
```

In what follows, the value of `$stars` is a single doc node. It can be used in front of an XPath expression, and that expression will apply to the XML document contained in the file `stars.xml`. □

For Clauses

The simple form of a for-clause is:

`for` *variable* `in` *expression*

The intent is that the expression is evaluated. The result of any expression is a sequence of items. The variable is assigned to each item, in turn, and what follows this for-clause in the query is executed once for each value of the variable. You will not be much deceived if you draw an analogy between an XQuery for-clause and a C for-statement. More generally, several variables may be set ranging over different sequences of items in one for-clause.

Example 12.11: We shall use the data suggested in Fig. 12.7 for a number of examples in this section. The data consists of two files, `stars.xml` in Fig. 12.7(a) and `movies.xml` in Fig. 12.7(b). Each of these files has data similar to what we used in Section 12.1, but the intent is that what is shown is just a small sample of the actual contents of these files.

Suppose we start a query:

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
    ... something done with each Movie element
```

```
 1)  <? xml version="1.0" encoding="utf-8" standalone="yes" ?>
 2)  <Stars>
 3)      <Star>
 4)          <Name>Carrie Fisher</Name>
 5)          <Address>
 6)              <Street>123 Maple St.</Street>
 7)              <City>Hollywood</City>
 8)          </Address>
 9)          <Address>
10)              <Street>5 Locust Ln.</Street>
11)              <City>Malibu</City>
12)          </Address>
13)      </Star>
              ... more stars
14)  </Stars>
```

(a) Document `stars.xml`

```
15)  <? xml version="1.0" encoding="utf-8" standalone="yes" ?>
16)  <Movies>
17)      <Movie title = "King Kong">
18)          <Version year = "1933">
19)              <Star>Fay Wray</Star>
20)          </Version>
21)          <Version year = "1976">
22)              <Star>Jeff Bridges</Star>
23)              <Star>Jessica Lange</Star>
24)          </Version>
25)          <Version year = "2005" />
26)      </Movie>
27)      <Movie title = "Footloose">
28)          <Version year = "1984">
29)              <Star>Kevin Bacon</Star>
30)              <Star>John Lithgow</Star>
31)              <Star>Sarah Jessica Parker</Star>
32)          </Version>
33)      </Movie>
              ... more movies
34)  </Movies>
```

(b) Document `movies.xml`

Figure 12.7: Data for XQuery examples

Boolean Values in XQuery

A comparision like `$x = 10` evaluates to true or false (strictly speaking, to one of the names `xs:true` or `xs:false` from the namespace for XML Schema). However, several other types of expressions can be interpreted as true or false, and so can serve as the value of a condition in a where-clause. The important coercions to remember are:

1. If the value is a sequence of items, then the empty sequence is interpreted as false and nonempty sequences as true.
2. Among numbers, 0 and NaN ("not a number," in essence an infinite number) are false, and other numbers are true.
3. Among strings, the empty string is false and other strings are true.

Notice that `$movies/Movies/Movie` is an XPath expression that tells us to start with the document in file `movies.xml`, then go to the root `Movies` element, and then form the sequence of all `Movie` subelements. The body of the "for-loop" will be executed first with `$m` equal to the element of lines (17) through (26) of Fig. 12.7, then with `$m` equal to the element of lines (27) through (33), and then with each of the remaining `Movie` elements in the document. □

The Where Clause

The form of a where-clause is:

`where` *condition*

This clause is applied to an item, and the *condition*, which is an expression, evaluates to true or false. If the value is true, then the return-clause is applied to the current values of any variables in the query. Otherwise, nothing is produced for the current values of variables.

The Return Clause

The form of this clause is:

`return` *expression*

The result of a FLWR expression, like that of any expression in XQuery, is a sequence of items. The sequence of items produced by the expression in the return-clause is appended to the sequence of items produced so far. Note that although there is only one return-clause, this clause may be executed many

times inside "for-loops," so the result of the query may be constructed in stages. We should not think of the return-clause as a "return-statement," since it does not end processing of the query.

Example 12.12: Let us complete the query we started in Example 12.11 by asking for a list of all the star elements found among the versions of all movies. The query is:

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
return $m/Version/Star
```

The first value of `$m` in the "for-loop" is the element of lines (17) through (26) of Fig. 12.7. From that `Movie` element, the XPath expression `/Version/Star` produces a sequence of the three `Star` elements at lines (19), (22), and (23). That sequence begins the result of the query.

```
<Star>Fay Wray</Star>
<Star>Jeff Bridges</Star>
<Star>Jessica Lange</Star>
<Star>Kevin Bacon</Star>
<Star>John Lithgow</Star>
<Star>Sarah Jessica Parker</Star>
    ...
```

Figure 12.8: Beginning of the result sequence for the query of Example 12.12

The next value of `$m` is the element of lines (27) through (33). Now, the result of the expression in the return-clause is the sequence of elements in lines (29), (30), and (31). Thus the beginning of the result sequence looks like that in Fig. 12.8. □

12.2.3 Replacement of Variables by Their Values

Let us consider a modification to the query of Example 12.12. Here, we want to produce not just a sequence of `<Star>` elements, but rather a sequence of `Movie` elements, each containing all the stars of movies with a given title, regardless of which version they starred in. The title will be an attribute of the `Movie` element.

Figure 12.9 shows an attempt that seems right, but in fact *is not correct.* The expression we return for each value of `$m` seems to be an opening `<Movie>` tag followed by the sequence of `Star` elements for that movie, and finally a closing `</Movie>` tag. The `<Movie>` tag has a `title` attribute that is a copy of the same attribute from the `Movie` element in file `movies.xml`. However, when we execute this program, what appears is:

Sequences of Sequences

We should remind the reader that sequences of items can have no internal structure. Thus, in Fig. 12.8, there is no separator between Jessica Lange and Kevin Bacon, or any grouping of the first three stars and the last three, even though these groups were produced by different executions of the return-clause.

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
return <Movie title = $m/@title>$m/Version/Star</Movie>
```

Figure 12.9: Erroneous attempt to produce `Movie` elements

```
<Movie title = "$m/@title">$m/Version/Star</Movie>
<Movie title = "$m/@title">$m/Version/Star</Movie>
    ...
```

The problem is that, between tags, or as the value of an attribute, any text string is permissible. This return statement looks no different, to the XQuery processor, than the return of Example 12.9, where we really were producing text inside matching tags. In order to get text interpreted as XQuery expressions inside tags, we need to surround the text by curly braces.

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
return <Movie title = {$m/@title}>{$m/Version/Star}</Movie>
```

Figure 12.10: Adding curly braces fixes the problem

The proper way to meet our goal is shown in Fig. 12.10. In this query, the expressions `$m/title` and `$m/Version/Star` inside the braces are properly interpreted as XPath expressions. The first is replaced by a text string, and the second is replaced by a sequence of `Star` elements, as intended.

Example 12.13: This example not only further illustrates the use of curly braces to force interpretation of expressions, but also emphasizes how any XQuery expression can be used wherever an expression of any kind is permitted. Our goal is to duplicate the result of Example 12.12, where we got a sequence of `Star` elements, but to make the entire sequence of stars be within a `Stars` element. We cannot use the trick of Fig. 12.10 with `Stars` in place of `Star`, because that would place many `Stars` tags around separate groups of stars.

```
let $starSeq := (
    let $movies := doc("movies.xml")
    for $m in $movies/Movies/Movie
    return $m/Version/Star
)
return <Stars>{$starSeq}</Stars>
```

Figure 12.11: Putting tags around a sequence

Figure 12.11 does the job. We assign the sequence of `Star` elements that results from the query of Example 12.12 to a local variable `$starSeq`. We then return that sequence, surrounded by tags, being careful to enclose the variable in braces so it is evaluated and not treated literally. □

12.2.4 Joins in XQuery

We can join two or more documents in XQuery in much the same way as we join two or more relations in SQL. In each case we need variables, each of which ranges over elements of one of the documents or tuples of one of the relations, respectively. In SQL, we use a from-clause to introduce the needed tuple variables (which may just be the table name itself); in XQuery we use a for-clause.

However, we must be very careful how we do comparisons in a join. First, there is the matter of comparison operators such as = or < operating on sequences with the meaning of "there exist elements that compare" as discussed in Section 12.1.9. We shall take up this point again in Section 12.2.5. Additionally, equality of elements is by "element identity" (analogous to "object identity"). That is, an element is not equal to a different element, even if it looks the same, character-by-character. Fortunately, we usually do not want to compare elements, but really the primitive values such as strings and integers that appear as values of their attributes and subelements. The comparison operators work as expected on primitive values; < is "precedes in lexicographic order" for strings.

There is a built-in function `data(E)` that extracts the value of an element E. We can use this function to extract the text from an element that is a string with matching tags.

Example 12.14: Suppose we want to find the cities in which stars mentioned in the `movies.xml` file of Fig. 12.7(b) live. We need to consult the `stars.xml` file of Fig. 12.7(a) to get that information. Thus, we set up a variable ranging over the `Star` elements of `movies.xml` and another variable ranging over the `Star` elements of `stars.xml`. When the data in a `Star` element of `movies.xml` matches the data in the `Name` subelement of a `Star` element of `stars.xml`, then we have a match, and we extract the `City` element of the latter.

Figure 12.12 shows a solution. The let-clause introduces variables to stand for the two documents. As before, this shorthand is not necessary, and we could have used the document nodes themselves in the XPath expressions of the next two lines. The for-clause introduces a doubly nested loop. Variable `$s1` ranges over each `Star` element of `movies.xml` and `$s2` does the same for `stars.xml`.

```
let $movies := doc("movies.xml"),
    $stars  := doc("stars.xml")
for $s1 in $movies/Movies/Movie/Version/Star,
    $s2 in $stars/Stars/Star
where data($s1) = data($s2/Name)
return $s2/Address/City
```

Figure 12.12: Finding the cities of stars

The where-clause uses the built-in function `data` to extract the strings that are the values of the elements `$s1` and `$s2`. Finally, the return-clause produces a `City` element. □

12.2.5 XQuery Comparison Operators

We shall now consider another puzzle where things don't quite work as expected. Our goal is to find the stars in `stars.xml` of Fig. 12.7(a) that live at 123 Maple St., Malibu. Our first attempt is in Fig. 12.13.

```
let $stars := doc("stars.xml")
for $s in $stars/Stars/Star
where $s/Address/Street = "123 Maple St." and
      $s/Address/City = "Malibu"
return $s/Name
```

Figure 12.13: An erroneous attempt to find who lives at 123 Maple St., Malibu

In the where-clause, we compare `Street` elements and `City` elements with strings, but that works as expected, because an element whose value is a string is coerced to that string, and the comparison will succeed when expected. The problem is seen when `$s` takes the `Star` element of lines (3) through (13) of Fig. 12.7 as its value. Then, XPath expression `$s/Address/Street` produces the sequence of two elements of lines (6) and (10) as its value. Since the `=` operator returns true if any pair of items, one from each side, equate, the value of the first condition is true; line (6), after coercion, is equal to the string `"123 Maple St."`. Similarly, the second condition compares the list of two `City` elements of lines (7) and (11) with the string `"Malibu"`, and equality is found for line (11). As a result, the `Name` element for Carrie Fisher [line (4)] is returned.

But Carrie Fisher doesn't live at 123 Maple St., Malibu. She lives at 123 Maple St., Hollywood, and elsewhere in Malibu. The existential nature of comparisons has caused us to fail to notice that we were getting a street and city from different addresses.

XQuery provides a set of comparison operators that only compare sequences consisting of a single item, and fail if either operand is a sequence of more than one item. These operators are two-letter abbreviations for the comparisons: `eq`, `ne`, `lt`, `gt`, `le`, and `ge`. We could use `eq` in place of `=` to catch the case where we are actually comparing a string with several streets or cities. The revised query is shown in Fig. 12.14.

```
let $stars := doc("stars.xml")
for $s in $stars/Stars/Star
where $s/Address/Street eq "123 Maple St." and
      $s/Address/City eq "Malibu"
return $s/Name
```

Figure 12.14: A second erroneous attempt to find who lives at 123 Maple St., Malibu

This query does not allow the Carrie-Fisher element to pass the test of the where-clause, because the left sides of the `eq` operator are not single items, and therefore the comparison fails. Unfortunately, it will not report any star with two or more addresses, even if one of those addresses is 123 Maple St., Malibu. Writing a correct query is tricky, regardless of which version of the comparison operators we use, and we leave a correct query as an exercise.

12.2.6 Elimination of Duplicates

XQuery allows us to eliminate duplicates in sequences of any kind, by applying the built-in function `distinct-values`. There is a subtlely that must be noted, however. Strictly speaking, distinct-values applies to primitive types. It will strip the tags from an element that is a tagged text-string, but it won't put them back. Thus, the input to `distinct-values` can be a list of elements and the result a list of strings.

Example 12.15: Figure 12.11 gathered all the `Star` elements from all the movies and returned them as a sequence. However, a star that appeared in several movies would appear several times in the sequence. By applying `distinct-values` to the result of the subquery that becomes the value of variable `$starseq`, we can eliminate all but one copy of each `Star` element. The new query is shown in Fig. 12.15.

Notice, however, that what is produced is a list of the names of the stars surrounded by the `Stars` tags, as:

```
<Stars>"Fay Wray" "Jeff Bridges" ··· </Stars>
```

```
let $starSeq := distinct-values(
    let $movies := doc("movies.xml")
    for $m in $movies/Movies/Movie
    return $m/Version/Star
)
return <Stars>{$starSeq}</Stars>
```

Figure 12.15: Eliminating duplicate stars

In comparison, the version in Fig. 12.11 produced

```
<Stars><Star>Fay Wray</Star> <Star>Jeff Bridges</Star> ···
                              </Stars>
```

but might produce duplicates. □

12.2.7 Quantification in XQuery

There are expressions that say, in effect, "for all" and "there exists." Their forms, respectively, are:

every *variable* in *expression1* satisfies *expression2*
some *variable* in *expression1* satisfies *expression2*

Here, *expression1* produces a sequence of items, and the variable takes on each item, in turn, as its value. For each such value, *expression2* (which normally involves the variable) is evaluated, and should produce a boolean value.

In the "every" version, the result of the entire expression is false if some item produced by *expression1* makes *expression2* false; the result is true otherwise. In the "some" version, the result of the entire expression is true if some item produced by *expression1* makes *expression2* true; the result is false otherwise.

```
let $stars := doc("stars.xml")
for $s in $stars/Stars/Star
where every $c in $s/Address/City satisfies
    $c = "Hollywood"
return $s/Name
```

Figure 12.16: Finding the stars who only live in Hollywood

Example 12.16: Using the data in the file `stars.xml` of Fig. 12.7(a), we want to find those stars who live in Hollywood and nowhere else. That is, no matter how many addresses they have, they all have city Hollywood. Figure 12.16 shows how to write this query. Notice that `$s/Address/City` produces the

sequence of `City` elements of the star `$s`. The where-clause is thus satisfied if and only if every element on that list is `<City>Hollywood</City>`.

Incidentally, we could change the "every" to "some" and find the stars that have at least one home in Hollywood. However, it is rarely necessary to use the "some" version, since most tests in XQuery are existentially quantified anyway. For instance,

```
let $stars := doc("stars.xml")
for $s in $stars/Stars/Star
where $s/Address/City = "Hollywood"
return $s/Name
```

produces the stars with a home in Hollywood, without using a "some" expression. Recall our discussion in Section 12.2.5 of how a comparision such as =, with a sequence of more than one item on either or both sides, is true if we can match any items from the two sides. □

12.2.8 Aggregations

XQuery provides built-in functions to compute the usual aggregations such as count, sum, or max. They take any sequence as argument; that is, they can be applied to the result of any XQuery expression.

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
where count($m/Version) > 1
return $m
```

Figure 12.17: Finding the movies with multiple versions

Example 12.17: Let us examine the data in file `movies.xml` of Fig. 12.7(b) and produce those `Movie` elements that have more than one version. Figure 12.17 does the job. The XPath expression `$m/Version` produces the sequence of `Version` elements for the movie `$m`. The number of items in the sequence is counted. If that count exceeds 1, the where-clause is satisfied, and the movie element `$m` is appended to the result. □

12.2.9 Branching in XQuery Expressions

There is an if-then-else expression in XQuery of the form

`if (`*expression1*`) then` *expression2* `else` *expression3*

To evaluate this expression, first evaluate *expression1*; if it is true, evaluate *expression2*, which becomes the result of the whole expression. If *expression1* is false, the result of the whole expression is *expression3*.

This expression is not a statement — there are no statements in XQuery, only expressions. Thus, the analog in C is the ?: expression, not the if-then-else statement. Like the expression in C, there is no way to omit the "else" part. However, we can use as *expression3* the empty sequence, which is denoted `()`. This choice makes the conditional expression produce the empty sequence when the test-condition is not satisfied.

Example 12.18: Our goal in this example is to produce each of the versions of *King Kong*, tagging the most recent version `Latest` and the earlier versions `Old`. In line (1), we set variable `$kk` to be the `Movie` element for `King Kong`. Notice that we have used an XPath condition in this line, to make sure that we produce only that one element. Of course, if there were several `Movie` elements that had the title `King Kong`, then all of them would be on the sequence of items that is the value of `$kk`, and the query would make no sense. However, we are assuming title is a key for movies in this structure, since we have explicitly grouped versions of movies with the same title.

```
1)  let $kk :=
            doc("movies.xml")/Movies/Movie[@title = "King Kong"]
2)  for $v in $kk/Version
3)  return
4)      if ($v/@year = max($kk/Version/@year))
5)      then <Latest>{$v}</Latest>
6)      else <Old>{$v}</Old>
```

Figure 12.18: Tagging the versions of *King Kong*

Line (2) causes `$v` to iterate over all versions of *King Kong*. For each such version, we return one of two elements. To tell which, we evaluate the condition of line (4). On the right of the equal-sign is the maximum year of any of the *King-Kong* versions, and on the left is the year of the version `$v`. If they are equal, then `$v` is the latest version, and we produce the element of line (5). If not, then `$v` is an old version, and we produce the element of line (6). □

12.2.10 Ordering the Result of a Query

It is possible to sort the results as part of a FLWR query, if we add an order-clause before the return-clause. In fact, the query form we have been concentrating on here is usually called FLWOR (but still pronounced "flower"), to acknowledge the optional presence of an order-clause. The form of this clause is:

`order` *list of expressions*

The sort is based on the value of the first expression, ties are broken by the value of the second expression, and so on. The default order is ascending, but the keyword `descending` following an expression reverses the order.

What happens when an order is present is analogous to what happens in SQL. Just before we reach the stage in query processing where the output is assembled (the SELECT clause in SQL; the return-clause in XQuery), the result of previous clauses is assembled and sorted. In the case of SQL, the intermediate result is a set of bindings of tuples to the tuple variables that range over each of the relations in the FROM clause. Specifically, it is all those bindings that pass the test of the WHERE clause.

In XQuery, we should think of the intermediate result as a sequence of bindings of variables to values. The variables are those defined in the for- and let-clauses that precede the order-clause, and the sequence consists of all those bindings that pass the test of the where-clause. These bindings are each used to evaluate the expressions in the order-clause, and the values of those expressions govern the position of the binding in the order of all the bindings. Once we have the order of bindings, we use them, in turn, to evaluate the expression in the return-clause.

Example 12.19: Let us consider all versions of all movies, order them by year, and produce a sequence of `Movie` elements with the title and year as attributes. The data comes from file `movies.xml` in Fig. 12.7(b), as usual. The query is shown in Fig. 12.19.

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie,
    $v in $m/Version
order $v/@year
return <Movie title = "{$m/@title}" year = "{$v/@year}" />
```

Figure 12.19: Construct the sequence of title-year pairs, ordered by year

When we reach the order-clause, bindings provide values for the three variables `$movies`, `$m`, and `$v`. The value `doc("movies.xml)` is bound to `$movies` in every one of these bindings. However, the values of `$m` and `$v` vary; for each pair consisting of a movie and a version of that movie, there will be one binding for the two variables. For instance, the first such binding associates with `$m` the element in lines (17) through (26) of Fig. 12.7(b) and associates with `$v` the element of lines (18) through (20).

The bindings are sorted according to the value of attribute `year` in the element to which `$v` is bound. There may be many movies with the same year, and the ordering does not specify how these are to be ordered. As a result, all we know is that the movie-version pairs with a given year will appear together in some order, and the groups for each year will be in the ascending order of year. If we wanted to specify a total ordering of the bindings, we could, for example, add a second term to the list in the order-clause, such as:

```
order $v/@year, $m/@title
```

to break ties alphabetically by title.

After sorting the bindings, each binding is passed to the return-clause, in the order chosen. By substituting for the variables in the return-clause, we produce from each binding a single `Movie` element. □

12.2.11 Exercises for Section 12.2

Exercise 12.2.1: Using the product data from Figs. 12.4 and 12.5, write the following in XQuery.

a) Find the `Printer` elements with a price less than 100.

b) Find the `Printer` elements with a price less than 100, and produce the sequence of these elements surrounded by a tag `<CheapPrinters>`.

! c) Find the names of the makers of both printers and laptops.

! d) Find the names of the makers that produce at least two PC's with a speed of 3.00 or more.

! e) Find the makers such that every PC they produce has a price no more than 1000.

!! f) Produce a sequence of elements of the form

`<Laptop><Model>`x`</Model><Maker>`y`</Maker></Laptop>`

where x is the model number and y is the name of the maker of the laptop.

Exercise 12.2.2: Using the battleships data of Fig. 12.6, write the following in XQuery.

a) Find the names of the classes that had at least 10 guns.

b) Find the names of the ships that had at least 10 guns.

c) Find the names of the ships that were sunk.

d) Find the names of the classes with at least 3 ships.

! e) Find the names of the classes such that no ship of that class was in a battle.

!! f) Find the names of the classes that had at least two ships launched in the same year.

!! g) Produce a sequence of items of the form

`<Battle name =` x`><Ship name =` y `/>`$\cdots$`</Battle>`

where x is the name of a battle and y the name of a ship in the battle. There may be more than one `Ship` element in the sequence.

! **Exercise 12.2.3:** Solve the problem of Section 12.2.5; write a query that finds the star(s) living at a given address, even if they have several addresses, without finding stars that do not live at that address.

! **Exercise 12.2.4:** Do there exist expressions E and F such that the expression `every $x in` E `satisfies` F is true, but `some $x in` E `satisfies` F is false? Either give an example or explain why it is impossible.

12.3 Extensible Stylesheet Language

XSLT (Extensible Stylesheet Language for Transformations) is a standard of the World-Wide-Web Consortium. Its original purpose was to allow XML documents to be transformed into HTML or similar forms that allowed the document to be viewed or printed. However, in practice, XSLT is another query language for XML. Like XPath or XQuery, we can use XSLT to extract data from documents or turn one document form into another form.

12.3.1 XSLT Basics

Like XML Schema, XSLT specifications are XML documents; these specifications are usually called *stylesheets*. The tags used in XSLT are found in a namespace, which is `http://www.w3.org/1999/XSL/Transform`. Thus, at the highest level, a stylesheet looks like Fig. 12.20.

```
<? xml version = "1.0" encoding = "utf-8" ?>
<xsl:stylesheet xmlns:xsl =
        "http://www.w3.org/1999/XSL/Transform">
    ...
</xsl:stylesheet>
```

Figure 12.20: The form of an XSLT stylesheet

12.3.2 Templates

A stylesheet will have one or more *templates*. To apply a stylesheet to an XML document, we go down the list of templates until we find one that matches the root. As processing proceeds, we often need to find matching templates for elements nested within the document. If so, we again search the list of templates for a match according to matching rules that we shall learn in this section. The simplest form of a template tag is:

<xsl:template match = "*XPath expression*">

The XPath expression, which can be either rooted (beginning with a slash) or relative, describes the elements of an XML document to which this template is applied. If the expression is rooted, then the template is applied to every element of the document that matches the path. Relative expressions are applied when a template T has within it a tag <xsl:apply-templates>. In that case, we look among the children of the elements to which T is applied. In that way, we can traverse an XML document's tree in a depth-first manner, performing complicated transformations on the document.

The simplest content of a template is text, typically HTML. When a template matches a document, the text inside that document is produced as output. Within the text can be calls to apply templates to the children and/or obtain values from the document itself, e.g., from attributes of the current element.

```
1)    <? xml version = "1.0" encoding = "utf-8" ?>
2)    <xsl:stylesheet xmlns:xsl =
3)            "http://www.w3.org/1999/XSL/Transform">
4)        <xsl:template match = "/">
5)            <HTML>
6)                <BODY>
7)                    <B>This is a document</b>
8)                </body>
9)            </html>
10)       </xsl:template>
11)   </xsl:stylesheet>
```

Figure 12.21: Printing output for any document

Example 12.20: In Fig. 12.21 is an exceedingly simple stylesheet. It applies to any document and produces the same HTML document, regardless of its input. This HTML document says "This is a document" in boldface.

Line (4) introduces the one template in the stylesheet. The value of the match attribute is "/", which matches only the root. The body of the template, lines (5) through (9), is simple HTML. When these lines are produced as output, the resulting file can be treated as HTML and displayed by a browser or other HTML processor. □

12.3.3 Obtaining Values From XML Data

It is unusual that the document we produce does not depend in any way on the input to the transformation, as was the case in Example 12.20. The simplest way to extract data from the input is with the value-of tag. The form of this tag is:

```
<? xml version="1.0" encoding="utf-8" standalone="yes" ?>
<Movies>
    <Movie title = "King Kong">
        <Version year = "1933">
            <Star>Fay Wray</Star>
        </Version>
        <Version year = "1976">
            <Star>Jeff Bridges</Star>
            <Star>Jessica Lange</Star>
        </Version>
        <Version year = "2005" />
    </Movie>
    <Movie title = "Footloose">
        <Version year = "1984">
            <Star>Kevin Bacon</Star>
            <Star>John Lithgow</Star>
            <Star>Sarah Jessica Parker</Star>
        </Version>
    </Movie>
          ... more movies
</Movies>
```

Figure 12.22: The file `movies.xml`

```
<xsl:value-of select = "expression" />
```

The expression is an XPath expression that should produce a string as value. Other values, such as elements containing text, are coerced into strings in the obvious way.

Example 12.21: In Fig. 12.22 we reproduce the file `movies.xml` that was used in Section 12.2 as a running example. In this example of a stylesheet, we shall use `value-of` to obtain all the titles of movies and print them, one to a line. The stylesheet is shown in Fig. 12.23.

At line (4), we see that the template matches every `Movie` element, so we process them one at a time. Line (5) applies the `value-of` operation with an XPath expression `@title`. That is, we go to the `title` attribute of each `Movie` element and take the value of that attribute. This value is produced as output, and followed at line (6) by the HTML break tag, so the next movie title will be printed on the next line. □

12.3.4 Recursive Use of Templates

The most interesting and powerful transformations require recursive application of templates at various elements of the input. Having selected a template to

```
1)    <? xml version = "1.0" encoding = "utf-8" ?>
2)    <xsl:stylesheet xmlns:xsl =
3)            "http://www.w3.org/1999/XSL/Transform">
4)        <xsl:template match = "/Movies/Movie">
5)            <xsl:value-of select = "@title" />
6)            <BR/>
7)        </xsl:template>
8)    </xsl:stylesheet>
```

Figure 12.23: Printing the titles of movies

apply to the root of the input document, we can ask that a template be applied to each of its subelements, by using the `apply-templates` tag. If we want to apply a certain template to only some subset of the subelements, e.g., those with a certain tag, we can use a `select` expression, as:

`<xsl:apply-templates select = "`*expression*`" />`

When we encounter such a tag within a template, we find the set of matching subelements of the current element (the element to which the template is being applied). For each subelement, we find the first template that matches and apply it to the subelement.

Example 12.22: In this example, we shall use XSLT to transform an XML document into another XML document, rather than into an HTML document. Let us examine Fig. 12.24. There are four templates, and together they process movie data in the form of Fig. 12.22. The first template, lines (4) through (8), matches the root. It says to output the text `<Movies>` and then apply templates to the children of the root element. We could have specified that templates were to be applied only to children that are tagged `<Movie>`, but since we expect no other tags among the children, we did not specify:

```
6)        <xsl:apply-templates select = "Movie" />
```

Notice that after applying templates to the `<Movie>` children (which will result in the printing of many elements), we close the `<Movies>` element in the output with the appropriate closing tag at line (7). Also observe that we can tell the difference between tags that are output text, such as lines (5) and (7), from tags that are XSLT, because all XSLT tags must be from the `xsl` namespace.

Now, let us see what applying templates to the `<Movie>` elements does. The first (and only) template that matches these elements is the second, at lines (9) through (15). This template begins by outputting the text `<Movie title = "` at line (10). Then, line (11) obtains the title of the movie and emits it to the output. Line (12) finishes the quoted attribute value and the `<Movie>` tag in the output. Line (13) applies templates to all the children of the movie, which should be versions. Finally, line (14) emits the matching `</Movie>` ending tag.

```
 1)   <? xml version = "1.0" encoding = "utf-8" ?>
 2)   <xsl:stylesheet xmlns:xsl =
 3)           "http://www.w3.org/1999/XSL/Transform">

 4)       <xsl:template match = "/Movies">
 5)           <Movies>
 6)           <xsl:apply-templates />
 7)           </Movies>
 8)       </xsl:template>

 9)       <xsl:template match = "Movie">
10)           <Movie title = "
11)           <xsl:value-of select = "@title" />
12)           ">
13)           <xsl:apply-templates />
14)           </Movie>
15)       </xsl:template>

16)       <xsl:template match = "Version">
17)           <xsl:apply-templates />
18)       </xsl:template>

19)       <xsl:template match = "Star">
20)           <Star name = "
21)           <xsl:value-of select = "." />
22)           " />
23)       </xsl:template>

24)   </xsl:stylesheet>
```

Figure 12.24: Transforming the `movies.xml` file

When line (13) calls for templates to be applied to all the versions of a movie, the only matching template is that of lines (16) through (18), which does nothing but apply templates to the children of the version, which should be `<Star>` elements. Thus, what gets generated between each opening `<Movie>` tag and its matched closing tag is determined by the last template of lines (19) through (23). This template is applied to each `<Star>` element.

Star elements from the input are transformed in the output. Instead of the star's name being text, as it is in Fig. 12.22, the template starting at line (19) produces a `<Star>` element with the name as an attribute. Line (21) says to select the `<Star>` element itself (the dot represents the "self" axis as an XPath expression) as a value for the output. However, all output is text, so the tags of the element are not part of the output. That result is exactly what we want,

since the value of the attribute name should be a string, not an element. The empty <Star> element is completed on line (22). For instance, given the input of Fig. 12.22, the output would be as shown in Fig. 12.25. □

```
<Movies>
    <Movie title = "King Kong">
        <Star name = "Fay Wray" />
        <Star name = "Jeff Bridges" />
        <Star name = "Jessica Lange" />
    </Movie>
    <Movie title = "Footloose">
        <Star name = "Kevin Bacon" />
        <Star name = "John Lithgow" />
        <Star name = "Sarah Jessica Parker" />
    </Movie>
          ... more movies
</Movies>
```

Figure 12.25: Output of the transform of Fig. 12.24

12.3.5 Iteration in XSLT

We can put a loop within a template that gives us freedom over the order in which we visit certain subelements of the element to which the template is being applied. The for-each tag creates the loop, with a form:

<xsl:for-each select = "*expression*">

The expression is an XPath expression whose value is a sequence of items. Whatever is between the opening <for-each> tag and its matched closing tag is executed for each item, in turn.

Example 12.23: In Fig. 12.26 is a copy of our document stars.xml; we wish to transform it to an HTML list of all the names of stars followed by an HTML list of all the cities in which stars live. Figure 12.27 has a template that does the job.

There is one template, which matches the root. The first thing that happens is at line (5), where the HTML tag <OL> is emitted to start an ordered list. Then, line (6) starts a loop, which iterates over each <Star> subelement. At lines (7) through (9), a list item with the name of that star is emitted. Line (11) ends the list of names and begins a list of cities. The second loop, lines (12) through (16), runs through each <Address> element and emits a list item for the city. Line (17) closes the second list. □

```
<? xml version="1.0" encoding="utf-8" standalone="yes" ?>
<Stars>
    <Star>
        <Name>Carrie Fisher</Name>
        <Address>
            <Street>123 Maple St.</Street>
            <City>Hollywood</City>
        </Address>
        <Address>
            <Street>5 Locust Ln.</Street>
            <City>Malibu</City>
        </Address>
    </Star>
          ... more stars
</Stars>
```

Figure 12.26: Document `stars.xml`

```
 1)    <? xml version = "1.0" encoding = "utf-8" ?>
 2)    <xsl:stylesheet xmlns:xsl =
 3)            "http://www.w3.org/1999/XSL/Transform">
 4)        <xsl:template match = "/">
 5)            <OL>
 6)            <xsl:for-each select = "Stars/Star" />
 7)                <LI>
 8)                <xsl:value-of select = "Name">
 9)                </li>
10)            </xsl:for-each>
11)            </ol><P/><OL>
12)            <xsl:for-each select = "Stars/Star/Address" />
13)                <LI>
14)                <xsl:value-of select = "City">
15)                </li>
16)            </xsl:for-each>
17)            </ol>
18)        </xsl:template>
19)    </xsl:stylesheet>
```

Figure 12.27: Printing names and cities of stars

12.3.6 Conditionals in XSLT

We can introduce branching into our templates by using an `if` tag. The form of this tag is:

```
<xsl:if test = "boolean expression">
```

Whatever appears between this tag and its matched closing tag is executed if and only if the boolean expression is true. There is no else-clause, but we can follow this expression by another `if` that has the opposite test condition should we wish.

```
 1)   <? xml version = "1.0" encoding = "utf-8" ?>
 2)   <xsl:stylesheet xmlns:xsl =
 3)           "http://www.w3.org/1999/XSL/Transform">
 4)       <xsl:template match = "/">
 5)           <TABLE border = "5"><TR><TH>Stars</th></tr>
 6)           <xsl:for-each select = "Stars/Star" />
 7)               <xsl:if test = "Address/City = 'Hollywood'">
 8)                   <TR><TD>
 9)                   <xsl:value-of select = "Name" />
10)                   </td></tr>
11)               </xsl:if>
12)           </xsl:for-each>
13)           </table>
14)       </xsl:template>
15)   </xsl:stylesheet>
```

Figure 12.28: Finding the names of the stars who live in Hollywood

Example 12.24: Figure 12.28 is a stylesheet that prints a one-column table, with header "Stars." There is one template, which matches the root. The first thing this template does is print the header row at line (5). The for-each loop of lines (6) through (12) iterates over each star. The conditional of line (7) tests whether the star has at least one home in Hollywood. Remember that the equal-sign represents a comparison is true if any item on the left equals any item on the right. That is what we want, since we asked whether any of the homes a star has is in Hollywood. Lines (8) through (10) print a row of the table. □

12.3.7 Exercises for Section 12.3

Exercise 12.3.1: Suppose our input XML document has the form of the product data of Figs. 12.4 and 12.5. Write XSLT stylesheets to produce each of the following documents.

a) An HTML file consisting of a header "Manufacturers" followed by an enumerated list of the names of all the makers of products listed in the input.

b) An HTML file consisting of a table with headers "Model" and "Price," with a row for each PC. That row should have the proper model and price for the PC.

! c) An HTML file consisting of a table whose headers are "Model," "Price," "Speed," and "Ram" for all Laptops, followed by another table with the same headers for PC's.

d) An XML file with root tag `<PCs>` and subelements having tag `<PC>`. This tag has attributes `model`, `price`, `speed`, and `ram`. In the output, there should be one `<PC>` element for each `<PC>` element of the input file, and the values of the attributes should be taken from the corresponding input element.

!! e) An XML file with root tag `<Products>` whose subelements are `<Product>` elements. Each `<Product>` element has attributes `type`, `maker`, `model`, and `price`, where the type is one of `"PC"`, `"Laptop"`, or `"Printer"`. There should be one `<Product>` element in the output for every PC, laptop, and printer in the input file, and the output values should be chosen appropriately from the input data.

! f) Repeat part (b), but make the output file a Latex file.

Exercise 12.3.2: Suppose our input XML document has the form of the product data of Fig. 12.6. Write XSLT stylesheets to produce each of the following documents.

a) An HTML file with a header for each class. Under each header is a table with column-headers "Name" and "Launched" with the appropriate entry for each ship of the class.

b) An HTML file with root tag `<Losers>` and subelements `<Ship>`, each of whose values is the name of one of the ships that were sunk.

! c) An XML file with root tag `<Ships>` and subelements `<Ship>` for each ship. These elements each should have attributes `name`, `class`, `country` and `numGuns` with the appropriate values taken from the input file.

! d) Repeat (c), but only list those ships that were in at least one battle.

e) An XML file identical to the input, except that `<Battle>` elements should be empty, with the outcome and name of the battle as two attributes.

12.4 Summary of Chapter 12

- *XPath*: This language is a simple way to express many queries about XML data. You describe paths from the root of the document by sequences of tags. The path may end at an attribute rather than an element.
- *The XPath Data Model*: All XPath values are sequences of items. An item is either a primitive value or an element. An element is an opening XML tag, its matched closing tag, and everything in between.
- *Axes*: Instead of proceeding down the tree in a path, one can follow another axis, including jumps to any descendant, a parent, or a sibling.
- *XPath Conditions*: Any step in a path can be constrained by a condition, which is a boolean-valued expression. This expression appears in square brackets.
- *XQuery*: This language is a more advanced form of query language for XML documents. It uses the same data model as XPath. XQuery is a functional language.
- *FLWR Expressions*: Many queries in XQuery consist of let-, for-, where- and return-clauses. "Let" introduces temporary definitions of variables; "for" creates loops; "where" supplies conditions to be tested, and "return" defines the result of the query.
- *Comparison Operators in XQuery and XPath*: The conventional comparison operators such as $<$ apply to sequences of items, and have a "there-exists" meaning. They are true if the stated relation holds between any pair of items, one from each of the lists. To be assured that single items are being compared, we can use letter codes for the operators, such as `lt` for "less than."
- *Other XQuery Expressions*: XQuery has many operations that resemble those in SQL. These operators include existential and universal quantification, aggregation, duplicate-elimination, and sorting of results.
- *XSLT*: This language is designed for transformations of XML documents, although it also can be used as a query language. A "program" in this language has the form of an XML document, with a special namespace that allows us to use tags to describe a transformation.
- *Templates*: The heart of XSLT is a template, which matches certain elements of the input document. The template describes output text, and can extract values from the input document for inclusion in the output. A template can also call for templates to be applied recursively to the children of an element.

- *XSLT Programming Constructs*: A template can also include XSLT constructs that behave like an iterative programming language. These constructs include for-loops and if-statements.

12.5 References for Chapter 12

The World-Wide-Web Consortium site for the definition of XPath is [2]. The site for XQuery is [3], and for XSLT it is [4].

[1] is an introduction to the XQuery language. There are tutorials for XPath, XQuery, and XSLT at [5].

1. D. D. Chamberlin, "XQuery: an XML Query Language," *IBM Systems Journal* **41**:4 (2002), pp. 597–615. See also

   `www.research.ibm.com/journal/sj/414/chamberlin.pdf`

2. World-Wide-Web Consortium `http://www.w3.org/TR/xpath`

3. World-Wide-Web Consortium `http://www.w3.org/TR/xquery`

4. World-Wide-Web Consortium `http://www.w3.org/TR/xslt`

5. W3 Schools, `http://www.w3schools.com`

Part IV

Database System Implementation

Chapter 13

Secondary Storage Management

Database systems always involve secondary storage — the disks and other devices that store large amounts of data that persists over time. This chapter summarizes what we need to know about how a typical computer system manages storage. We review the memory hierarchy of devices with progressively slower access but larger capacity. We examine disks in particular and see how the speed of data access is affected by how we organize our data on the disk. We also study mechanisms for making disks more reliable.

Then, we turn to how data is represented. We discuss the way tuples of a relation or similar records or objects are stored. Efficiency, as always, is the key issue. We cover ways to find records quickly, and how to manage insertions and deletions of records, as well as records whose sizes grow and shrink.

13.1 The Memory Hierarchy

We begin this section by examining the memory hierarchy of a computer system. We then focus on disks, by far the most common device at the "secondary-storage" level of the hierarchy. We give the rough parameters that determine the speed of access and look at the transfer of data from disks to the lower levels of the memory hierarchy.

13.1.1 The Memory Hierarchy

A typical computer system has several different components in which data may be stored. These components have data capacities ranging over at least seven orders of magnitude and also have access speeds ranging over seven or more orders of magnitude. The cost per byte of these components also varies, but more slowly, with perhaps three orders of magnitude between the cheapest and

most expensive forms of storage. Not surprisingly, the devices with smallest capacity also offer the fastest access speed and have the highest cost per byte. A schematic of the memory hierarchy is shown in Fig. 13.1.

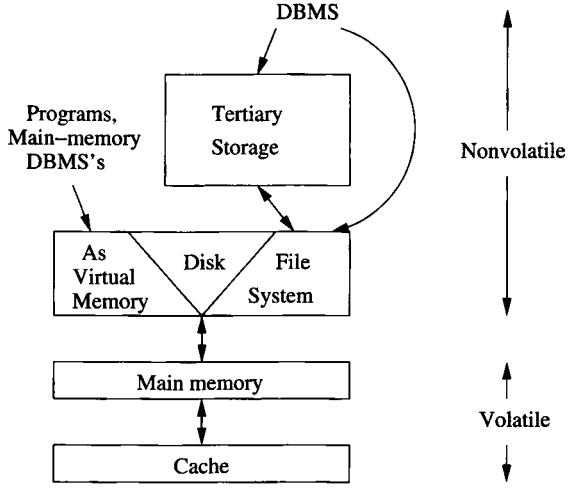

Figure 13.1: The memory hierarchy

Here are brief descriptions of the levels, from the lowest, or fastest-smallest level, up.

1. *Cache.* A typical machine has a megabyte or more of cache storage. *On-board cache* is found on the same chip as the microprocessor itself, and additional *level-2 cache* is found on another chip. Data and instructions are moved to cache from main memory when they are needed by the processor. Cached data can be accessed by the processor in a few nanoseconds.

2. *Main Memory.* In the center of the action is the computer's *main memory.* We may think of everything that happens in the computer — instruction executions and data manipulations — as working on information that is resident in main memory (although in practice, it is normal for what is used to migrate to the cache). A typical machine in 2008 is configured with about a gigabyte of main memory, although much larger main memories are possible. Typical times to move data from main memory to the processor or cache are in the 10–100 nanosecond range.

3. *Secondary Storage.* Secondary storage is typically magnetic disk, a device we shall consider in detail in Section 13.2. In 2008, single disk units have capacities of up to a terabyte, and one machine can have several disk units. The time to transfer a single byte between disk and main

Computer Quantities are Powers of 2

It is conventional to talk of sizes or capacities of computer components as if they were powers of 10: megabytes, gigabytes, and so on. In reality, since it is most efficient to design components such as memory chips to hold a number of bits that is a power of 2, all these numbers are really shorthands for nearby powers of 2. Since $2^{10} = 1024$ is very close to a thousand, we often maintain the fiction that $2^{10} = 1000$, and talk about 2^{10} with the prefix "kilo," 2^{20} as "mega," 2^{30} as "giga," 2^{40} as "tera," and 2^{50} as "peta," even though these prefixes in scientific parlance refer to 10^3, 10^6, 10^9, 10^{12} and 10^{15}, respectively. The discrepancy grows as we talk of larger numbers. A "gigabyte" is really 1.074×10^9 bytes.

We use the standard abbreviations for these numbers: K, M, G, T, and P for kilo, mega, giga, tera, and peta, respectively. Thus, 16Gb is sixteen gigabytes, or strictly speaking 2^{34} bytes. Since we sometimes want to talk about numbers that are the conventional powers of 10, we shall reserve for these the traditional numbers, without the prefixes "kilo," "mega," and so on. For example, "one million bytes" is 1,000,000 bytes, while "one megabyte" is 1,048,576 bytes.

A recent trend is to use "kilobyte," "megabyte," and so on for exact powers of ten, and to replace the third and fourth letters by "bi" to represent the similar powers of two. Thus, "kibibyte" is 1024 bytes, "mebibyte" is 1,048,576 bytes, and so on. We shall not use this convention.

memory is around 10 miliseconds. However, large numbers of bytes can be transferred at one time, so the matter of how fast data moves from and to disk is somewhat complex.

4. *Tertiary Storage.* As capacious as a collection of disk units can be, there are databases much larger than what can be stored on the disk(s) of a single machine, or even several machines. To serve such needs, *tertiary storage* devices have been developed to hold data volumes measured in terabytes. Tertiary storage is characterized by significantly higher read/write times than secondary storage, but also by much larger capacities and smaller cost per byte than is available from magnetic disks. Many tertiary devices involve robotic arms or conveyors that bring storage media such as magnetic tape or optical disks (e.g., DVD's) to a reading device. Retrieval takes seconds or minutes, but capacities in the petabyte range are possible.

13.1.2 Transfer of Data Between Levels

Normally, data moves between adjacent levels of the hierarchy. At the secondary and tertiary levels, accessing the desired data or finding the desired place to store data takes a great deal of time, so each level is organized to transfer large amounts of data to or from the level below, whenever any data at all is needed. Especially important for understanding the operation of a database system is the fact that the disk is organized into *disk blocks* (or just *blocks*, or as in operating systems, *pages*) of perhaps 4–64 kilobytes. Entire blocks are moved to or from a continuous section of main memory called a *buffer*. Thus, a key technique for speeding up database operations is to arrange data so that when one piece of a disk block is needed, it is likely that other data on the same block will also be needed at about the same time.

The same idea applies to other hierarchy levels. If we use tertiary storage, we try to arrange so that when we select a unit such as a DVD to read, we need much of what is on that DVD. At a lower level, movement between main memory and cache is by units of *cache lines*, typically 32 consecutive bytes. The hope is that entire cache lines will be used together. For example, if a cache line stores consecutive instructions of a program, we hope that when the first instruction is needed, the next few instructions will also be executed immediately thereafter.

13.1.3 Volatile and Nonvolatile Storage

An additional distinction among storage devices is whether they are *volatile* or *nonvolatile*. A volatile device "forgets" what is stored in it when the power goes off. A nonvolatile device, on the other hand, is expected to keep its contents intact even for long periods when the device is turned off or there is a power failure. The question of volatility is important, because one of the characteristic capabilities of a DBMS is the ability to retain its data even in the presence of errors such as power failures.

Magnetic and optical materials hold their data in the absence of power. Thus, essentially all secondary and tertiary storage devices are nonvolatile. On the other hand, main memory is generally volatile (although certain types of more expensive memory chips, such as flash memory, can hold their data after a power failure). A significant part of the complexity in a DBMS comes from the requirement that no change to the database can be considered final until it has migrated to nonvolatile, secondary storage.

13.1.4 Virtual Memory

Typical software executes in *virtual-memory*, an address space that is typically 32 bits; i.e., there are 2^{32} bytes, or 4 gigabytes, in a virtual memory. The operating system manages virtual memory, keeping some of it in main memory and the rest on disk. Transfer between memory and disk is in units of disk

Moore's Law

Gordon Moore observed many years ago that integrated circuits were improving in many ways, following an exponential curve that doubles about every 18 months. Some of these parameters that follow "Moore's law" are:

1. The number of instructions per second that can be executed for unit cost. Until about 2005, the improvement was achieved by making processor chips faster, while keeping the cost fixed. After that year, the improvement has been maintained by putting progressively more processors on a single, fixed-cost chip.
2. The number of memory bits that can be bought for unit cost and the number of bits that can be put on one chip.
3. The number of bytes per unit cost on a disk and the capacity of the largest disks.

On the other hand, there are some other important parameters that do not follow Moore's law; they grow slowly if at all. Among these slowly growing parameters are the speed of accessing data in main memory and the speed at which disks rotate. Because they grow slowly, "latency" becomes progressively larger. That is, the time to move data between levels of the memory hierarchy appears enormous today, and will only get worse.

blocks (pages). Virtual memory is an artifact of the operating system and its use of the machine's hardware, and it is not a level of the memory hierarchy.

The path in Fig. 13.1 involving virtual memory represents the treatment of conventional programs and applications. It does *not* represent the typical way data in a database is managed, since a DBMS manages the data itself. However, there is increasing interest in *main-memory database systems*, which do indeed manage their data through virtual memory, relying on the operating system to bring needed data into main memory through the paging mechanism. Main-memory database systems, like most applications, are most useful when the data is small enough to remain in main memory without being swapped out by the operating system.

13.1.5 Exercises for Section 13.1

Exercise 13.1.1: Suppose that in 2008 the typical computer has a processor chip with two processors ("cores") that each run at 3 gigahertz, has a disk of 250 gigabytes, and a main memory of 1 gigabyte. Assume that Moore's law (these factors double every 18 months) holds into the indefinite future.

a) When will petabyte disks be common?

b) When will terabyte main memories be common?

c) When will terahertz processor chips be common (i.e., the total number of cycles per second of all the cores on a chip will be approximately 10^{12}?

d) What will be a typical configuration (processor, disk, memory) in the year 2015?

! **Exercise 13.1.2:** Commander Data, the android from the 24th century on *Star Trek: The Next Generation* once proudly announced that his processor runs at "12 teraops." While an operation and a cycle may not be the same, let us suppose they are, and that Moore's law continues to hold for the next 300 years. If so, what would Data's true processor speed be?

13.2 Disks

The use of secondary storage is one of the important characteristics of a DBMS, and secondary storage is almost exclusively based on magnetic disks. Thus, to motivate many of the ideas used in DBMS implementation, we must examine the operation of disks in detail.

13.2.1 Mechanics of Disks

The two principal moving pieces of a disk drive are shown in Fig. 13.2; they are a *disk assembly* and a *head assembly*. The disk assembly consists of one or more circular *platters* that rotate around a central spindle. The upper and lower surfaces of the platters are covered with a thin layer of magnetic material, on which bits are stored. 0's and 1's are represented by different patterns in the magnetic material. A common diameter for disk platters is 3.5 inches, although disks with diameters from an inch to several feet have been built.

The disk is organized into *tracks*, which are concentric circles on a single platter. The tracks that are at a fixed radius from the center, among all the surfaces, form one *cylinder*. Tracks occupy most of a surface, except for the region closest to the spindle, as can be seen in the top view of Fig. 13.3. The density of data is much greater along a track than radially. In 2008, a typical disk has about 100,000 tracks per inch but stores about a million bits per inch along the tracks.

Tracks are organized into *sectors*, which are segments of the circle separated by *gaps* that are not magnetized to represent either 0's or 1's.[1] The sector is an indivisible unit, as far as reading and writing the disk is concerned. It is also indivisible as far as errors are concerned. Should a portion of the magnetic layer

[1] We show each track with the same number of sectors in Fig. 13.3. However, the number of sectors per track normally varies, with the outer tracks having more sectors than inner tracks.

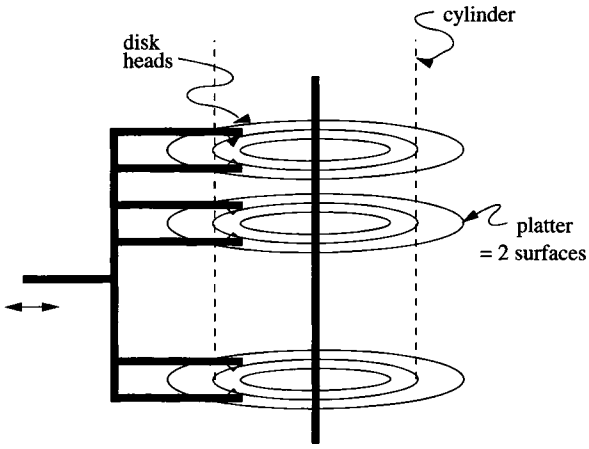

Figure 13.2: A typical disk

be corrupted in some way, so that it cannot store information, then the entire sector containing this portion cannot be used. Gaps often represent about 10% of the total track and are used to help identify the beginnings of sectors. As we mentioned in Section 13.1.2, blocks are logical units of data that are transferred between disk and main memory; blocks consist of one or more sectors.

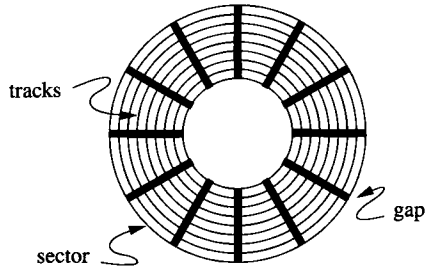

Figure 13.3: Top view of a disk surface

The second movable piece shown in Fig. 13.2, the head assembly, holds the *disk heads*. For each surface there is one head, riding extremely close to the surface but never touching it (or else a "head crash" occurs and the disk is destroyed). A head reads the magnetism passing under it, and can also alter the magnetism to write information on the disk. The heads are each attached to an arm, and the arms for all the surfaces move in and out together, being part of the rigid head assembly.

Example 13.1: The *Megatron 747* disk has the following characteristics, which

are typical of a large vintage-2008 disk drive.

- There are eight platters providing sixteen surfaces.
- There are 2^{16}, or 65,536, tracks per surface.
- There are (on average) $2^8 = 256$ sectors per track.
- There are $2^{12} = 4096$ bytes per sector.

The capacity of the disk is the product of 16 surfaces, times 65,536 tracks, times 256 sectors, times 4096 bytes, or 2^{40} bytes. The Megatron 747 is thus a terabyte disk. A single track holds 256×4096 bytes, or 1 megabyte. If blocks are 2^{14}, or 16,384 bytes, then one block uses 4 consecutive sectors, and there are (on average) $256/4 = 32$ blocks on a track. □

13.2.2 The Disk Controller

One or more disk drives are controlled by a *disk controller*, which is a small processor capable of:

1. Controlling the mechanical actuator that moves the head assembly, to position the heads at a particular radius, i.e., so that any track of one particular cylinder can be read or written.
2. Selecting a sector from among all those in the cylinder at which the heads are positioned. The controller is also responsible for knowing when the rotating spindle has reached the point where the desired sector is beginning to move under the head.
3. Transferring bits between the desired sector and the computer's main memory.
4. Possibly, buffering an entire track or more in local memory of the disk controller, hoping that many sectors of this track will be read soon, and additional accesses to the disk can be avoided.

Figure 13.4 shows a simple, single-processor computer. The processor communicates via a data bus with the main memory and the disk controller. A disk controller can control several disks; we show three disks in this example.

13.2.3 Disk Access Characteristics

Accessing (reading or writing) a block requires three steps, and each step has an associated delay.

1. The disk controller positions the head assembly at the cylinder containing the track on which the block is located. The time to do so is the *seek time*.

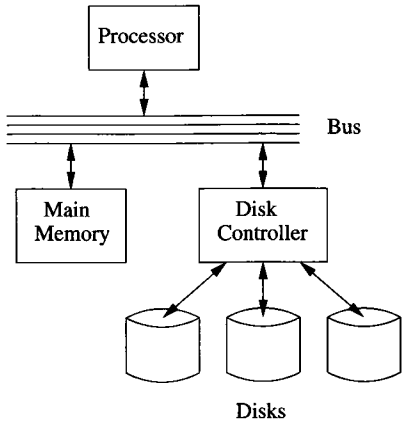

Figure 13.4: Schematic of a simple computer system

2. The disk controller waits while the first sector of the block moves under the head. This time is called the *rotational latency.*

3. All the sectors and the gaps between them pass under the head, while the disk controller reads or writes data in these sectors. This delay is called the *transfer time.*

The sum of the seek time, rotational latency, and transfer time is the *latency* of the disk.

The seek time for a typical disk depends on the distance the heads have to travel from where they are currently located. If they are already at the desired cylinder, the seek time is 0. However, it takes roughly a millisecond to start the disk heads moving, and perhaps 10 milliseconds to move them across all the tracks.

A typical disk rotates once in roughly 10 milliseconds. Thus, rotational latency ranges from 0 to 10 milliseconds, and the average is 5. Transfer times tend to be much smaller, since there are often many blocks on a track. Thus, transfer times are in the sub-millisecond range. When you add all three delays, the typical average latency is about 10 milliseconds, and the maximum latency about twice that.

Example 13.2: Let us examine the time it takes to read a 16,384-byte block from the Megatron 747 disk. First, we need to know some timing properties of the disk:

- The disk rotates at 7200 rpm; i.e., it makes one rotation in 8.33 milliseconds.

- To move the head assembly between cylinders takes one millisecond to start and stop, plus one additional millisecond for every 4000 cylinders

traveled. Thus, the heads move one track in 1.00025 milliseconds and move from the innermost to the outermost track, a distance of 65,536 tracks, in about 17.38 milliseconds.

- Gaps occupy 10% of the space around a track.

Let us calculate the minimum, maximum, and average times to read that 16,384-byte block. The minimum time is just the transfer time. That is, the block might be on a track over which the head is positioned already, and the first sector of the block might be about to pass under the head.

Since there are 4096 bytes per sector on the Megatron 747 (see Example 13.1 for the physical specifications of the disk), the block occupies four sectors. The heads must therefore pass over four sectors and the three gaps between them. We assume that gaps represent 10% of the circle and sectors the remaining 90%. There are 256 gaps and 256 sectors around the circle. Since the gaps together cover 36 degrees of arc and sectors the remaining 324 degrees, the total degrees of arc covered by 3 gaps and 4 sectors is $36 \times 3/256 + 324 \times 4/256 = 5.48$ degrees. The transfer time is thus $(5.48/360) \times 0.00833 = .00013$ seconds. That is, 5.48/360 is the fraction of a rotation needed to read the entire block, and .00833 seconds is the amount of time for a 360-degree rotation.

Now, let us look at the maximum possible time to read the block. In the worst case, the heads are positioned at the innermost cylinder, and the block we want to read is on the outermost cylinder (or vice versa). Thus, the first thing the controller must do is move the heads. As we observed above, the time it takes to move the Megatron 747 heads across all cylinders is about 17.38 milliseconds. This quantity is the seek time for the read.

The worst thing that can happen when the heads arrive at the correct cylinder is that the beginning of the desired block has just passed under the head. Assuming we must read the block starting at the beginning, we have to wait essentially a full rotation, or 8.33 milliseconds, for the beginning of the block to reach the head again. Once that happens, we have only to wait an amount equal to the transfer time, 0.13 milliseconds, to read the entire block. Thus, the worst-case latency is $17.38 + 8.33 + 0.13 = 25.84$ milliseconds.

Last, let us compute the average latency. Two of the components of the latency are easy to compute: the transfer time is always 0.13 milliseconds, and the average rotational latency is the time to rotate the disk half way around, or 4.17 milliseconds. We might suppose that the average seek time is just the time to move across half the tracks. However, that is not quite right, since typically, the heads are initially somewhere near the middle and therefore will have to move less than half the distance, on average, to the desired cylinder. We leave it as an exercise to show that the average distance traveled is 1/3 of the way across the disk.

The time it takes the Megatron 747 to move 1/3 of the way across the disk is $1 + (65536/3)/4000 = 6.46$ milliseconds. Our estimate of the average latency is thus $6.46 + 4.17 + 0.13 = 10.76$ milliseconds; the three terms represent average seek time, average rotational latency, and transfer time, respectively. □

13.2.4 Exercises for Section 13.2

Exercise 13.2.1: The *Megatron 777* disk has the following characteristics:

1. There are ten surfaces, with 100,000 tracks each.
2. Tracks hold an average of 1000 sectors of 1024 bytes each.
3. 20% of each track is used for gaps.
4. The disk rotates at 10,000 rpm.
5. The time it takes the head to move n tracks is $1 + 0.0002n$ milliseconds.

Answer the following questions about the Megatron 777.

a) What is the capacity of the disk?

b) If tracks are located on the outer inch of a 3.5-inch-diameter surface, what is the average density of bits in the sectors of a track?

c) What is the maximum seek time?

d) What is the maximum rotational latency?

e) If a block is 65,546 bytes (i.e., 64 sectors), what is the transfer time of a block?

! f) What is the average seek time?

g) What is the average rotational latency?

! **Exercise 13.2.2:** Suppose the Megatron 747 disk head is at cylinder 8192, i.e., 1/8 of the way across the cylinders. Suppose that the next request is for a block on a random cylinder. Calculate the average time to read this block.

!! **Exercise 13.2.3:** Prove that if we move the head from a random cylinder to another random cylinder, the average distance we move is 1/3 of the way across the disk (neglecting edge effects due to the fact that the number of cylinders is finite).

!! **Exercise 13.2.4:** Exercise 13.2.3 assumes that we move from a random track to another random track. Suppose, however, that the number of sectors per track is proportional to the length (or radius) of the track, so the bit density is the same for all tracks. Suppose also that we need to move the head from a random *sector* to another random sector. Since the sectors tend to congregate at the outside of the disk, we might expect that the average head move would be less than 1/3 of the way across the tracks. Assuming that tracks occupy radii from 0.75 inches to 1.75 inches, calculate the average number of tracks the head travels when moving between two random sectors.

! **Exercise 13.2.5:** To modify a block on disk, we must read it into main memory, perform the modification, and write it back. Assume that the modification in main memory takes less time than it does for the disk to rotate, and that the disk controller postpones other requests for disk access until the block is ready to be written back to the disk. For the Megatron 747 disk, what is the time to modify a block?

13.3 Accelerating Access to Secondary Storage

Just because a disk takes an average of, say, 10 milliseconds to access a block, it does not follow that an application such as a database system will get the data it requests 10 milliseconds after the request is sent to the disk controller. If there is only one disk, the disk may be busy with another access for the same process or another process. In the worst case, a request for a disk access arrives more than once every 10 milliseconds, and these requests back up indefinitely. In that case, the *scheduling latency* becomes infinite.

There are several things we can do to decrease the average time a disk access takes, and thus improve the *throughput* (number of disk accesses per second that the system can accomodate). We begin this section by arguing that the "I/O model" is the right one for measuring the time database operations take. Then, we consider a number of techniques for speeding up typical database accesses to disk:

1. Place blocks that are accessed together on the same cylinder, so we can often avoid seek time, and possibly rotational latency as well.

2. Divide the data among several smaller disks rather than one large one. Having more head assemblies that can go after blocks independently can increase the number of block accesses per unit time.

3. "Mirror" a disk: making two or more copies of the data on different disks. In addition to saving the data in case one of the disks fails, this strategy, like dividing the data among several disks, lets us access several blocks at once.

4. Use a disk-scheduling algorithm, either in the operating system, in the DBMS, or in the disk controller, to select the order in which several requested blocks will be read or written.

5. Prefetch blocks to main memory in anticipation of their later use.

13.3.1 The I/O Model of Computation

Let us imagine a simple computer running a DBMS and trying to serve a number of users who are performing queries and database modifications. For the moment, assume our computer has one processor, one disk controller, and

one disk. The database itself is much too large to fit in main memory. Key parts of the database may be buffered in main memory, but generally, each piece of the database that one of the users accesses will have to be retrieved initially from disk. The following rule, which defines the *I/O model of computation*, can thus be assumed.

> **Dominance of I/O cost**: The time taken to perform a disk access is much larger than the time likely to be used manipulating that data in main memory. Thus, the number of block accesses (*Disk I/O's*) is a good approximation to the time needed by the algorithm and should be minimized.

Example 13.3: Suppose our database has a relation R and a query asks for the tuple of R that has a certain key value k. It is quite desirable to have an index on R to identify the disk block on which the tuple with key value k appears. However it is generally unimportant whether the index tells us where on the block this tuple appears.

For instance, if we assume a Megatron 747 disk, it will take on the order of 11 milliseconds to read a 16K-byte block. In 11 milliseconds, a modern microprocessor can execute millions of instructions. However, searching for the key value k once the block is in main memory will only take thousands of instructions, even if the dumbest possible linear search is used. The additional time to perform the search in main memory will therefore be less than 1% of the block access time and can be neglected safely. □

13.3.2 Organizing Data by Cylinders

Since seek time represents about half the time it takes to access a block, it makes sense to store data that is likely to be accessed together, such as relations, on a single cylinder, or on as many adjacent cylinders as are needed. In fact, if we choose to read all the blocks on a single track or on a cylinder consecutively, then we can neglect all but the first seek time (to move to the cylinder) and the first rotational latency (to wait until the first of the blocks moves under the head). In that case, we can approach the theoretical transfer rate for moving data on or off the disk.

Example 13.4: Suppose relation R requires 1024 blocks of a Megatron 747 disk to hold its tuples. Suppose also that we need to access all the tuples of R; for example we may be doing a search without an index or computing a sum of the values of a particular attribute of R. If the blocks holding R are distributed around the disk at random, then we shall need an average latency (10.76 milliseconds — see Example 13.2) to access each, for a total of 11 seconds.

However, 1024 blocks are exactly one cylinder of the Megatron 747. We can access them all by performing one average seek (6.46 milliseconds), after which we can read the blocks in some order, one right after another. We can read all the blocks on a cylinder in 16 rotations of the disk, since there are 16 tracks.

Sixteen rotations take $16 \times 8.33 = 133$ milliseconds. The total time to access R is thus about 139 milliseconds, and we speed up the operation on R by a factor of about 80. □

13.3.3 Using Multiple Disks

We can often improve the performance of our system if we replace one disk, with many heads locked together, by several disks with their independent heads. The arrangement was suggested in Fig. 13.4, where we showed three disks connected to a single controller. As long as the disk controller, bus, and main memory can handle n times the data-transfer rate, then n disks will have approximately the performance of one disk that operates n times as fast.

Thus, using several disks can increase the ability of a database system to handle heavy loads of disk-access requests. However, as long as the system is not overloaded (when requests will queue up and are delayed for a long time or ignored), there is no change in how long it takes to perform any single block access. If we have several disks, then the technique known as *striping* (described in the next example) will speed up access to large database objects — those that occupy a large number of blocks.

Example 13.5: Suppose we have four Megatron 747 disks and want to access the relation R of Example 13.4 faster than the 139-millisecond time that was suggested for storing R on one cylinder of one disk. We can "stripe" R by dividing it among the four disks. The first disk can receive blocks $1, 5, 9, \ldots$ of R, the second disk holds blocks $2, 6, 10, \ldots$, the third holds blocks $3, 7, 11, \ldots$, and the last disk holds blocks $4, 8, 12, \ldots$, as suggested by Fig. 13.5. Let us contrive that on each of the disks, all the blocks of R are on four tracks of a single cylinder.

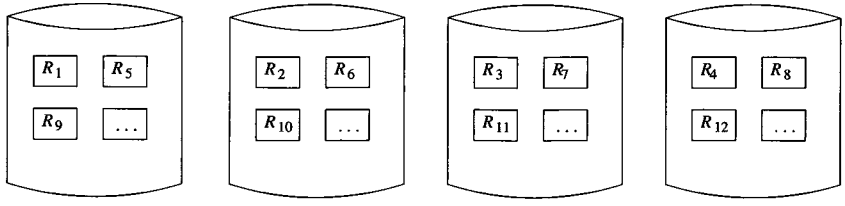

Figure 13.5: Striping a relation across four disks

Then to retrieve the 256 blocks of R on one of the disks requires an average seek time (6.46 milliseconds) plus four rotations of the disk, one rotation for each track. That is $6.46 + 4 \times 8.33 = 39.8$ milliseconds. Of course we have to wait for the last of the four disks to finish, and there is a high probability that one will take substantially more seek time than average. However, we should get a speedup in the time to access R by about a factor of three on the average, when there are four disks. □

13.3.4 Mirroring Disks

There are situations where it makes sense to have two or more disks hold identical copies of data. The disks are said to be *mirrors* of each other. One important motivation is that the data will survive a head crash by either disk, since it is still readable on a mirror of the disk that crashed. Systems designed to enhance reliability often use pairs of disks as mirrors of each other.

If we have n disks, each holding the same data, then the rate at which we can read blocks goes up by a factor of n, since the disk controller can assign a read request to any of the n disks. In fact, the speedup could be even greater than n, if a clever controller chooses to read a block from the disk whose head is currently closest to that block. Unfortunately, the writing of disk blocks does not speed up at all. The reason is that the new block must be written to each of the n disks.

13.3.5 Disk Scheduling and the Elevator Algorithm

Another effective way to improve the throughput of a disk system is to have the disk controller choose which of several requests to execute first. This approach cannot be used if accesses have to be made in a certain sequence, but if the requests are from independent processes, they can all benefit, on the average, from allowing the scheduler to choose among them judiciously.

A simple and effective way to schedule large numbers of block requests is known as the *elevator algorithm*. We think of the disk head as making sweeps across the disk, from innermost to outermost cylinder and then back again, just as an elevator makes vertical sweeps from the bottom to top of a building and back again. As heads pass a cylinder, they stop if there are one or more requests for blocks on that cylinder. All these blocks are read or written, as requested. The heads then proceed in the same direction they were traveling until the next cylinder with blocks to access is encountered. When the heads reach a position where there are no requests ahead of them in their direction of travel, they reverse direction.

Example 13.6: Suppose we are scheduling a Megatron 747 disk, which we recall has average seek, rotational latency, and transfer times of 6.46, 4.17, and 0.13, respectively (in this example, all times are in milliseconds). Suppose that at some time there are pending requests for block accesses at cylinders 8000, 24,000, and 56,000. The heads are located at cylinder 8000. In addition, there are three more requests for block accesses that come in at later times, as summarized in Fig. 13.6. For instance, the request for a block from cylinder 16,000 is made at time 10 milliseconds.

We shall assume that each block access incurs time 0.13 for transfer and 4.17 for average rotational latency, i.e., we need 4.3 milliseconds plus whatever the seek time is for each block access. The seek time can be calculated by the rule for the Megatron 747 given in Example 13.2: 1 plus the number of tracks divided by 4000. Let us see what happens if we schedule disk accesses using

Cylinder of request	First time available
8000	0
24000	0
56000	0
16000	10
64000	20
40000	30

Figure 13.6: Arrival times for four block-access requests

the elevator algorithm. The first request, at cylinder 8000, requires no seek, since the heads are already there. Thus, at time 4.3 the first access will be complete. The request for cylinder 16,000 has not arrived at this point, so we move the heads to cylinder 24,000, the next requested "stop" on our sweep to the highest-numbered tracks. The seek from cylinder 8000 to 24,000 takes 5 milliseconds, so we arrive at time 9.3 and complete the access in another 4.3. Thus, the second access is complete at time 13.6. By this time, the request for cylinder 16,000 has arrived, but we passed that cylinder at time 7.3 and will not come back to it until the next pass.

We thus move next to cylinder 56,000, taking time 9 to seek and 4.3 for rotation and transfer. The third access is thus complete at time 26.9. Now, the request for cylinder 64,000 has arrived, so we continue outward. We require 3 milliseconds for seek time, so this access is complete at time 26.9+3+4.3 = 34.2.

At this time, the request for cylinder 40,000 has been made, so it and the request at cylinder 16,000 remain. We thus sweep inward, honoring these two requests. Figure 13.7 summarizes the times at which requests are honored.

Cylinder of request	Time completed
8000	4.3
24000	13.6
56000	26.9
64000	34.2
40000	45.5
16000	56.8

Figure 13.7: Finishing times for block accesses using the elevator algorithm

Let us compare the performance of the elevator algorithm with a more naive approach such as first-come-first-served. The first three requests are satisfied in exactly the same manner, assuming that the order of the first three requests was 8000, 24,000, and 56,000. However, at that point, we go to cylinder 16,000,

because that was the fourth request to arrive. The seek time is 11 for this request, since we travel from cylinder 56,000 to 16,000, more than half way across the disk. The fifth request, at cylinder 64,000, requires a seek time of 13, and the last, at 40,000, uses seek time 7. Figure 13.8 summarizes the activity caused by first-come-first-served scheduling. The difference between the two algorithms — 14 milliseconds — may not appear significant, but recall that the number of requests in this simple example is small and the algorithms were assumed not to deviate until the fourth of the six requests. □

Cylinder of request	Time completed
8000	4.3
24000	13.6
56000	26.9
16000	42.2
64000	59.5
40000	70.8

Figure 13.8: Finishing times for block accesses using the first-come-first-served algorithm

13.3.6 Prefetching and Large-Scale Buffering

Our final suggestion for speeding up some secondary-memory algorithms is called *prefetching* or sometimes *double buffering*. In some applications we can predict the order in which blocks will be requested from disk. If so, then we can load them into main memory buffers before they are needed. One advantage to doing so is that we are thus better able to schedule the disk, such as by using the elevator algorithm, to reduce the average time needed to access a block. In the extreme case, where there are many access requests waiting at all times, we can make the seek time per request be very close to the minimum seek time, rather than the average seek time.

13.3.7 Exercises for Section 13.3

Exercise 13.3.1: Suppose we are scheduling I/O requests for a Megatron 747 disk, and the requests in Fig. 13.9 are made, with the head initially at track 32,000. At what time is each request serviced fully if:

a) We use the elevator algorithm (it is permissible to start moving in either direction at first).

b) We use first-come-first-served scheduling.

Cylinder of Request	First time available
8000	0
48000	1
4000	10
40000	20

Figure 13.9: Arrival times for four block-access requests

! **Exercise 13.3.2:** Suppose we use two Megatron 747 disks as mirrors of one another. However, instead of allowing reads of any block from either disk, we keep the head of the first disk in the inner half of the cylinders, and the head of the second disk in the outer half of the cylinders. Assuming read requests are on random tracks, and we never have to write:

a) What is the average rate at which this system can read blocks?

b) How does this rate compare with the average rate for mirrored Megatron 747 disks with no restriction?

c) What disadvantages do you foresee for this system?

! **Exercise 13.3.3:** Let us explore the relationship between the arrival rate of requests, the throughput of the elevator algorithm, and the average delay of requests. To simplify the problem, we shall make the following assumptions:

1. A pass of the elevator algorithm always proceeds from the innermost to outermost track, or vice-versa, even if there are no requests at the extreme cylinders.

2. When a pass starts, only those requests that are already pending will be honored, not requests that come in while the pass is in progress, even if the head passes their cylinder.[2]

3. There will never be two requests for blocks on the same cylinder waiting on one pass.

Let A be the interarrival rate, that is the time between requests for block accesses. Assume that the system is in steady state, that is, it has been accepting and answering requests for a long time. For a Megatron 747 disk, compute as a function of A:

[2]The purpose of this assumption is to avoid having to deal with the fact that a typical pass of the elevator algorithm goes fast at first, as there will be few waiting requests where the head has recently been, and slows down as it moves into an area of the disk where it has not recently been. The analysis of the way request density varies during a pass is an interesting exercise in its own right.

a) The average time taken to perform one pass.

b) The number of requests serviced on one pass.

c) The average time a request waits for service.

!! **Exercise 13.3.4:** In Example 13.5, we saw how dividing the data to be sorted among four disks could allow more than one block to be read at a time. Suppose our data is divided randomly among n disks, and requests for data are also random. Requests must be executed in the order in which they are received because there are dependencies among them that must be respected (see Chapter 18, for example, for motivation for this constraint). What is the average throughput for such a system?

! **Exercise 13.3.5:** If we read k randomly chosen blocks from one cylinder, on the average how far around the cylinder must we go before we pass all of the blocks?

13.4 Disk Failures

In this section we shall consider the ways in which disks can fail and what can be done to mitigate these failures.

1. The most common form of failure is an *intermittent failure*, where an attempt to read or write a sector is unsuccessful, but with repeated tries we are able to read or write successfully.

2. A more serious form of failure is one in which a bit or bits are permanently corrupted, and it becomes impossible to read a sector correctly no matter how many times we try. This form of error is called *media decay*.

3. A related type of error is a *write failure*, where we attempt to write a sector, but we can neither write successfully nor can we retrieve the previously written sector. A possible cause is that there was a power outage during the writing of the sector.

4. The most serious form of disk failure is a *disk crash*, where the entire disk becomes unreadable, suddenly and permanently.

We shall discuss parity checks as a way to detect intermittent failures. We also discuss "stable storage," a technique for organizing a disk so that media decays or failed writes do not result in permanent loss. Finally, we examine techniques collectively known as "RAID" for coping with disk crashes.

13.4.1 Intermittent Failures

An intermittent failure occurs if we try to read a sector, but the correct content of that sector is not delivered to the disk controller. If the controller has a way to tell that the sector is good or bad (as we shall discuss in Section 13.4.2), then the controller can reissue the read request when bad data is read, until the sector is returned correctly, or some preset limit, like 100 tries, is reached.

Similarly, the controller may attempt to write a sector, but the contents of the sector are not what was intended. The only way to check that the write was correct is to let the disk go around again and read the sector. A straightforward way to perform the check is to read the sector and compare it with the sector we intended to write. However, instead of performing the complete comparison at the disk controller, it is simpler to read the sector and see if a good sector was read. If so, we assume the write was correct, and if the sector read is bad, then the write was apparently unsuccessful and must be repeated.

13.4.2 Checksums

How a reading operation can determine the good/bad status of a sector may appear mysterious at first. Yet the technique used in modern disk drives is quite simple: each sector has some additional bits, called the *checksum*, that are set depending on the values of the data bits stored in that sector. If, on reading, we find that the checksum is not proper for the data bits, then we know there is an error in reading. If the checkum is proper, there is still a small chance that the block was not read correctly, but by using many checksum bits we can make the probability of missing a bad read arbitrarily small.

A simple form of checksum is based on the *parity* of all the bits in the sector. If there is an odd number of 1's among a collection of bits, we say the bits have *odd* parity and add a parity bit that is 1. Similarly, if there is an even number of 1's among the bits, then we say the bits have *even* parity and add parity bit 0. As a result:

- The number of 1's among a collection of bits and their parity bit is always even.

When we write a sector, the disk controller can compute the parity bit and append it to the sequence of bits written in the sector. Thus, every sector will have even parity.

Example 13.7: If the sequence of bits in a sector were 01101000, then there is an odd number of 1's, so the parity bit is 1. If we follow this sequence by its parity bit we have 011010001. If the given sequence of bits were 11101110, we have an even number of 1's, and the parity bit is 0. The sequence followed by its parity bit is 111011100. Note that each of the nine-bit sequences constructed by adding a parity bit has even parity. □

Any one-bit error in reading or writing the bits and their parity bit results in a sequence of bits that has *odd parity*; i.e., the number of 1's is odd. It is easy for the disk controller to count the number of 1's and to determine the presence of an error if a sector has odd parity.

Of course, more than one bit of the sector may be corrupted. If so, the probability is 50% that the number of 1-bits will be even, and the error will not be detected. We can increase our chances of detecting errors if we keep several parity bits. For example, we could keep eight parity bits, one for the first bit of every byte, one for the second bit of every byte, and so on, up to the eighth and last bit of every byte. Then, on a massive error, the probability is 50% that any one parity bit will detect an error, and the chance that none of the eight do so is only one in 2^8, or $1/256$. In general, if we use n independent bits as a checksum, then the chance of missing an error is only $1/2^n$. For instance, if we devote 4 bytes to a checksum, then there is only one chance in about four billion that the error will go undetected.

13.4.3 Stable Storage

While checksums will almost certainly detect the existence of a media failure or a failure to read or write correctly, it does not help us correct the error. Moreover, when writing we could find ourselves in a position where we overwrite the previous contents of a sector and yet cannot read the new contents correctly. That situation could be serious if, say, we were adding a small increment to an account balance and now have lost both the original balance and the new balance. If we could be assured that the contents of the sector contained either the new or old balance, then we would only have to determine whether the write was successful or not.

To deal with the problems above, we can implement a policy known as *stable storage* on a disk or on several disks. The general idea is that sectors are paired, and each pair represents one sector-contents X. We shall refer to the pair of sectors representing X as the "left" and "right" copies, X_L and X_R. We continue to assume that the copies are written with a sufficient number of parity-check bits so that we can rule out the possibility that a bad sector looks good when the parity checks are considered. Thus, we shall assume that if the read function returns a good value w for either X_L or X_R, then w is the true value of X. The stable-storage writing policy is:

1. Write the value of X into X_L. Check that the value has status "good"; i.e., the parity-check bits are correct in the written copy. If not, repeat the write. If after a set number of write attempts, we have not successfully written X into X_L, assume that there is a media failure in this sector. A fix-up such as substituting a spare sector for X_L must be adopted.

2. Repeat (1) for X_R.

The stable-storage reading policy is to alternate trying to read X_L and X_R,

until a good value is returned. Only if no good value is returned after some large, prechosen number of tries, is X truly unreadable.

13.4.4 Error-Handling Capabilities of Stable Storage

The policies described in Section 13.4.3 are capable of compensating for several different kinds of errors. We shall outline them here.

1. *Media failures.* If, after storing X in sectors X_L and X_R, one of them undergoes a media failure and becomes permanently unreadable, we can always read X from the other. If both X_L and X_R have failed, then we cannot read X, but the probability of both failing is extremely small.

2. *Write failure.* Suppose that as we write X, there is a system failure — e.g., a power outage. It is possible that X will be lost in main memory, and also the copy of X being written at the time will be garbled. For example, half the sector may be written with part of the new value of X, while the other half remains as it was. When the system becomes available and we examine X_L and X_R, we are sure to be able to determine either the old or new value of X. The possible cases are:

   (a) The failure occurred as we were writing X_L. Then we shall find that the status of X_L is "bad." However, since we never got to write X_R, its status will be "good" (unless there is a coincident media failure at X_R, which is extremely unlikely). Thus, we can obtain the old value of X. We may also copy X_R into X_L to repair the damage to X_L.

   (b) The failure occurred after we wrote X_L. Then we expect that X_L will have status "good," and we may read the new value of X from X_L. Since X_R may or may not have the correct value of X, we should also copy X_L into X_R.

13.4.5 Recovery from Disk Crashes

The most serious mode of failure for disks is the "disk crash" or "head crash," where data is permanently destroyed. If the data was not backed up on another medium, such as a tape backup system, or on a mirror disk as we discussed in Section 13.3.4, then there is nothing we can do to recover the data. This situation represents a disaster for many DBMS applications, such as banking and other financial applications.

Several schemes have been developed to reduce the risk of data loss by disk crashes. They generally involve redundancy, extending the idea of parity checks from Section 13.4.2 or duplicated sectors, as in Section 13.4.3. The common term for this class of strategies is RAID, or *Redundant Arrays of Independent Disks.*

The rate at which disk crashes occur is generally measured by the *mean time to failure*, the time after which 50% of a population of disks can be expected to fail and be unrecoverable. For modern disks, the mean time to failure is about 10 years. We shall make the convenient assumption that if the mean time to failure is n years, then in any given year, $1/n$th of the surviving disks fail. In reality, there is a tendency for disks, like most electronic equipment, to fail early or fail late. That is, a small percentage have manufacturing defects that lead to their early demise, while those without such defects will survive for many years, until wear-and-tear causes a failure.

However, the mean time to a disk crash does not have to be the same as the mean time to data loss. The reason is that there are a number of schemes available for assuring that if one disk fails, there are others to help recover the data of the failed disk. In the remainder of this section, we shall study the most common schemes.

Each of these schemes starts with one or more disks that hold the data (we'll call these the *data disks*) and adding one or more disks that hold information that is completely determined by the contents of the data disks. The latter are called *redundant disks*. When there is a disk crash of either a data disk or a redundant disk, the other disks can be used to restore the failed disk, and there is no permanent information loss.

13.4.6 Mirroring as a Redundancy Technique

The simplest scheme is to mirror each disk, as discussed in Section 13.3.4. We shall call one of the disks the *data disk*, while the other is the *redundant disk*; which is which doesn't matter in this scheme. Mirroring, as a protection against data loss, is often referred to as *RAID level 1*. It gives a mean time to memory loss that is much greater than the mean time to disk failure, as the following example illustrates. Essentially, with mirroring and the other redundancy schemes we discuss, the only way data can be lost is if there is a second disk crash while the first crash is being repaired.

Example 13.8: Suppose each disk has a 10-year mean time to failure, which we shall take to mean that the probability of failure in any given year is 10%. If disks are mirrored, then when a disk fails, we have only to replace it with a good disk and copy the mirror disk to the new one. At the end, we have two disks that are mirrors of each other, and the system is restored to its former state.

The only thing that could go wrong is that during the copying the mirror disk fails. Now, both copies of at least part of the data have been lost, and there is no way to recover.

But how often will this sequence of events occur? Suppose that the process of replacing the failed disk takes 3 hours, which is 1/8 of a day, or 1/2920 of a year. Since we assume the average disk lasts 10 years, the probability that the mirror disk will fail during copying is $(1/10) \times (1/2920)$, or one in 29,200. If

one disk fails every 10 years, then one of the two disks will fail once in 5 years on the average. One in every 29,200 of these failures results in data loss. Put another way, the mean time to a failure involving data loss is $5 \times 29{,}200 = 146{,}000$ years. □

13.4.7 Parity Blocks

While mirroring disks is an effective way to reduce the probability of a disk crash involving data loss, it uses as many redundant disks as there are data disks. Another approach, often called *RAID level 4*, uses only one redundant disk, no matter how many data disks there are. We assume the disks are identical, so we can number the blocks on each disk from 1 to some number n. Of course, all the blocks on all the disks have the same number of bits; for instance, the 16,384-byte blocks of the Megatron 747 have $8 \times 16{,}384 = 131{,}072$ bits. In the redundant disk, the ith block consists of parity checks for the ith blocks of all the data disks. That is, the jth bits of all the ith blocks, including both the data disks and the redundant disk, must have an even number of 1's among them, and we always choose the bit of the redundant disk to make this condition true.

We saw in Example 13.7 how to force the condition to be true. In the redundant disk, we choose bit j to be 1 if an odd number of the data disks have 1 in that bit, and we choose bit j of the redundant disk to be 0 if there are an even number of 1's in that bit among the data disks. The term for this calculation is the *modulo-2 sum*. That is, the modulo-2 sum of bits is 0 if there are an even number of 1's among those bits, and 1 if there are an odd number of 1's.

Example 13.9: Suppose for sake of an extremely simple example that blocks consist of only one byte — eight bits. Let there be three data disks, called 1, 2, and 3, and one redundant disk, called disk 4. Focus on the first block of all these disks. If the data disks have in their first blocks the following bit sequences:

disk 1: 11110000
disk 2: 10101010
disk 3: 00111000

then the redundant disk will have in block 1 the parity check bits:

disk 4: 01100010

Notice how in each position, an even number of the four 8-bit sequences have 1's. There are two 1's in positions 1, 2, 4, 5, and 7, four 1's in position 3, and zero 1's in positions 6 and 8. □

Reading

Reading blocks from a data disk is no different from reading blocks from any disk. There is generally no reason to read from the redundant disk, but we could.

Writing

When we write a new block of a data disk, we need not only to change that block, but we need to change the corresponding block of the redundant disk so it continues to hold the parity checks for the corresponding blocks of all the data disks. A naive approach would read the corresponding blocks of the n data disks, take their modulo-2 sum, and rewrite the block of the redundant disk. That approach requires a write of the data block that is rewritten, the reading of the $n-1$ other data blocks, and a write of the block of the redundant disk. The total is thus $n+1$ disk I/O's.

A better approach is to look only at the old and new versions of the data block i being rewritten. If we take their modulo-2 sum, we know in which positions there is a change in the number of 1's among the blocks numbered i on all the disks. Since these changes are always by one, any even number of 1's changes to an odd number. If we change the same positions of the redundant block, then the number of 1's in each position becomes even again. We can perform these calculations using four disk I/O's:

1. Read the old value of the data block being changed.
2. Read the corresponding block of the redundant disk.
3. Write the new data block.
4. Recalculate and write the block of the redundant disk.

Example 13.10: Suppose the three first blocks of the data disks are as in Example 13.9:

disk 1: 11110000
disk 2: 10101010
disk 3: 00111000

Suppose also that the block on the second disk changes from 10101010 to 11001100. We take the modulo-2 sum of the old and new values of the block on disk 2, to get 01100110. That tells us we must change positions 2, 3, 6, and 7 of the first block of the redundant disk. We read that block: 01100010. We replace this block by a new block that we get by changing the appropriate positions; in effect we replace the redundant block by the modulo-2 sum of itself and 01100110, to get 00000100. Another way to express the new redundant block is that it is the modulo-2 sum of the old and new versions of the block

The Algebra of Modulo-2 Sums

It may be helpful for understanding some of the tricks used with parity checks to know the algebraic rules involving the modulo-2 sum operation on bit vectors. We shall denote this operation $\oplus$. As an example, $1100 \oplus 1010 = 0110$. Here are some useful rules about $\oplus$:

- The *commutative law*: $x \oplus y = y \oplus x$.
- The *associative law*: $x \oplus (y \oplus z) = (x \oplus y) \oplus z$.
- The all-0 vector of the appropriate length, which we denote $\bar{0}$, is the *identity* for $\oplus$; that is, $x \oplus \bar{0} = \bar{0} \oplus x = x$.
- $\oplus$ is its own inverse: $x \oplus x = \bar{0}$. As a useful consequence, if $x \oplus y = z$, then we can "add" x to both sides and get $y = x \oplus z$.

being rewritten and the old value of the redundant block. In our example, the first blocks of the four disks — three data disks and one redundant — have become:

disk 1: 11110000
disk 2: 11001100
disk 3: 00111000
disk 4: 00000100

after the write to the block on the second disk and the necessary recomputation of the redundant block. Notice that in the blocks above, each column continues to have an even number of 1's. □

Failure Recovery

Now, let us consider what we would do if one of the disks crashed. If it is the redundant disk, we swap in a new disk, and recompute the redundant blocks. If the failed disk is one of the data disks, then we need to swap in a good disk and recompute its data from the other disks. The rule for recomputing any missing data is actually simple, and doesn't depend on which disk, data or redundant, is failed. Since we know that the number of 1's among corresponding bits of all disks is even, it follows that:

- The bit in any position is the modulo-2 sum of all the bits in the corresponding positions of all the other disks.

If one doubts the above rule, one has only to consider the two cases. If the bit in question is 1, then the number of corresponding bits in the other disks

that are 1 must be odd, so their modulo-2 sum is 1. If the bit in question is 0, then there are an even number of 1's among the corresponding bits of the other disks, and their modulo-2 sum is 0.

Example 13.11: Suppose that disk 2 fails. We need to recompute each block of the replacement disk. Following Example 13.9, let us see how to recompute the first block of the second disk. We are given the corresponding blocks of the first and third data disks and the redundant disk, so the situation looks like:

disk 1: 11110000
disk 2: ????????
disk 3: 00111000
disk 4: 01100010

If we take the modulo-2 sum of each column, we deduce that the missing block is 10101010, as was initially the case in Example 13.9. □

13.4.8 An Improvement: RAID 5

The RAID level 4 strategy described in Section 13.4.7 effectively preserves data unless there are two almost simultaneous disk crashes. However, it suffers from a bottleneck defect that we can see when we re-examine the process of writing a new data block. Whatever scheme we use for updating the disks, we need to read and write the redundant disk's block. If there are n data disks, then the number of disk writes to the redundant disk will be n times the average number of writes to any one data disk.

However, as we observed in Example 13.11, the rule for recovery is the same as for the data disks and redundant disks: take the modulo-2 sum of corresponding bits of the other disks. Thus, we do not have to treat one disk as the redundant disk and the others as data disks. Rather, we could treat each disk as the redundant disk for some of the blocks. This improvement is often called *RAID level 5*.

For instance, if there are $n+1$ disks numbered 0 through n, we could treat the ith cylinder of disk j as redundant if j is the remainder when i is divided by $n+1$.

Example 13.12: In our running example, $n = 3$ so there are 4 disks. The first disk, numbered 0, is redundant for its cylinders numbered 4, 8, 12, and so on, because these are the numbers that leave remainder 0 when divided by 4. The disk numbered 1 is redundant for blocks numbered 1, 5, 9, and so on; disk 2 is redundant for blocks 2, 6, 10, ..., and disk 3 is redundant for 3, 7, 11, ...

As a result, the reading and writing load for each disk is the same. If all blocks are equally likely to be written, then for one write, each disk has a 1/4 chance that the block is on that disk. If not, then it has a 1/3 chance that it will be the redundant disk for that block. Thus, each of the four disks is involved in $1/4 + (3/4) \times (1/3) = 1/2$ of the writes. □