# Advanced Topics in Relational Databases

This chapter introduces additional topics that are of interest to the database programmer. We begin with a section on the SQL standard for authorization of access to database elements. Next, we see the SQL extension that allows for recursive programming in SQL — queries that use their own results. Then, we look at the object-relational model, and how it is implemented in the SQL standard.

The remainder of the chapter concerns "OLAP," or on-line analytic processing. OLAP refers to complex queries of a nature that causes them to take significant time to execute. Because they are so expensive, some special technology has developed to handle them efficiently. One important direction is an implementation of relations, called the "data cube," that is rather different from the conventional bag-of-tuples approach of SQL.

## 1 Security and User Authorization in SQL

SQL postulates the existence of *authorization ID's*, which are essentially user names. SQL also has a special authorization ID called `PUBLIC`, which includes any user. Authorization ID's may be granted privileges, much as they would be in the file system environment maintained by an operating system. For example, a UNIX system generally controls three kinds of privileges: read, write, and execute. That list of privileges makes sense, because the protected objects of a UNIX system are files, and these three operations characterize well the things one typically does with files. However, databases are much more complex than file systems, and the kinds of privileges used in SQL are correspondingly more complex.

In this section, we shall first learn what privileges SQL allows on database elements. We shall then see how privileges may be acquired by users (by

authorization ID's, that is). Finally, we shall see how privileges may be taken away.

## 1.1 Privileges

SQL defines nine types of privileges: `SELECT`, `INSERT`, `DELETE`, `UPDATE`, `REFERENCES`, `USAGE`, `TRIGGER`, `EXECUTE`, and `UNDER`. The first four of these apply to a relation, which may be either a base table or a view. As their names imply, they give the holder of the privilege the right to query (select from) the relation, insert into the relation, delete from the relation, and update tuples of the relation, respectively.

A SQL statement cannot be executed without the privileges appropriate to that statement; e.g., a select-from-where statement requires the `SELECT` privilege on every table it accesses. We shall see how the module can get those privileges shortly. `SELECT`, `INSERT`, and `UPDATE` may also have an associated list of attributes, for instance, `SELECT(name, addr)`. If so, then only those attributes may be seen in a selection, specified in an insertion, or changed in an update, respectively. Note that, when granted, privileges such as these will be associated with a particular relation, so it will be clear at that time to what relation attributes `name` and `addr` belong.

The `REFERENCES` privilege on a relation is the right to refer to that relation in an integrity constraint. These constraints may take any of the forms, such as assertions, attribute- or tuple-based checks, or referential integrity constraints. The `REFERENCES` privilege may also have an attached list of attributes, in which case only those attributes may be referenced in a constraint. A constraint cannot be created unless the owner of the schema in which the constraint appears has the `REFERENCES` privilege on all data involved in the constraint.

`USAGE` is a privilege that applies to several kinds of schema elements other than relations and assertions; it is the right to use that element in one's own declarations. The `TRIGGER` privilege on a relation is the right to define triggers on that relation. `EXECUTE` is the right to execute a piece of code, such as a PSM procedure or function. Finally, `UNDER` is the right to create subtypes of a given type. The matter of types appears in Section 4.

**Example 1:** Let us consider what privileges are needed to execute the insertion statement of Fig. 1. First, it is an insertion into the relation `Studio`, so we require an `INSERT` privilege on `Studio`. However, since the insertion specifies only the component for attribute `name`, it is acceptable to have either the privilege `INSERT` or the privilege `INSERT(name)` on relation `Studio`. The latter privilege allows us to insert `Studio` tuples that specify only the `name` component and leave other components to take their default value or `NULL`, which is what Fig. 1 does.

However, notice that the insertion statement of Fig. 1 involves two subqueries, starting at lines (2) and (5). To carry out these selections we require

### Triggers and Privileges

It is a bit subtle how privileges are handled for triggers. First, if you have the `TRIGGER` privilege for a relation, you can attempt to create any trigger you like on that relation. However, since the condition and action portions of the trigger are likely to query and/or modify portions of the database, the trigger creator must have the necessary privileges for those actions. When someone performs an activity that awakens the trigger, they do not need the privileges that the trigger condition and action require; the trigger is executed under the privileges of its creator.

```
1) INSERT INTO Studio(name)
2)     SELECT DISTINCT studioName
3)     FROM Movies
4)     WHERE studioName NOT IN
5)         (SELECT name
6)          FROM Studio);
```

Figure 1: Adding new studios

the privileges needed for the subqueries. Thus, we need the `SELECT` privilege on both relations involved in `FROM` clauses: `Movies` and `Studio`. Note that just because we have the `INSERT` privilege on `Studio` doesn't mean we have the `SELECT` privilege on `Studio`, or vice versa. Since it is only particular attributes of `Movies` and `Studio` that get selected, it is sufficient to have the privilege `SELECT(studioName)` on `Movies` and the privilege `SELECT(name)` on `Studio`, or privileges that include these attributes within a list of attributes. □

## 1.2 Creating Privileges

There are two aspects to the awarding of privileges: how they are created initially, and how they are passed from user to user. We shall discuss initialization here and the transmission of privileges in Section 1.4.

First, SQL elements such as schemas or modules have an owner. The owner of something has all privileges associated with that thing. There are three points at which ownership is established in SQL.

1. When a schema is created, it and all the tables and other schema elements in it are owned by the user who created it. This user thus has all possible privileges on elements of the schema.

2. When a session is initiated by a `CONNECT` statement, there is an opportunity to indicate the user with an `AUTHORIZATION` clause. For instance,

the connection statement

```
CONNECT TO Starfleet-sql-server AS conn1
    AUTHORIZATION kirk;
```

would create a connection called `conn1` to a database server whose name is `Starfleet-sql-server`, on behalf of user `kirk`. Presumably, the SQL implementation would verify that the user name is valid, for example by asking for a password. It is also possible to include the password in the `AUTHORIZATION` clause. That approach is somewhat insecure, since passwords are then visible to someone looking over Kirk's shoulder.

3. When a module is created, there is an option to give it an owner by using an `AUTHORIZATION` clause. For instance, a clause

```
AUTHORIZATION picard;
```

in a module-creation statement would make user `picard` the owner of the module. It is also acceptable to specify no owner for a module, in which case the module is publicly executable, but the privileges necessary for executing any operations in the module must come from some other source, such as the user associated with the connection and session during which the module is executed.

## 1.3 The Privilege-Checking Process

As we saw above, each module, schema, and session has an associated user; in SQL terms, there is an associated authorization ID for each. Any SQL operation has two parties:

1. The database elements upon which the operation is performed and
2. The agent that causes the operation.

The privileges available to the agent derive from a particular authorization ID called the *current authorization ID*. That ID is either

a) The module authorization ID, if the module that the agent is executing has an authorization ID, or

b) The session authorization ID if not.

We may execute the SQL operation only if the current authorization ID possesses all the privileges needed to carry out the operation on the database elements involved.

**Example 2:** To see the mechanics of checking privileges, let us reconsider Example 1. We might suppose that the referenced tables — `Movies` and `Studio` — are part of a schema called `MovieSchema`, which was created by and is owned by user `janeway`. At this point, user `janeway` has all privileges on these tables and any other elements of the schema `MovieSchema`. She may choose to grant some privileges to others by the mechanism to be described in Section 1.4, but let us assume none have been granted yet. There are several ways that the insertion of Example 1 can be executed.

1. The insertion could be executed as part of a module created by user `janeway` and containing an `AUTHORIZATION janeway` clause. The module authorization ID, if there is one, always becomes the current authorization ID. Then, the module and its SQL insertion statement have exactly the same privileges user `janeway` has, which includes all privileges on the tables `Movies` and `Studio`.

2. The insertion could be part of a module that has no owner. User `janeway` opens a connection with an `AUTHORIZATION janeway` clause in the `CONNECT` statement. Now, `janeway` is again the current authorization ID, so the insertion statement has all the privileges needed.

3. User `janeway` grants all privileges on tables `Movies` and `Studio` to user `archer`, or perhaps to the special user `PUBLIC`, which stands for "all users." Suppose the insertion statement is in a module with the clause

   ```
   AUTHORIZATION archer
   ```

   Since the current authorization ID is now `archer`, and this user has the needed privileges, the insertion is again permitted.

4. As in (3), suppose user `janeway` has given user `archer` the needed privileges. Also, suppose the insertion statement is in a module without an owner; it is executed in a session whose authorization ID was set by an `AUTHORIZATION archer` clause. The current authorization ID is thus `archer`, and that ID has the needed privileges.

□

There are several principles that are illustrated by Example 2. We shall summarize them below.

- The needed privileges are always available if the data is owned by the same user as the user whose ID is the current authorization ID. Scenarios (1) and (2) above illustrate this point.
- The needed privileges are available if the user whose ID is the current authorization ID has been granted those privileges by the owner of the data, or if the privileges have been granted to user `PUBLIC`. Scenarios (3) and (4) illustrate this point.

- Executing a module owned by the owner of the data, or by someone who has been granted privileges on the data, makes the needed privileges available. Of course, one needs the `EXECUTE` privilege on the module itself. Scenarios (1) and (3) illustrate this point.

- Executing a publicly available module during a session whose authorization ID is that of a user with the needed privileges is another way to execute the operation legally. Scenarios (2) and (4) illustrate this point.

## 1.4 Granting Privileges

So far, the only way we have seen to have privileges on a database element is to be the creator and owner of that element. SQL provides a `GRANT` statement to allow one user to give a privilege to another. The first user retains the privilege granted, as well; thus `GRANT` can be thought of as "copy a privilege."

There is one important difference between granting privileges and copying. Each privilege has an associated *grant option*. That is, one user may have a privilege like `SELECT` on table `Movies` "with grant option," while a second user may have the same privilege, but without the grant option. Then the first user may grant the privilege `SELECT` on `Movies` to a third user, and moreover that grant may be with or without the grant option. However, the second user, who does not have the grant option, may not grant the privilege `SELECT` on `Movies` to anyone else. If the third user got the privilege with the grant option, then that user may grant the privilege to a fourth user, again with or without the grant option, and so on.

A *grant statement* has the form:

`GRANT` <privilege list> `ON` <database element> `TO` <user list>

possibly followed by `WITH GRANT OPTION`.

The database element is typically a relation, either a base table or a view. If it is another kind of element, the name of the element is preceded by the type of that element, e.g., `ASSERTION`. The privilege list is a list of one or more privileges, e.g., `SELECT` or `INSERT(name)`. Optionally, the keywords `ALL PRIVILEGES` may appear here, as a shorthand for all the privileges that the grantor may legally grant on the database element in question.

In order to execute this grant statement legally, the user executing it must possess the privileges granted, and these privileges must be held with the grant option. However, the grantor may hold a more general privilege (with the grant option) than the privilege granted. For instance, the privilege `INSERT(name)` on table `Studio` might be granted, while the grantor holds the more general privilege `INSERT` on `Studio`, with grant option.

**Example 3:** User `janeway`, who is the owner of the `MovieSchema` schema that contains tables

```
Movies(title, year, length, genre, studioName, producerC#)
Studio(name, address, presC#)
```

grants the INSERT and SELECT privileges on table Studio and privilege SELECT on Movies to users kirk and picard. Moreover, she includes the grant option with these privileges. The grant statements are:

```
GRANT SELECT, INSERT ON Studio TO kirk, picard
    WITH GRANT OPTION;
GRANT SELECT ON Movies TO kirk, picard
    WITH GRANT OPTION;
```

Now, picard grants to user sisko the same privileges, but without the grant option. The statements executed by picard are:

```
GRANT SELECT, INSERT ON Studio TO sisko;
GRANT SELECT ON Movies TO sisko;
```

Also, kirk grants to sisko the minimal privileges needed for the insertion of Fig. 1, namely SELECT and INSERT(name) on Studio and SELECT on Movies. The statements are:

```
GRANT SELECT, INSERT(name) ON Studio TO sisko;
GRANT SELECT ON Movies TO sisko;
```

Note that sisko has received the SELECT privilege on Movies and Studio from two different users. He has also received the INSERT(name) privilege on Studio twice: directly from kirk and via the generalized privilege INSERT from picard. □

## 1.5 Grant Diagrams

Because of the complex web of grants and overlapping privileges that may result from a sequence of grants, it is useful to represent grants by a graph called a *grant diagram*. A SQL system maintains a representation of this diagram to keep track of both privileges and their origins (in case a privilege is revoked; see Section 1.6).

The nodes of a grant diagram correspond to a user and a privilege. Note that the ability to do something (e.g., SELECT on relation $R$) with the grant option and the same ability without the grant option are different privileges. These two different privileges, even if they belong to the same user, must be represented by two different nodes. Likewise, a user may hold two privileges, one of which is strictly more general than the other (e.g., SELECT on $R$ and SELECT on $R(A)$. These two privileges are also represented by two different nodes.

If user $U$ grants privilege $P$ to user $V$, and this grant was based on the fact that $U$ holds privilege $Q$ ($Q$ could be $P$ with the grant option, or it could be

some generalization of $P$, again with the grant option), then we draw an arc from the node for $U/Q$ to the node for $V/P$. As we shall see, privileges may be lost when arcs of this graph are deleted. That is why we use separate nodes for a pair of privileges, one of which includes the other, such as a privilege with and without the grant option. If the more powerful privilege is lost, the less powerful one might still be retained.

**Example 4:** Figure 2 shows the grant diagram that results from the sequence of grant statements of Example 3. We use the convention that a $*$ after a user-privilege combination indicates that the privilege includes the grant option. Also, $**$ after a user-privilege combination indicates that the privilege derives from ownership of the database element in question and was not due to a grant of the privilege from elsewhere. This distinction will prove important when we discuss revoking privileges in Section 1.6. A doubly starred privilege automatically includes the grant option. □

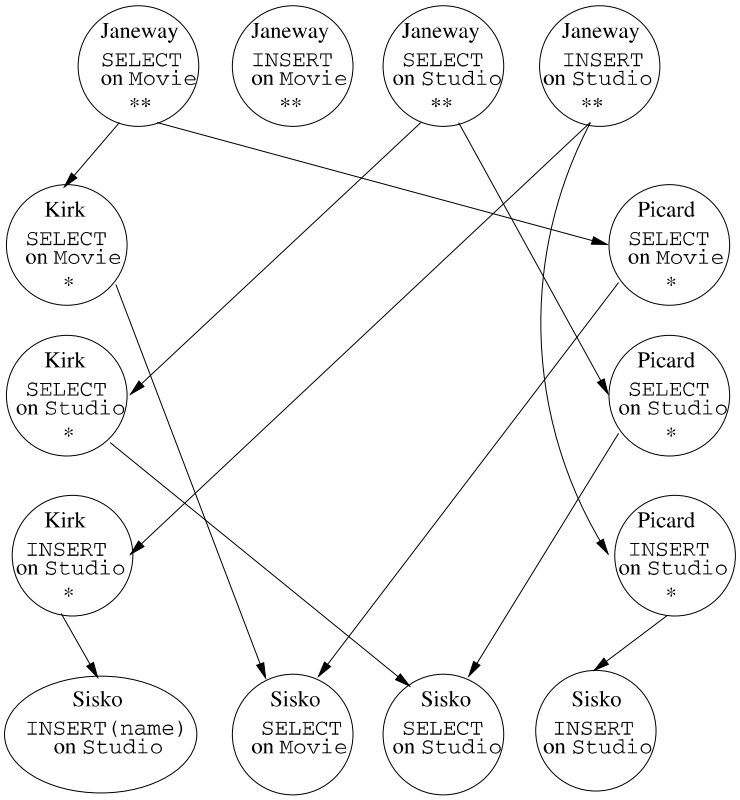



Figure 2: A grant diagram

## 1.6 Revoking Privileges

A granted privilege can be revoked at any time. The revoking of privileges may be required to *cascade*, in the sense that revoking a privilege with the grant option that has been passed on to other users may require those privileges to be revoked too. The simple form of a *revoke statement* begins:

`REVOKE` <privilege list> `ON` <database element> `FROM` <user list>

The statement ends with one of the following:

1. `CASCADE`. If chosen, then when the specified privileges are revoked, we also revoke any privileges that were granted *only* because of the revoked privileges. More precisely, if user $U$ has revoked privilege $P$ from user $V$, based on privilege $Q$ belonging to $U$, then we delete the arc in the grant diagram from $U/Q$ to $V/P$. Now, any node that is not accessible from some ownership node (doubly starred node) is also deleted.

2. `RESTRICT`. In this case, the revoke statement cannot be executed if the cascading rule described in the previous item would result in the revoking of any privileges due to the revoked privileges having been passed on to others.

It is permissible to replace `REVOKE` by `REVOKE GRANT OPTION FOR`, in which case the core privileges themselves remain, but the option to grant them to others is removed. We may have to modify a node, redirect arcs, or create a new node to reflect the changes for the affected users. This form of `REVOKE` also must be followed by either `CASCADE` or `RESTRICT`.

**Example 5:** Continuing with Example 3, suppose that `janeway` revokes the privileges she granted to `picard` with the statements:

```
REVOKE SELECT, INSERT ON Studio FROM picard CASCADE;
REVOKE SELECT ON Movies FROM picard CASCADE;
```

We delete the arcs of Fig. 2 from these `janeway` privileges to the corresponding `picard` privileges. Since `CASCADE` was stipulated, we also have to see if there are any privileges that are not reachable in the graph from a doubly starred (ownership-based) privilege. Examining Fig. 2, we see that `picard`'s privileges are no longer reachable from a doubly starred node (they might have been, had there been another path to a `picard` node). Also, `sisko`'s privilege to `INSERT` into `Studio` is no longer reachable. We thus delete not only `picard`'s privileges from the grant diagram, but we delete `sisko`'s `INSERT` privilege.

Note that we do not delete `sisko's SELECT` privileges on `Movies` and `Studio` or his `INSERT(name)` privilege on `Studio`, because these are all reachable from `janeway`'s ownership-based privileges via `kirk`'s privileges. The resulting grant diagram is shown in Fig. 3. □

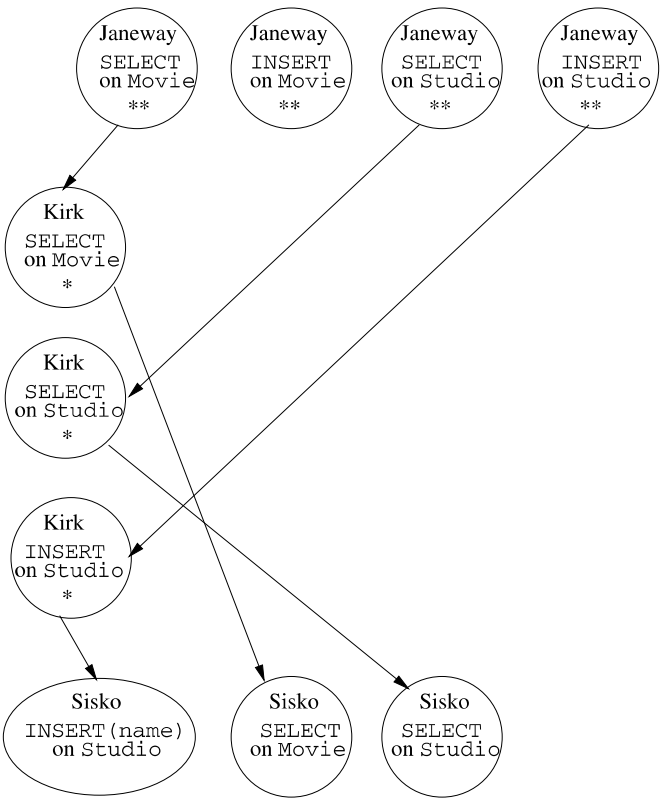


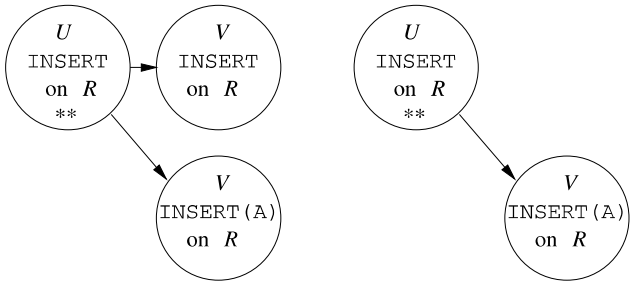

Figure 3: Grant diagram after revocation of `picard`'s privileges

**Example 6:** There are a few subtleties that we shall illustrate with abstract examples. First, when we revoke a general privilege $p$, we do not also revoke a privilege that is a special case of $p$. For instance, consider the following sequence of steps, whereby user $U$, the owner of relation $R$, grants the `INSERT` privilege on relation $R$ to user $V$, and also grants the `INSERT(A)` privilege on the same relation.

| Step | By | Action |
|---|---|---|
| 1 | $U$ | `GRANT INSERT ON` $R$ `TO` $V$ |
| 2 | $U$ | `GRANT INSERT(A) ON` $R$ `TO` $V$ |
| 3 | $U$ | `REVOKE INSERT ON` $R$ `FROM` $V$ `RESTRICT` |

When $U$ revokes `INSERT` from $V$, the `INSERT(A)` privilege remains. The grant diagrams after steps (2) and (3) are shown in Fig. 4.

Notice that after step (2) there are two separate nodes for the two similar but distinct privileges that user $V$ has. Also observe that the `RESTRICT` option in step (3) does not prevent the revocation, because $V$ had not granted the




(a) After step (2) (b) After step (3)

Figure 4: Revoking a general privilege leaves a more specific privilege

option to any other user. In fact, $V$ could not have granted either privilege, because $V$ obtained them without grant option. □

**Example 7:** Now, let us consider a similar example where $U$ grants $V$ a privilege $p*$ that includes the grant option and then revokes only the grant option. Assume the grant by $U$ was based on its privilege $q*$. In this case, we must replace the arc from the $U/q*$ node to $V/p*$ by an arc from $U/q*$ to $V/p$, i.e., the same privilege without the grant option. If there was no such node $V/p$, it must be created. In normal circumstances, the node $V/p*$ becomes unreachable, and any grants of $p$ made by $V$ will also be unreachable. However, it may be that $V$ was granted $p*$ by some other user besides $U$, in which case the $V/p*$ node remains accessible.

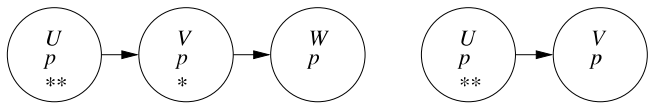
Here is a typical sequence of steps:

| Step | By | Action |
|---|---|---|
| 1 | $U$ | `GRANT` $p$ `TO` $V$ `WITH GRANT OPTION` |
| 2 | $V$ | `GRANT` $p$ `TO` $W$ |
| 3 | $U$ | `REVOKE GRANT OPTION FOR` $p$ `FROM` $V$ `CASCADE` |

In step (1), $U$ grants the privilege $p$ to $V$ with the grant option. In step (2), $V$ uses the grant option to grant $p$ to $W$. The diagram is then as shown in Fig. 5(a).

Then in step (3), $U$ revokes the grant option for privilege $p$ from $V$, but does not revoke the privilege itself. Since there is no node $V/p$, we create one. The arc from $U/p**$ to $V/P*$ is removed and replaced by one from $U/p**$ to $V/p$.

Now, the nodes $V/p*$ and $W/p$ are not reachable from any ** node. Thus, these nodes are deleted from the diagram. The resulting grant diagram is shown in Fig. 5(b). □




(a) After step (2) (b) After step (3)

Figure 5: Revoking a grant option leaves the underlying privilege

## 1.7 Exercises for Section 1

**Exercise 1.1:** Show the grant diagrams after steps (4) through (6) of the sequence of actions listed in Fig. 6. Assume $A$ is the owner of the relation to which privilege $p$ refers.

| Step | By | Action |
|---|---|---|
| 1 | $A$ | `GRANT` $p$ `TO` $B$ `WITH GRANT OPTION` |
| 2 | $A$ | `GRANT` $p$ `TO` $C$ |
| 3 | $B$ | `GRANT` $p$ `TO` $D$ `WITH GRANT OPTION` |
| 4 | $D$ | `GRANT` $p$ `TO` $B$, $C$, $E$ `WITH GRANT OPTION` |
| 5 | $B$ | `REVOKE` $p$ `FROM` $D$ `CASCADE` |
| 6 | $A$ | `REVOKE` $p$ `FROM` $C$ `CASCADE` |

Figure 6: Sequence of actions for Exercise 1.1

**Exercise 1.2:** Show the grant diagrams after steps (5) and (6) of the sequence of actions listed in Fig. 7. Assume $A$ is the owner of the relation to which privilege $p$ refers.

| Step | By | Action |
|---|---|---|
| 1 | $A$ | GRANT $p$ TO $B$, $E$ WITH GRANT OPTION |
| 2 | $B$ | GRANT $p$ TO $C$ WITH GRANT OPTION |
| 3 | $C$ | GRANT $p$ TO $D$ WITH GRANT OPTION |
| 4 | $E$ | GRANT $p$ TO $C$ |
| 5 | $E$ | GRANT $p$ TO $D$ WITH GRANT OPTION |
| 6 | $A$ | REVOKE GRANT OPTION FOR $p$ FROM $B$ CASCADE |

Figure 7: Sequence of actions for Exercise 1.2

**! Exercise 1.3:** Show the final grant diagram after the following steps, assuming $A$ is the owner of the relation to which privilege $p$ refers.

| Step | By | Action |
|---|---|---|
| 1 | $A$ | GRANT $p$ TO $B$ WITH GRANT OPTION |
| 2 | $B$ | GRANT $p$ TO $B$ WITH GRANT OPTION |
| 3 | $A$ | REVOKE $p$ FROM $B$ CASCADE |

# 2 Recursion in SQL

The SQL-99 standard includes provision for recursive definitions of queries. Although this feature is not part of the "core" SQL-99 standard that every DBMS is expected to implement, at least one major system — IBM's DB2 — does implement the SQL-99 proposal, which we describe in this section.

## 2.1 Defining Recursive Relations in SQL

The WITH statement in SQL allows us to define temporary relations, recursive or not. To define a recursive relation, the relation can be used within the WITH statement itself. A simple form of the WITH statement is:

WITH $R$ AS <definition of $R$> <query involving $R$>

That is, one defines a temporary relation named $R$, and then uses $R$ in some query. The temporary relation is not available outside the query that is part of the WITH statement.

More generally, one can define several relations after the WITH, separating their definitions by commas. Any of these definitions may be recursive. Several defined relations may be mutually recursive; that is, each may be defined in terms of some of the other relations, optionally including itself. However, any relation that is involved in a recursion must be preceded by the keyword RECURSIVE. Thus, a more general form of WITH statement is shown in Fig. 8.

**Example 8:** Many examples of the use of recursion can be found in a study of paths in a graph. Figure 9 shows a graph representing some flights of two

```
WITH
   [RECURSIVE] R1 AS <definition of R1>,
   [RECURSIVE] R2 AS <definition of R2>,
     ...
   [RECURSIVE] Rn AS <definition of Rn>
<query involving R1, R2, ..., Rn >
```

Figure 8: Form of a `WITH` statement defining several temporary relations

hypothetical airlines — *Untried Airlines* (UA), and *Arcane Airlines* (AA) — among the cities San Francisco, Denver, Dallas, Chicago, and New York. The data of the graph can be represented by a relation

```
Flights(airline, frm, to, departs, arrives)
```

and the particular tuples in this table are shown in Fig. 9.

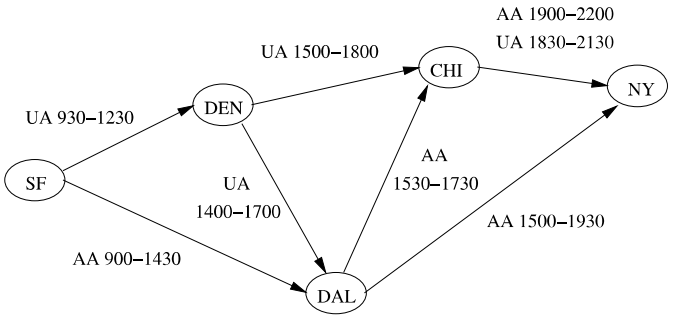



Figure 9: A map of some airline flights

| *airline* | *from* | *to* | *departs* | *arrives* |
|---|---|---|---|---|
| UA | SF | DEN | 930 | 1230 |
| AA | SF | DAL | 900 | 1430 |
| UA | DEN | CHI | 1500 | 1800 |
| UA | DEN | DAL | 1400 | 1700 |
| AA | DAL | CHI | 1530 | 1730 |
| AA | DAL | NY | 1500 | 1930 |
| AA | CHI | NY | 1900 | 2200 |
| UA | CHI | NY | 1830 | 2130 |

Figure 10: Tuples in the relation `Flights`

The simplest recursive question we can ask is "For what pairs of cities $(x, y)$ is it possible to get from city $x$ to city $y$ by taking one or more flights?" Before

writing this query in recursive SQL, it is useful to express the recursion in Datalog notation. Since many concepts involving recursion are easier to express in Datalog than in SQL, you may wish to review the terminology of that section before proceeding. The following two Datalog rules describe a relation `Reaches(x,y)` that contains exactly these pairs of cities.

1. `Reaches(x,y)` ← `Flights(a,x,y,d,r)`
2. `Reaches(x,y)` ← `Reaches(x,z) AND Reaches(z,y)`

The first rule says that `Reaches` contains those pairs of cities for which there is a direct flight from the first to the second; the airline $a$, departure time $d$, and arrival time $r$ are arbitrary in this rule. The second rule says that if you can reach from city $x$ to city $z$ and you can reach from $z$ to city $y$, then you can reach from $x$ to $y$.

Evaluating a recursive relation requires that we apply the Datalog rules repeatedly, starting by assuming there are no tuples in `Reaches`. We begin by using Rule (1) to get the following pairs in `Reaches`: (`SF`, `DEN`), (`SF`, `DAL`), (`DEN`, `CHI`), (`DEN`, `DAL`), (`DAL`, `CHI`), (`DAL`, `NY`), and (`CHI`, `NY`). These are the seven pairs represented by arcs in Fig. 9.

In the next round, we apply the recursive Rule (2) to put together pairs of arcs such that the head of one is the tail of the next. That gives us the additional pairs (`SF`, `CHI`), (`DEN`, `NY`), and (`SF`, `NY`). The third round combines all one- and two-arc pairs together to form paths of length up to four arcs. In this particular diagram, we get no new pairs. The relation `Reaches` thus consists of the ten pairs $(x, y)$ such that $y$ is reachable from $x$ in the diagram of Fig. 9. Because of the way we drew the diagram, these pairs happen to be exactly those $(x, y)$ such that $y$ is to the right of $x$ in Fig. 9.

From the two Datalog rules for `Reaches` in Example 8, we can develop a SQL query that produces the relation `Reaches`. This SQL query places the Datalog rules for `Reaches` in a `WITH` statement, and follows it by a query. In our example, the desired result was the entire `Reaches` relation, but we could also ask some query about `Reaches`, for instance the set of cities reachable from Denver.

```
1)  WITH RECURSIVE Reaches(frm, to) AS
2)          (SELECT frm, to FROM Flights)
3)      UNION
4)          (SELECT R1.frm, R2.to
5)           FROM Reaches R1, Reaches R2
6)           WHERE R1.to = R2.frm)
7)  SELECT * FROM Reaches;
```

Figure 11: Recursive SQL query for pairs of reachable cities

Figure 11 shows how to express `Reaches` as a SQL query. Line (1) introduces the definition of `Reaches`, while the actual definition of this relation is in

### Mutual Recursion

There is a graph-theoretic way to check whether two relations or predicates are mutually recursive. Construct a *dependency graph* whose nodes correspond to the relations (or predicates if we are using Datalog rules). Draw an arc from relation $A$ to relation $B$ if the definition of $B$ depends directly on the definition of $A$. That is, if Datalog is being used, then $A$ appears in the body of a rule with $B$ at the head. In SQL, $A$ would appear in a `FROM` clause, somewhere in the definition of $B$, possibly in a subquery. If there is a cycle involving nodes $R$ and $S$, then $R$ and $S$ are *mutually recursive*. The most common case will be a loop from $R$ to $R$, indicating that $R$ depends recursively upon itself.

lines (2) through (6).

That definition is a union of two queries, corresponding to the two Datalog rules by which `Reaches` was defined. Line (2) is the first term of the union and corresponds to the first, or basis rule. It says that for every tuple in the `Flights` relation, the second and third components (the `frm` and `to` components) are a tuple in `Reaches`.

Lines (4) through (6) correspond to Rule (2), the recursive rule, in the definition of `Reaches`. The two `Reaches` subgoals in Rule (2) are represented in the `FROM` clause by two aliases `R1` and `R2` for `Reaches`. The first component of `R1` corresponds to $x$ in Rule (2), and the second component of `R2` corresponds to $y$. Variable $z$ is represented by both the second component of `R1` and the first component of `R2`; note that these components are equated in line (6).

Finally, line (7) describes the relation produced by the entire query. It is a copy of the `Reaches` relation. As an alternative, we could replace line (7) by a more complex query. For instance,

```
7)  SELECT to FROM Reaches WHERE frm = 'DEN';
```

would produce all those cities reachable from Denver. □

## 2.2 Problematic Expressions in Recursive SQL

The SQL standard for recursion does not allow an arbitrary collection of mutually recursive relations to be written in a `WITH` clause. There is a small matter that the standard requires only that *linear* recursion be supported. A linear recursion, in Datalog terms, is one in which no rule has more than one subgoal that is mutually recursive with the head. Notice that Rule (2) in Example 8 has two subgoals with predicate `Reaches` that are mutually recursive with the head (a predicate is always mutually recursive with itself; see the box on Mutual

Recursion). Thus, technically, a DBMS might refuse to execute Fig. 11 and yet conform to the standard.[1]

But there is a more important restriction on SQL recursions, one that, if violated leads to recursions that cannot be executed by the query processor in any meaningful way. To be a legal SQL recursion, the definition of a recursive relation $R$ may involve only the use of a mutually recursive relation $S$ (including $R$ itself) if that use is "monotone" in $S$. A use of $S$ is *monotone* if adding an arbitrary tuple to $S$ might add one or more tuples to $R$, or it might leave $R$ unchanged, but it can never cause any tuple to be deleted from $R$. The following example suggests what can happen if the monotonicity requirement is not respected.

**Example 9:** Suppose relation $R$ is a unary (one-attribute) relation, and its only tuple is (0). $R$ is used as an EDB relation in the following Datalog rules:

1. `P(x)` ← `R(x) AND NOT Q(x)`
2. `Q(x)` ← `R(x) AND NOT P(x)`

Informally, the two rules tell us that an element $x$ in $R$ is either in $P$ or in $Q$ but not both. Notice that $P$ and $Q$ are mutually recursive.

If we start out, assuming that both $P$ and $Q$ are empty, and apply the rules once, we find that $P = \{(0)\}$ and $Q = \{(0)\}$; that is, (0) is in both IDB relations. On the next round, we apply the rules to the new values for $P$ and $Q$ again, and we find that now both are empty. This cycle repeats as long as we like, but we never converge to a solution.

In fact, there are two "solutions" to the Datalog rules:

| | | |
|---|---|---|
| a) | $P = \{(0)\}$ | $Q = \emptyset$ |
| b) | $P = \emptyset$ | $Q = \{(0)\}$ |

However, there is no reason to assume one over the other, and the simple iteration we suggested as a way to compute recursive relations never converges to either. Thus, we cannot answer a simple question such as "Is $P(0)$ true?"

The problem is not restricted to Datalog. The two Datalog rules of this example can be expressed in recursive SQL. Figure 12 shows one way of doing so. This SQL does not adhere to the standard, and no DBMS should execute it. □

The problem in Example 9 is that the definitions of $P$ and $Q$ in Fig. 12 are not monotone. Look at the definition of $P$ in lines (2) through (5) for instance. $P$ depends on $Q$, with which it is mutually recursive, but adding a tuple to $Q$ can delete a tuple from $P$. Notice that if $R = \{(0)\}$ and $Q$ is empty, then $P = \{(0)\}$. But if we add (0) to $Q$, then we delete (0) from $P$. Thus, the definition of $P$ is not monotone in $Q$, and the SQL code of Fig. 12 does not meet the standard.

[1]Note, however, that we can replace either one of the uses of `Reaches` in line (5) of Fig. 11 by `Flights`, and thus make the recursion linear. Nonlinear recursions can frequently — although not always — be made linear in this fashion.

```
 1)  WITH
 2)      RECURSIVE P(x) AS
 3)              (SELECT * FROM R)
 4)          EXCEPT
 5)              (SELECT * FROM Q),

 6)      RECURSIVE Q(x) AS
 7)              (SELECT * FROM R)
 8)          EXCEPT
 9)              (SELECT * FROM P)

10)  SELECT * FROM P;
```

Figure 12: Query with nonmonotonic behavior, illegal in SQL

**Example 10:** Aggregation can also lead to nonmonotonicity. Suppose we have unary (one-attribute) relations $P$ and $Q$ defined by the following two conditions:

1. $P$ is the union of $Q$ and an EDB relation $R$.
2. $Q$ has one tuple that is the sum of the members of $P$.

We can express these conditions by a `WITH` statement, although this statement violates the monotonicity requirement of SQL. The query shown in Fig. 13 asks for the value of $P$.

```
 1)  WITH
 2)      RECURSIVE P(x) AS
 3)              (SELECT * FROM R)
 4)          UNION
 5)              (SELECT * FROM Q),

 6)      RECURSIVE Q(x) AS
 7)          SELECT SUM(x) FROM P

 8)  SELECT * FROM P;
```

Figure 13: Nonmonotone query involving aggregation, illegal in SQL

Suppose that $R$ consists of the tuples (12) and (34), and initially $P$ and $Q$ are both empty. Figure 14 summarizes the values computed in the first six rounds. Note that both relations are computed, in one round, from the values of the relations at the previous round. Thus, $P$ is computed in the first round

| Round | $P$ | $Q$ |
|---|---|---|
| 1) | $\{(12), (34)\}$ | $\{\texttt{NULL}\}$ |
| 2) | $\{(12), (34), \texttt{NULL}\}$ | $\{(46)\}$ |
| 3) | $\{(12), (34), (46)\}$ | $\{(46)\}$ |
| 4) | $\{(12), (34), (46)\}$ | $\{(92)\}$ |
| 5) | $\{(12), (34), (92)\}$ | $\{(92)\}$ |
| 6) | $\{(12), (34), (92)\}$ | $\{(138)\}$ |

Figure 14: Iterative calculation for a nonmonotone aggregation

to be the same as $R$, and $Q$ is $\{\texttt{NULL}\}$, since the old, empty value of $P$ is used in line (7).

At the second round, the union of lines (3) through (5) is the set

$$R \cup \{\texttt{NULL}\} = \{(12),\ (34),\ \texttt{NULL}\}$$

so that set becomes the new value of $P$. The old value of $P$ was $\{(12),\ (34)\}$, so on the second round $Q = \{(46)\}$. That is, 46 is the sum of 12 and 34.

At the third round, we get $P = \{(12), (34), (46)\}$ at lines (2) through (5). Using the old value of $P$, $\{(12), (34), \texttt{NULL}\}$, $Q$ is defined by lines (6) and (7) to be $\{(46)\}$ again. Remember that `NULL` is ignored in a sum.

At the fourth round, $P$ has the same value, $\{(12), (34), (46)\}$, but $Q$ gets the value $\{(92)\}$, since 12+34+46=92. Notice that $Q$ has lost the tuple (46), although it gained the tuple (92). That is, adding the tuple (46) to $P$ has caused a tuple (by coincidence the same tuple) to be deleted from $Q$. That behavior is the nonmonotonicity that SQL prohibits in recursive definitions, confirming that the query of Fig. 13 is illegal. In general, at the $2i$th round, $P$ will consist of the tuples (12), (34), and $(46i - 46)$, while $Q$ consists only of the tuple $(46i)$. □

## 2.3 Exercises for Section 2

**Exercise 2.1:** The relation

```
Flights(airline, frm, to, departs, arrives)
```

from Example 8 has arrival- and departure-time information that we did not consider. Suppose we are interested not only in whether it is possible to reach one city from another, but whether the journey has reasonable connections. That is, when using more than one flight, each flight must arrive at least an hour before the next flight departs. You may assume that no journey takes place over more than one day, so it is not necessary to worry about arrival close to midnight followed by a departure early in the morning.

a) Write this recursion in Datalog.

b) Write the recursion in SQL.

**! Exercise 2.2:** In Example 8 we used `frm` as an attribute name. Why did we not use the more obvious name `from`?

**Exercise 2.3:** Suppose we have a relation

```
SequelOf(movie, sequel)
```

that gives the immediate sequels of a movie, of which there can be more than one. We want to define a recursive relation `FollowOn` whose pairs $(x, y)$ are movies such that $y$ was either a sequel of $x$, a sequel of a sequel, or so on.

a) Write the definition of `FollowOn` as recursive Datalog rules.

b) Write the definition of `FollowOn` as a SQL recursion.

c) Write a recursive SQL query that returns the set of pairs $(x, y)$ such that movie $y$ is a follow-on to movie $x$, but is not a sequel of $x$.

d) Write a recursive SQL query that returns the set of pairs $(x, y)$ meaning that $y$ is a follow-on of $x$, but is neither a sequel nor a sequel of a sequel.

**!** e) Write a recursive SQL query that returns the set of movies $x$ that have at least two follow-ons. Note that both could be sequels, rather than one being a sequel and the other a sequel of a sequel.

**!** f) Write a recursive SQL query that returns the set of pairs $(x, y)$ such that movie $y$ is a follow-on of $x$ but $y$ has at most one follow-on.

**Exercise 2.4:** Suppose we have a relation

```
Rel(class, rclass, mult)
```

that describes how one ODL class is related to other classes. Specifically, this relation has tuple $(c, d, m)$ if there is a relation from class $c$ to class $d$. This relation is multivalued if $m =$ `'multi'` and it is single-valued if $m =$ `'single'`. It is possible to view `Rel` as defining a graph whose nodes are classes and in which there is an arc from $c$ to $d$ labeled $m$ if and only if $(c, d, m)$ is a tuple of `Rel`. Write a recursive SQL query that produces the set of pairs $(c, d)$ such that:

a) There is a path from class $c$ to class $d$ in the graph described above.

b) There is a path from $c$ to $d$ along which every arc is labeled `single`.

**!** c) There is a path from $c$ to $d$ along which at least one arc is labeled `multi`.

d) There is a path from $c$ to $d$ but no path along which all arcs are labeled `single`.

! e) There is a path from $c$ to $d$ along which arc labels alternate `single` and `multi`.

f) There are paths from $c$ to $d$ and from $d$ to $c$ along which every arc is labeled `single`.

# 3 The Object-Relational Model

The relational model and the object-oriented model typified by ODL are two important points in a spectrum of options that could underlie a DBMS. For an extended period, the relational model was dominant in the commercial DBMS world. Object-oriented DBMS's made limited inroads during the 1990's, but never succeeded in winning significant market share from the vendors of relational DBMS's. Rather, the vendors of relational systems have moved to incorporate many of the ideas found in ODL or other object-oriented-database proposals. As a result, many DBMS products that used to be called "relational" are now called "object-relational."

This section extends the abstract relational model to incorporate several important object-relational ideas. It is followed by sections that cover object-relational extensions of SQL. We introduce the concept of object-relations in Section 3.1, then discuss one of its earliest embodiments — nested relations — in Section 3.2. ODL-like references for object-relations are discussed in Section 3.3, and in Section 3.4 we compare the object-relational model with the pure object-oriented approach.

## 3.1 From Relations to Object-Relations

While the relation remains the fundamental concept, the relational model has been extended to the *object-relational model* by incorporation of features such as:

1. *Structured types for attributes.* Instead of allowing only atomic types for attributes, object-relational systems support a type system like ODL's: types built from atomic types and type constructors for structs, sets, and bags, for instance. Especially important is a type that is a bag of structs, which is essentially a relation. That is, a value of one component of a tuple can be an entire relation, called a "nested relation."

2. *Methods.* These are similar to methods in ODL or any object-oriented programming system.

3. *Identifiers for tuples.* In object-relational systems, tuples play the role of objects. It therefore becomes useful in some situations for each tuple to have a unique ID that distinguishes it from other tuples, even from tuples that have the same values in all components. This ID, like the object-identifier assumed in ODL, is generally invisible to the user, although

there are even some circumstances where users can see the identifier for a tuple in an object-relational system.

4. *References.* While the pure relational model has no notion of references or pointers to tuples, object-relational systems can use these references in various ways.

In the next sections, we shall elaborate upon and illustrate each of these additional capabilities of object-relational systems.

## 3.2 Nested Relations

In the *nested-relational model*, we allow attributes of relations to have a type that is not atomic; in particular, a type can be a relation schema. As a result, there is a convenient, recursive definition of the types of attributes and the types (schemas) of relations:

**BASIS**: An atomic type (integer, real, string, etc.) can be the type of an attribute.

**INDUCTION**: A relation's type can be any *schema* consisting of names for one or more attributes, and any legal type for each attribute. In addition, a schema also can be the type of any attribute.

In what follows, we shall generally omit atomic types where they do not matter. An attribute that is a schema will be represented by the attribute name and a parenthesized list of the attributes of its schema. Since those attributes may themselves have structure, parentheses can be nested to any depth.

**Example 11:** Let us design a nested-relation schema for stars that incorporates within the relation an attribute `movies`, which will be a relation representing all the movies in which the star has appeared. The relation schema for attribute `movies` will include the title, year, and length of the movie. The relation schema for the relation `Stars` will include the name, address, and birthdate, as well as the information found in `movies`. Additionally, the `address` attribute will have a relation type with attributes `street` and `city`. We can record in this relation several addresses for the star. The schema for `Stars` can be written:

```
Stars(name, address(street, city), birthdate,
   movies(title, year, length))
```

An example of a possible relation for nested relation `Stars` is shown in Fig. 15. We see in this relation two tuples, one for Carrie Fisher and one for Mark Hamill. The values of components are abbreviated to conserve space, and the dashed lines separating tuples are only for convenience and have no notational significance.

| *name* | *address* | *birthdate* | *movies* |
|---|---|---|---|
| Fisher | *street* \| *city*<br>Maple \| H'wood<br>Locust \| Malibu | 9/9/99 | *title* \| *year* \| *length*<br>Star Wars \| 1977 \| 124<br>Empire \| 1980 \| 127<br>Return \| 1983 \| 133 |
| Hamill | *street* \| *city*<br>Oak \| B'wood | 8/8/88 | *title* \| *year* \| *length*<br>Star Wars \| 1977 \| 124<br>Empire \| 1980 \| 127<br>Return \| 1983 \| 133 |

Figure 15: A nested relation for stars and their movies

In the Carrie Fisher tuple, we see her name, an atomic value, followed by a relation for the value of the address component. That relation has two attributes, `street` and `city`, and there are two tuples, corresponding to her two houses. Next comes the birthdate, another atomic value. Finally, there is a component for the `movies` attribute; this attribute has a relation schema as its type, with components for the title, year, and length of a movie. The relation for the `movies` component of the Carrie Fisher tuple has tuples for her three best-known movies.

The second tuple, for Mark Hamill, has the same components. His relation for `address` has only one tuple, because in our imaginary data, he has only one house. His relation for `movies` looks just like Carrie Fisher's because their best-known movies happen, by coincidence, to be the same. Note that these two relations are two different tuple-components. These components happen to be identical, just like two components that happened to have the same integer value, e.g., 124. □

## 3.3 References

The fact that movies like *Star Wars* will appear in several relations that are values of the `movies` attribute in the nested relation `Stars` is a cause of redundancy. In effect, the schema of Example 11 has the nested-relation analog of not being in BCNF. However, decomposing this `Stars` relation will not eliminate the redundancy. Rather, we need to arrange that among all the tuples of all the `movies` relations, a movie appears only once.

To cure the problem, object-relations need the ability for one tuple $t$ to refer to another tuple $s$, rather than incorporating $s$ directly in $t$. We thus add to our model an additional inductive rule: the type of an attribute also can be a

reference to a tuple with a given schema or a set of references to tuples with a given schema.

If an attribute $A$ has a type that is a reference to a single tuple with a relation schema named $R$, we show the attribute $A$ in a schema as $A(*R)$. Notice that this situation is analogous to an ODL relationship $A$ whose type is $R$; i.e., it connects to a single object of type $R$. Similarly, if an attribute $A$ has a type that is a set of references to tuples of schema $R$, then $A$ will be shown in a schema as $A(\{*R\})$. This situation resembles an ODL relationship $A$ that has type `Set<`$R$`>`.

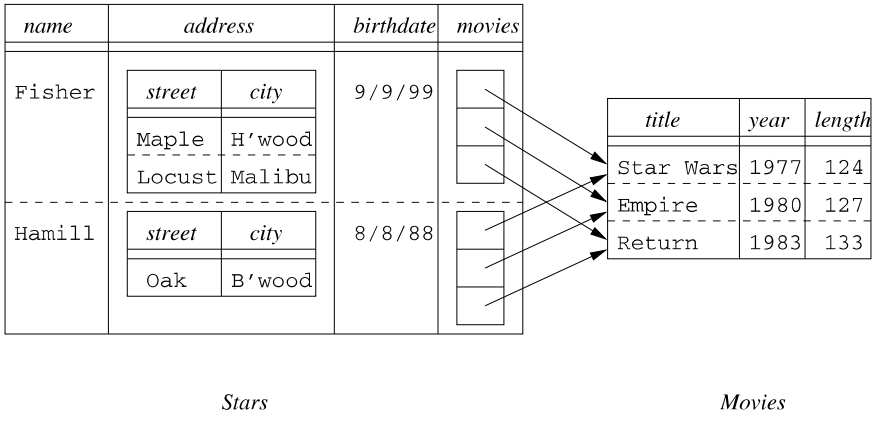



Figure 16: Sets of references as the value of an attribute

**Example 12:** An appropriate way to fix the redundancy in Fig. 15 is to use two relations, one for stars and one for movies. In this example only, we shall use a relation called `Movies` that is an ordinary relation with the same schema as the attribute `movies` in Example 11. A new relation `Stars` has a schema similar to the nested relation `Stars` of that example, but the movies attribute will have a type that is a set of references to `Movies` tuples. The schemas of the two relations are thus:

```
Movies(title, year, length)
Stars(name, address(street, city), birthdate,
    movies({*Movies}))
```

The data of Fig. 15, converted to this new schema, is shown in Fig. 16. Notice that, because each movie has only one tuple, although it can have many references, we have eliminated the redundancy inherent in the schema of Example 11. □

## 3.4 Object-Oriented Versus Object-Relational

The object-oriented data model, as typified by ODL, and the object-relational model discussed here, are remarkably similar. Some of the salient points of comparison follow.

### Objects and Tuples

An object's value is really a struct with components for its attributes and relationships. It is not specified in the ODL standard how relationships are to be represented, but we may assume that an object is connected to related objects by some collection of references. A tuple is likewise a struct, but in the conventional relational model, it has components for only the attributes. Relationships would be represented by tuples in another relation. However the object-relational model, by allowing sets of references to be a component of tuples, also allows relationships to be incorporated directly into the tuples that represent an "object" or entity.

### Methods

We did not discuss the use of methods as part of an object-relational schema. However, in practice, the SQL-99 standard and all implementations of object-relational ideas allow the same ability as ODL to declare and define methods associated with any class or type.

### Type Systems

The type systems of the object-oriented and object-relational models are quite similar. Each is based on atomic types and construction of new types by struct- and collection-type-constructors. The choice of collection types may vary, but all variants include at least sets and bags. Moreover, the set (or bag) of structs type plays a special role in both models. It is the type of classes in ODL, and the type of relations in the object-relational model.

### References and Object-ID's

A pure object-oriented model uses object-ID's that are completely hidden from the user, and thus cannot be seen or queried. The object-relational model allows references to be part of a type, and thus it is possible under some circumstances for the user to see their values and even remember them for future use. You may regard this situation as anything from a serious bug to a stroke of genius, depending on your point of view, but in practice it appears to make little difference.

### Backwards Compatibility

With little difference in essential features of the two models, it is interesting to consider why object-relational systems have dominated the pure object-oriented systems in the marketplace. The reason, we believe, is as follows. As relational DBMS's evolved into object-relational DBMS's, the vendors were careful to maintain backwards compatibility. That is, newer versions of the system would still run the old code and accept the same schemas, should the user not care to adopt any of the object-oriented features. On the other hand, migration to a pure object-oriented DBMS would require the installations to rewrite and reorganize extensively. Thus, whatever competitive advantage could be argued for object-oriented database systems was insufficient to motivate many to make the switch.

## 3.5 Exercises for Section 3

**Exercise 3.1:** Using the notation developed for nested relations and relations with references, give one or more relation schemas that represent the following information. In each case, you may exercise some discretion regarding what attributes of a relation are included, but try to keep close to the attributes found in our running movie example. Also, indicate whether your schemas exhibit redundancy, and if so, what could be done to avoid it.

a) Movies, with the usual attributes plus all their stars and the usual information about the stars.

! b) Studios, all the movies made by that studio, and all the stars of each movie, including all the usual attributes of studios, movies, and stars.

c) Movies with their studio, their stars, and all the usual attributes of these.

# 4 User-Defined Types in SQL

We now turn to the way SQL-99 incorporates many of the object-oriented features that we saw in Section 3. The central extension that turns the relational model into the object-relational model in SQL is the *user-defined type*, or UDT. We find UDT's used in two distinct ways:

1. A UDT can be the type of a table.
2. A UDT can be the type of an attribute belonging to some table.

## 4.1 Defining Types in SQL

The SQL-99 standard allows the programmer to define UDT's in several ways. The simplest is as a renaming of an existing type.

`CREATE TYPE` $T$ `AS` <primitive type>`;`

renames a primitive type such as `INTEGER`. Its purpose is to prevent errors caused by accidental coercions among values that logically should not be compared or interchanged, even though they have the same primitive data type. An example should make the purpose clear.

**Example 13:** In our running movies example, there are several attributes of type `INTEGER`. These include `length` of `Movies`, `cert#` of `MovieExec`, and `presC#` of `Studio`. It makes sense to compare a value of `cert#` with a value of `presC#`, and we could even take a value from one of these two attributes and store it in a tuple as the value of the other attribute. However, It would not make sense to compare a movie length with the certificate number of a movie executive, or to take a `length` value from a `Movies` tuple and store it in the `cert#` attribute of a `MovieExec` tuple.

If we create types:

```
CREATE TYPE CertType AS INTEGER;
CREATE TYPE LengthType AS INTEGER;
```

then we can declare `cert#` and `presC#` to be of type `CertType` instead of `INTEGER` in their respective relation declarations, and we can declare `length` to be of type `LengthType` in the `Movies` declaration. In that case, an object-relational DBMS will intercept attempts to compare values of one type with the other, or to use a value of one type in place of the other. □

A more powerful form of UDT declaration in SQL is similar to a class declaration in ODL, with some distinctions. First, key declarations for a relation with a user-defined type are part of the table definition, not the type definition; that is, many SQL relations can be declared to have the same UDT but different keys and other constraints. Second, in SQL we do not treat relationships as properties. A relationship can be represented by a separate relation

or through references, which are covered in Section 4.5. This form of UDT definition is:

`CREATE TYPE` $T$ `AS (`<attribute declarations>`);`

**Example 14:** Figure 17 shows two UDT's, `AddressType` and `StarType`. A tuple of type `AddressType` has two components, whose attributes are `street` and `city`. The types of these components are character strings of length 50 and 20, respectively. A tuple of type `StarType` also has two components. The first is attribute `name`, whose type is a 30-character string, and the second is `address`, whose type is itself a UDT `AddressType`, that is, a tuple with `street` and `city` components. □

```
CREATE TYPE AddressType AS (
    street  CHAR(50),
    city    CHAR(20)
);

CREATE TYPE StarType AS (
    name    CHAR(30),
    address AddressType
);
```

Figure 17: Two type definitions

## 4.2 Method Declarations in UDT's

The declaration of a method resembles the way a function in PSM is introduced. There is no analog of PSM procedures as methods. That is, every method returns a value of some type. While function declarations and definitions in PSM are combined, a method needs both a declaration, which follows the parenthesized list of attributes in the `CREATE TYPE` statement, and a separate definition, in a `CREATE METHOD` statement. The actual code for the method need not be PSM, although it could be. For example, the method body could be Java with JDBC used to access the database.

A method declaration looks like a PSM function declaration, with the keyword `METHOD` replacing `CREATE FUNCTION`. However, SQL methods typically have no arguments; they are applied to rows, just as ODL methods are applied to objects. In the definition of the method, `SELF` refers to this tuple, if necessary.

**Example 15:** Let us extend the definition of the type `AddressType` of Fig. 17 with a method `houseNumber` that extracts from the `street` component the portion devoted to the house address. For instance, if the `street` component

were '123 Maple St.', then houseNumber should return '123'. Exactly how houseNumber works is not visible in its declaration; the details are left for the definition. The revised type definition is thus shown in Fig. 18.

```
CREATE TYPE AddressType AS (
    street  CHAR(50),
    city    CHAR(20)
    )
    METHOD houseNumber() RETURNS CHAR(10);
```

Figure 18: Adding a method declaration to a UDT

We see the keyword METHOD, followed by the name of the method and a parenthesized list of its arguments and their types. In this case, there are no arguments, but the parentheses are still needed. Had there been arguments, they would have appeared, followed by their types, such as (a INT, b CHAR(5)). □

## 4.3 Method Definitions

Separately, we need to define the method. A simple form of method definition is:

CREATE METHOD <method name, arguments, and return type>
FOR <UDT name>
<method body>

That is, the UDT for which the method is defined is indicated in a FOR clause. The method definition need not be contiguous to, or part of, the definition of the type to which it belongs.

**Example 16:** For instance, we could define the method houseNumber from Example 15 as in Fig. 19. We have omitted the body of the method because accomplishing the intended separation of the string string as intended is non-trivial, even if a general-purpose host language is used. □

```
CREATE METHOD houseNumber() RETURNS CHAR(10)
FOR AddressType
BEGIN
    ...
END;
```

Figure 19: Defining a method

## 4.4 Declaring Relations with a UDT

Having declared a type, we may declare one or more relations whose tuples are of that type. In the form of relation declarations, the attribute declarations are omitted from the parenthesized list of elements, and replaced by a clause with `OF` and the name of the UDT. That is, the alternative form of a `CREATE TABLE` statement, using a UDT, is:

`CREATE TABLE` <table name> `OF` <UDT name>
`(`<list of elements>`);`

The parenthesized list of elements can include keys, foreign keys, and tuple-based constraints. Note that all these elements are declared for a particular table, not for the UDT. Thus, there can be several tables with the same UDT as their row type, and these tables can have different constraints, and even different keys. If there are no constraints or key declarations desired for the table, then the parentheses are not needed.

**Example 17:** We could declare `MovieStar` to be a relation whose tuples are of type `StarType` by

```
CREATE TABLE MovieStar OF StarType (
    PRIMARY KEY (name)
);
```

As a result, table `MovieStar` has two attributes, `name` and `address`. The first attribute, `name`, is an ordinary character string, but the second, `address`, has a type that is itself a UDT, namely the type `AddressType`. Attribute `name` is a key for this relation, so it is not possible to have two tuples with the same name. □

## 4.5 References

The effect of object identity in object-oriented languages is obtained in SQL through the notion of a *reference*. A table may have a *reference column* that serves as the "identity" for its tuples. This column could be the primary key of the table, if there is one, or it could be a column whose values are generated and maintained unique by the DBMS, for example. We shall defer to Section 4.6 the matter of defining reference columns until we first see how reference types are used.

To refer to the tuples of a table with a reference column, an attribute may have as its type a reference to another type. If $T$ is a UDT, then `REF`($T$) is the type of a reference to a tuple of type $T$. Further, the reference may be given a *scope*, which is the name of the relation whose tuples are referred to. Thus, an attribute $A$ whose values are references to tuples in relation $R$, where $R$ is a table whose type is the UDT $T$, would be declared by:

```
CREATE TYPE StarType AS (
    name      CHAR(30),
    address   AddressType,
    bestMovie REF(MovieType) SCOPE Movies
);
```

Figure 20: Adding a best movie reference to `StarType`

A `REF`($T$) `SCOPE` $R$

If no scope is specified, the reference can go to any relation of type $T$.

**Example 18:** Let us record in `MovieStar` the best movie for each star. Assume that we have declared an appropriate relation `Movies`, and that the type of this relation is the UDT `MovieType`; we shall define both `MovieType` and `Movies` later, in Fig. 21. Figure 20 is a new definition of `StarType` that includes an attribute `bestMovies` that is a reference to a movie. Now, if relation `MovieStar` is defined to have the UDT of Fig. 20, then each star tuple will have a component that refers to a `Movies` tuple — the star's best movie. □

## 4.6 Creating Object ID's for Tables

In order to refer to rows of a table, such as `Movies` in Example 18, that table needs to have an "object-ID" for its tuples. Such a table is said to be *referenceable*. In a `CREATE TABLE` statement where the type of the table is a UDT (as in Section 4.4), we may include an element of the form:

`REF IS` <attribute name> <how generated>

The attribute name is a name given to the column that will serve as the object-ID for tuples. The "how generated" clause can be:

1. `SYSTEM GENERATED`, meaning that the DBMS is responsible for maintaining a unique value in this column of each tuple, or
2. `DERIVED`, meaning that the DBMS will use the primary key of the relation to produce unique values for this column.

**Example 19:** Figure 21 shows how the UDT `MovieType` and relation `Movies` could be declared so that `Movies` is referenceable. The UDT is declared in lines (1) through (4). Then the relation `Movies` is defined to have this type in lines (5) through (7). Notice that we have declared `title` and `year`, together, to be the key for relation `Movies` in line (7).

We see in line (6) that the name of the "identity" column for `Movies` is `movieID`. This attribute, which automatically becomes a fourth attribute of

```
1)  CREATE TYPE MovieType AS (
2)      title   CHAR(30),
3)      year    INTEGER,
4)      genre   CHAR(10)
    );

5)  CREATE TABLE Movies OF MovieType (
6)      REF IS movieID SYSTEM GENERATED,
7)      PRIMARY KEY (title, year)
    );
```

Figure 21: Creating a referenceable table

`Movies`, along with `title`, `year`, and `genre`, may be used in queries like any other attribute of `Movies`.

Line (6) also says that the DBMS is responsible for generating the value of `movieID` each time a new tuple is inserted into `Movies`. Had we replaced `SYSTEM GENERATED` by `DERIVED`, then new tuples would get their value of `movieID` by some calculation, performed by the system, on the values of the primary-key attributes `title` and `year` taken from the new tuple. □

**Example 20:** Now, let us see how to represent the many-many relationship between movies and stars using references. Previously, we represented this relationship by a relation like `StarsIn` that contains tuples with the keys of `Movies` and `MovieStar`. As an alternative, we may define `StarsIn` to have references to tuples from these two relations.

First, we need to redefine $\widehat{\text{MovieStar}}$ so it is a referenceable table, thusly:

```
CREATE TABLE MovieStar OF StarType (
    REF IS starID SYSTEM GENERATED,
    PRIMARY KEY (name)
);
```

Then, we may declare the relation `StarsIn` to have two attributes, which are references, one to a movie tuple and one to a star tuple. Here is a direct definition of this relation:

```
CREATE TABLE StarsIn (
    star    REF(StarType) SCOPE MovieStar,
    movie   REF(MovieType) SCOPE Movies
);
```

Optionally, we could have defined a UDT as above, and then declared `StarsIn` to be a table of that type. □

### 4.7 Exercises for Section 4

**Exercise 4.1:** For our running movies example, choose type names for the attributes of each of the relations. Give attributes the same UDT if their values can reasonably be compared or exchanged, and give them different UDT's if they should not have their values compared or exchanged.

**Exercise 4.2:** Write type declarations for the following types:

a) `NameType`, with components for first, middle, and last names and a title.

b) `PersonType`, with a name of the person and references to the persons that are their mother and father. You must use the type from part (a) in your declaration.

c) `MarriageType`, with the date of the marriage and references to the husband and wife.

## 5 Operations on Object-Relational Data

All appropriate SQL operations from previous chapters apply to tables that are declared with a UDT or that have attributes whose type is a UDT. There are also some entirely new operations we can use, such as reference-following. However, some familiar operations, especially those that access or modify columns whose type is a UDT, involve new syntax.

### 5.1 Following References

Suppose $x$ is a value of type `REF(`$T$`)`. Then $x$ refers to some tuple $t$ of type $T$. We can obtain tuple $t$ itself, or components of $t$, by two means:

1. Operator `->` has essentially the same meaning as this operator does in C. That is, if $x$ is a reference to a tuple $t$, and $a$ is an attribute of $t$, then `x->a` is the value of the attribute $a$ in tuple $t$.

2. The `DEREF` operator applies to a reference and produces the tuple referenced.

**Example 21:** Let us use the relation `StarsIn` from Example 20 to find the movies in which Brad Pitt starred. Recall that the schema is

```
StarsIn(star, movie)
```

where `star` and `movie` are references to tuples of `MovieStar` and `Movies`, respectively. A possible query is:

```
1)  SELECT DEREF(movie)
2)  FROM StarsIn
3)  WHERE star->name = 'Brad Pitt';
```

In line (3), the expression `star->name` produces the value of the `name` component of the `MovieStar` tuple referred to by the `star` component of any given `StarsIn` tuple. Thus, the `WHERE` clause identifies those `StarsIn` tuples whose `star` components are references to the Brad-Pitt `MovieStar` tuple. Line (1) then produces the movie tuple referred to by the `movie` component of those tuples. All three attributes — `title`, `year`, and `genre` — will appear in the printed result.

Note that we could have replaced line (1) by:

```
1)  SELECT  movie
```

However, had we done so, we would have gotten a list of system-generated gibberish that serves as the internal unique identifiers for certain movie tuples. We would not see the information in the referenced tuples. □

## 5.2 Accessing Components of Tuples with a UDT

When we define a relation to have a UDT, the tuples must be thought of as single objects, rather than lists with components corresponding to the attributes of the UDT. As a case in point, consider the relation `Movies` declared in Fig. 21. This relation has UDT `MovieType`, which has three attributes: `title`, `year`, and `genre`. However, a tuple $t$ in `Movies` has only *one* component, not three. That component is the object itself.

If we "drill down" into the object, we can extract the values of the three attributes in the type `MovieType`, as well as use any methods defined for that type. However, we have to access these attributes properly, since they are not attributes of the tuple itself. Rather, every UDT has an implicitly defined *observer method* for each attribute of that UDT. The name of the observer

method for an attribute $x$ is $x()$. We apply this method as we would any other method for this UDT; we attach it with a dot to an expression that evaluates to an object of this type. Thus, if $t$ is a variable whose value is of type $T$, and $x$ is an attribute of $T$, then $t.x()$ is the value of $x$ in the tuple (object) denoted by $t$.

**Example 22:** Let us find, from the relation `Movies` of Fig. 21 the year(s) of movies with title *King Kong*. Here is one way to do so:

```
SELECT m.year()
FROM Movies m
WHERE m.title() = 'King Kong';
```

Even though the tuple variable $m$ would appear not to be needed here, we need a variable whose value is an object of type `MovieType` — the UDT for relation `Movies`. The condition of the `WHERE` clause compares the constant `'King Kong'` to the value of `m.title()`, the observer method for attribute `title` applied to a `MovieType` object $m$. Similarly, the value in the `SELECT` clause is expressed `m.year()`; this expression applies the observer method for `year` to the object $m$. □

In practice, object-relational DBMS's do not use method syntax to extract an attribute from an object. Rather, the parentheses are dropped, and we shall do so in what follows. For instance, the query of Example 22 will be written:

```
SELECT m.year
FROM Movies m
WHERE m.title = 'King Kong';
```

The tuple variable $m$ is still necessary, however.

The dot operator can be used to apply methods as well as to find attribute values within objects. These methods should have the parentheses attached, even if they take no arguments.

**Example 23:** Suppose relation `MovieStar` has been declared to have UDT `StarType`, which we should recall from Example 14 has an attribute `address` of type `AddressType`. That type, in turn, has a method `houseNumber()`, which extracts the house number from an object of type `AddressType` (see Example 15). Then the query

```
SELECT MAX(s.address.houseNumber())
FROM MovieStar s
```

extracts the address component from a `StarType` object $s$, then applies the `houseNumber()` method to that `AddressType` object. The result returned is the largest house number of any movie star. □

## 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 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 17) an object constructed from these values, using calls to the proper generator and mutator functions. Recall from Example 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. 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 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 24 with an `INSERT` statement of a familiar form:

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

## 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 25:** Let us consider a possible ordering on the UDT `StarType` from Example 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. 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 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.

## 5.5 Exercises for Section 5

**Exercise 5.1:** Use the `StarsIn` relation of Example 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 5.2:** Assuming the function `AddrLEG` of Fig. 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 5.3:** Write a procedure to take a star name as argument and delete from `StarsIn` and `MovieStar` all tuples involving that star.

# 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 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 7.

## 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 properties of serializable transactions. 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.

## 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 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. 24.

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

Figure 24: Find average sales price by state

Notice how the query of Fig. 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.

## 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 26. Often, it helps to think of the objects in the fact table as arranged in a multidimensional space, or "cube." Figure 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. 25 we could think of each point as a sale of a single automobile, while the dimensions represent properties of that sale.

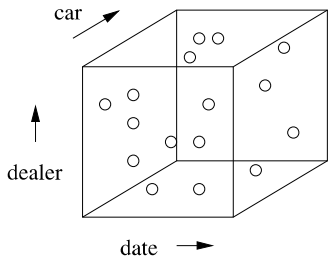



Figure 25: Data organized in a multidimensional space

A data space such as Fig. 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 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 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.

## 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 27:** The `Sales` relation from Example 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. 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.

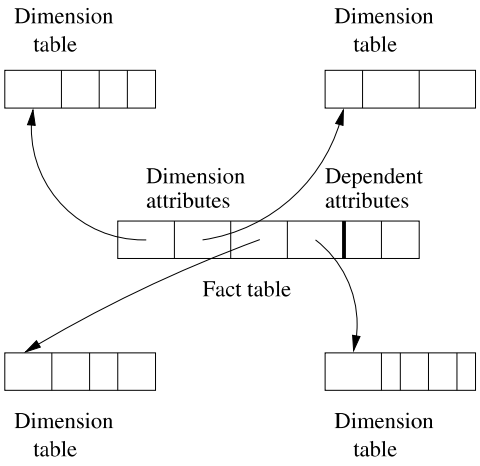



Figure 26: The dimension attributes in the fact table reference the keys of the dimension tables

**Example 28:** For the automobile data of Example 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." □

## 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. 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.

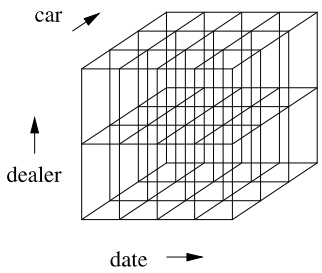



Figure 27: Dicing the cube by partitioning along each dimension

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

**Example 29:** Figure 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. □

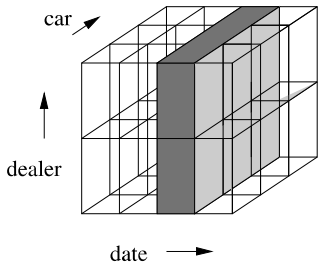



Figure 28: Selecting a slice of a diced cube

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 30:** Let us continue with our automobile example, but include the conceptual `Days` dimension table for time discussed in Example 28. If the Gobi

### Drill-Down and Roll-Up

Example 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 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.

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. 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 29: Final slicing-and-dicing query about red Gobi sales

## 6.6 Exercises for Section 6

**Exercise 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 6.2:** Suppose that we want to examine the data of Exercise 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.

# 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 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.

## 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 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

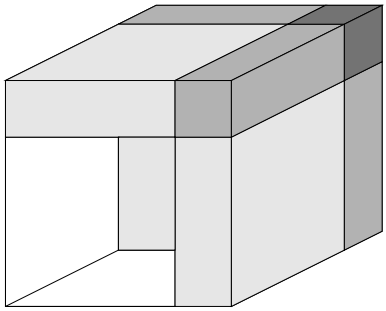


Figure 30: The cube operator augments a data cube with a border of aggregations in all combinations of dimensions

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 31:** Let us reconsider the Aardvark database from Example 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. □

## 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 32:** We can construct a materialized view that is the data cube we called CUBE(Sales) in Example 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 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. □

## 7.3 Exercises for Section 7

**Exercise 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 7.2:** Use the materialized view `SalesCube` from Example 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 7.3:** What help, if any, would the rollup `SalesRollup` of Example 33 be for each of the queries of Exercise 7.2?

**Exercise 7.4:** In Exercise 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 7.5:** Answer the following queries using the data cube from Exercise 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 7.6:** The cube tuples mentioned in Example 32 are not in the rollup of Example 33. Are there other rollups that would contain these tuples?

**!! Exercise 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?

# 8 Summary

- *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.

# 9 References

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

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.

# 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.

## 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 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.

### 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.

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.

## 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 1:** Figure 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

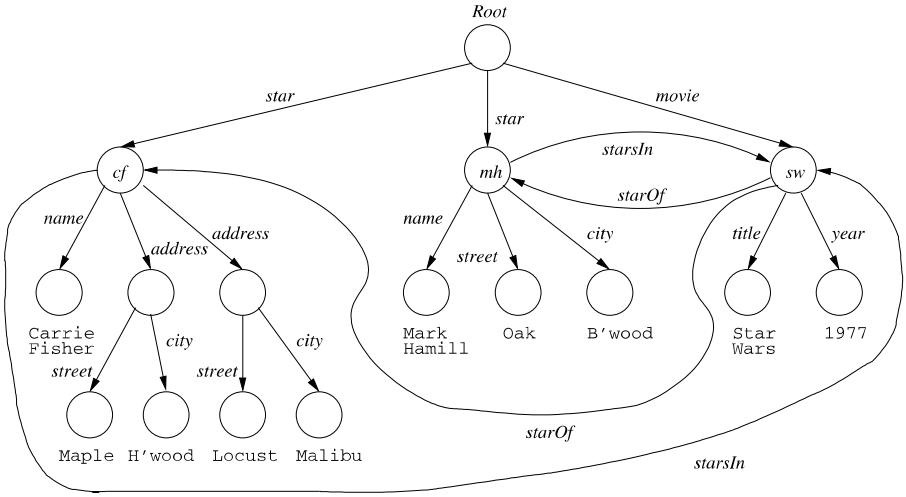



Figure 1: Semistructured data representing a movie and stars

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 2:** Consider Fig. 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. 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. 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. □

## 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 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. 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.

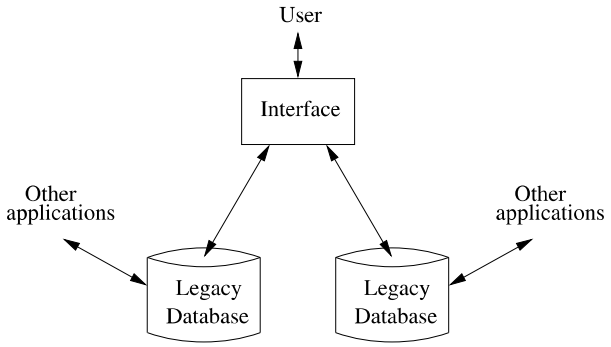



Figure 2: Integrating two legacy databases through an interface that supports semistructured data

**Example 3:** We can see in Fig. 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. 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. □

## 1.4 Exercises for Section 1

**Exercise 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. 1 the facts that *Star Wars* was directed by George Lucas and produced by Gary Kurtz.

b) Add to Fig. 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 1.2:** Suggest how typical data about banks and customers could be represented in the semistructured model.

**Exercise 1.3:** Suggest how typical data about players, teams, and fans could be represented in the semistructured model.

**Exercise 1.4:** Suggest how typical data about a genealogy could be represented in the semistructured model.

**! Exercise 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?

# 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 1. In particular, tags can play the same role as the labels on the arcs of a semistructured-data graph.

## 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 2.4).

## 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 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 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.

## 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 4:** In Fig. 3 is an XML document that corresponds roughly to the data in Fig. 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 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 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. 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. 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 4 is an example of this representation. □

## 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 4: Nesting movies within stars

a semistructured-data graph. Attributes can also be used to represent the "sideways" arcs as in Fig. 1.

**Example 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. 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. □

## 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 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 6:** Figure 5 can be interpreted as an exact representation in XML of the semistructured data graph of Fig. 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. 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 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*. □

## 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 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 7:** Suppose we want to say that in element `StarMovieData` of Fig. 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:`. □

## 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. 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, 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 specialized XML query languages.

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. 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.

### 2.8 Exercises for Section 2

**Exercise 2.1:** Repeat Exercise 1.1 using XML.

**Exercise 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 2.3:** How would you represent an empty element (one that had neither text nor subelements) in the database schema of Section 2.7?

**! Exercise 2.4:** In Section 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.

## 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.

### 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 8:** In Fig. 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 7 is an example of a document that conforms to the DTD of Fig. 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 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. 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 7: Example of a document following the DTD of Fig. 6

## 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 9:** Here is how we might introduce the document of Fig. 7 to assert that it is intended to conform to the DTD of Fig. 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. □

## 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 10:** Instead of having the title and year be subelements of a `<Movie>` element, we could make these be attributes instead. Figure 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 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 9: A DTD for stars and movies, using ID's and IDREF's

## 3.4 Identifiers and References

Recall from Section 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:** Figure 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. 1. The structure differs from that of the DTD in Fig. 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. 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 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. 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. □

### 3.5 Exercises for Section 3

**Exercise 3.1:** Add to the document of Fig. 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 3.2:** Suggest how typical data about banks and customers could be represented as a DTD.

**Exercise 3.3:** Suggest how typical data about players, teams, and fans could be represented as a DTD.

**Exercise 3.4:** Suggest how typical data about a genealogy could be represented as a DTD.

**! Exercise 3.5:** Using your representation from Exercise 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.

## 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.

### 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 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.

## 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 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. □

## 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: 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. 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 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. 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 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 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. 12 says the same thing as the DTD we show in Fig. 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 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.

## 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 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 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 14: Using attributes in place of simple elements

**Example 15:** Figure 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 15 shows the DTD resembling Fig. 14. □

## 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 15: DTD equivalent for Fig. 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. 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 16: Form of a range restriction

**Example 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. 12 or for the attribute `year` in line (5) of Fig. 14, we could define a new simple type as in Fig. 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 17: A type that restricts integer values to be 1915 or greater

**Example 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 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 18: A enumerated type in XML Schema

## 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 18:** Suppose we want to say, about the semistructured data in Fig. 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. 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.

**Example 19:** In Fig. 19 we see an elaboration of Fig. 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 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 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. □

## 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 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 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 20:** Figure 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. 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>`. □

## 4.8 Exercises for Section 4

**Exercise 4.1:** Give an example of a document that conforms to the XML Schema definition of Fig. 12 and an example of one that has all the elements mentioned, but does not conform to the definition.

**Exercise 4.2:** Rewrite Fig. 12 so that there is a named complex type for `Movies`, but no named type for `Movie`.

**Exercise 4.3:** Write the XML Schema definitions of Fig. 19 and 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 20: Stars with a foreign key

# 5 Summary

- *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.

# 6 References

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`

# Index

Page references followed by "f" indicate illustrated figures or photographs; followed by "t" indicates a table.

## B



## C

## D



## E

## N



## O



## P



## Q

## R



## S