Constraints and Triggers

In this chapter we shall cover those aspects of SQL that let us create "active" elements. An *active* element is an expression or statement that we write once and store in the database, expecting the element to execute at appropriate times. The time of action might be when a certain event occurs, such as an insertion into a particular relation, or it might be whenever the database changes so that a certain boolean-valued condition becomes true.

One of the serious problems faced by writers of applications that update the database is that the new information could be wrong in a variety of ways. For example, there are often typographical or transcription errors in manually entered data. We could write application programs in such a way that every insertion, deletion, and update command has associated with it the checks necessary to assure correctness. However, it is better to store these checks in the database, and have the DBMS administer the checks. In this way, we can be sure a check will not be forgotten, and we can avoid duplication of work.

SQL provides a variety of techniques for expressing *integrity constraints* as part of the database schema. In this chapter we shall study the principal methods. We have already seen key constraints, where an attribute or set of attributes is declared to be a key for a relation. SQL supports a form of referential integrity, called a "foreign-key constraint," the requirement that a value in an attribute or attributes of one relation must also appear as a value in an attribute or attributes of another relation. SQL also allows constraints on attributes, constraints on tuples, and interrelation constraints called "assertions." Finally, we discuss "triggers," which are a form of active element that is called into play on certain specified events, such as insertion into a specific relation.

1 Keys and Foreign Keys

Recall that SQL allows us to define an attribute or attributes to be a key for a relation with the keywords `PRIMARY KEY` or `UNIQUE`. SQL also uses the term "key" in connection with certain referential-integrity constraints. These

constraints, called "foreign-key constraints," assert that a value appearing in one relation must also appear in the primary-key component(s) of another relation.

1.1 Declaring Foreign-Key Constraints

A foreign key constraint is an assertion that values for certain attributes must make sense. Consider, for instance, how to express in relational algebra the constraint that the producer "certificate number" for each movie was also the certificate number of some executive in the `MovieExec` relation.

In SQL we may declare an attribute or attributes of one relation to be a *foreign key*, referencing some attribute(s) of a second relation (possibly the same relation). The implication of this declaration is twofold:

1. The referenced attribute(s) of the second relation must be declared `UNIQUE` or the `PRIMARY KEY` for their relation. Otherwise, we cannot make the foreign-key declaration.

2. Values of the foreign key appearing in the first relation must also appear in the referenced attributes of some tuple. More precisely, let there be a foreign-key F that references set of attributes G of some relation. Suppose a tuple t of the first relation has non-`NULL` values in all the attributes of F; call the list of t's values in these attributes $t[F]$. Then in the referenced relation there must be some tuple s that agrees with $t[F]$ on the attributes G. That is, $s[G] = t[F]$.

As for primary keys, we have two ways to declare a foreign key.

a) If the foreign key is a single attribute we may follow its name and type by a declaration that it "references" some attribute (which must be a key — primary or unique) of some table. The form of the declaration is

`REFERENCES` <table>`(`<attribute>`)`

b) Alternatively, we may append to the list of attributes in a `CREATE TABLE` statement one or more declarations stating that a set of attributes is a foreign key. We then give the table and its attributes (which must be a key) to which the foreign key refers. The form of this declaration is:

`FOREIGN KEY (`<attributes>`) REFERENCES` <table>`(`<attributes>`)`

Example 1: Suppose we wish to declare the relation

```
Studio(name, address, presC#)
```

whose primary key is `name` and which has a foreign key `presC#` that references `cert#` of relation

```
MovieExec(name, address, cert#, netWorth)
```

We may declare `presC#` directly to reference `cert#` as follows:

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT REFERENCES MovieExec(cert#)
);
```

An alternative form is to add the foreign key declaration separately, as

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT,
    FOREIGN KEY (presC#) REFERENCES MovieExec(cert#)
);
```

Notice that the referenced attribute, `cert#` in `MovieExec`, is a key of that relation, as it must be. The meaning of either of these two foreign key declarations is that whenever a value appears in the `presC#` component of a `Studio` tuple, that value must also appear in the `cert#` component of some `MovieExec` tuple. The one exception is that, should a particular `Studio` tuple have `NULL` as the value of its `presC#` component, there is no requirement that `NULL` appear as the value of a `cert#` component (but note that `cert#` is a primary key and therefore cannot have `NULL`'s anyway). □

1.2 Maintaining Referential Integrity

The schema designer may choose from among three alternatives to enforce a foreign-key constraint. We can learn the general idea by exploring Example 1, where it is required that a `presC#` value in relation `Studio` also be a `cert#` value in `MovieExec`. The following actions will be prevented by the DBMS (i.e., a run-time exception or error will be generated).

a) We try to insert a new `Studio` tuple whose `presC#` value is not `NULL` and is not the `cert#` component of any `MovieExec` tuple.

b) We try to update a `Studio` tuple to change the `presC#` component to a non-`NULL` value that is not the `cert#` component of any `MovieExec` tuple.

c) We try to delete a `MovieExec` tuple, and its `cert#` component, which is not `NULL`, appears as the `presC#` component of one or more `Studio` tuples.

d) We try to update a `MovieExec` tuple in a way that changes the `cert#` value, and the old `cert#` is the value of `presC#` of some movie studio.

For the first two modifications, where the change is to the relation where the foreign-key constraint is declared, there is no alternative; the system has to reject the violating modification. However, for changes to the referenced relation, of which the last two modifications are examples, the designer can choose among three options:

1. *The Default Policy: Reject Violating Modifications.* SQL has a default policy that any modification violating the referential integrity constraint is rejected.

2. *The Cascade Policy.* Under this policy, changes to the referenced attribute(s) are mimicked at the foreign key. For example, under the cascade policy, when we delete the `MovieExec` tuple for the president of a studio, then to maintain referential integrity the system will delete the referencing tuple(s) from `Studio`. If we update the `cert#` for some movie executive from c_1 to c_2, and there was some `Studio` tuple with c_1 as the value of its `presC#` component, then the system will also update this `presC#` component to have value c_2.

3. *The Set-Null Policy.* Here, when a modification to the referenced relation affects a foreign-key value, the latter is changed to `NULL`. For instance, if we delete from `MoveExec` the tuple for a president of a studio, the system would change the `presC#` value for that studio to `NULL`. If we updated that president's certificate number in `MovieExec`, we would again set `presC#` to `NULL` in `Studio`.

These options may be chosen for deletes and updates, independently, and they are stated with the declaration of the foreign key. We declare them with `ON DELETE` or `ON UPDATE` followed by our choice of `SET NULL` or `CASCADE`.

Example 2: Let us see how we might modify the declaration of

```
Studio(name, address, presC#)
```

in Example 1 to specify the handling of deletes and updates in the

```
MovieExec(name, address, cert#, netWorth)
```

relation. Figure 1 takes the first of the `CREATE TABLE` statements in that example and expands it with `ON DELETE` and `ON UPDATE` clauses. Line (5) says that when we delete a `MovieExec` tuple, we set the `presC#` of any studio of which he or she was the president to `NULL`. Line (6) says that if we update the `cert#` component of a `MovieExec` tuple, then any tuples in `Studio` with the same value in the `presC#` component are changed similarly.

```
1)  CREATE TABLE Studio (
2)     name CHAR(30) PRIMARY KEY,
3)     address VARCHAR(255),
4)     presC# INT REFERENCES MovieExec(cert#)
5)         ON DELETE SET NULL
6)         ON UPDATE CASCADE
    );
```

Figure 1: Choosing policies to preserve referential integrity

Dangling Tuples and Modification Policies

A tuple with a foreign key value that does not appear in the referenced relation is said to be a *dangling tuple*. Recall that a tuple which fails to participate in a join is also called "dangling." The two ideas are closely related. If a tuple's foreign-key value is missing from the referenced relation, then the tuple will not participate in a join of its relation with the referenced relation, if the join is on equality of the foreign key and the key it references (called a *foreign-key join*). The dangling tuples are exactly the tuples that violate referential integrity for this foreign-key constraint.

Note that in this example, the set-null policy makes more sense for deletes, while the cascade policy seems preferable for updates. We would expect that if, for instance, a studio president retires, the studio will exist with a "null" president for a while. However, an update to the certificate number of a studio president is most likely a clerical change. The person continues to exist and to be the president of the studio, so we would like the `presC#` attribute in `Studio` to follow the change. □

1.3 Deferred Checking of Constraints

Let us assume the situation of Example 1, where `presC#` in `Studio` is a foreign key referencing `cert#` of `MovieExec`. Arnold Schwarzenegger retires as Governor of California and decides to found a movie studio, called La Vista Studios, of which he will naturally be the president. If we execute the insertion:

```
INSERT INTO Studio
VALUES('La Vista', 'New York', 23456);
```

we are in trouble. The reason is that there is no tuple of `MovieExec` with certificate number 23456 (the presumed newly issued certificate for Arnold Schwarzenegger), so there is an obvious violation of the foreign-key constraint.

One possible fix is first to insert the tuple for La Vista without a president's certificate, as:

```
INSERT INTO Studio(name, address)
VALUES('La Vista', 'New York');
```

This change avoids the constraint violation, because the La-Vista tuple is inserted with `NULL` as the value of `presC#`, and `NULL` in a foreign key does not require that we check for the existence of any value in the referenced column. However, we must insert a tuple for Arnold Schwarzenegger into `MovieExec`, with his correct certificate number before we can apply an update statement such as

```
UPDATE Studio
SET presC# = 23456
WHERE name = 'La Vista';
```

If we do not fix `MovieExec` first, then this update statement will also violate the foreign-key constraint.

Of course, inserting Arnold Schwarzenegger and his certificate number into `MovieExec` before inserting La Vista into `Studio` will surely protect against a foreign-key violation in this case. However, there are cases of *circular constraints* that cannot be fixed by judiciously ordering the database modification steps we take.

Example 3: If movie executives were limited to studio presidents, then we might want to declare `cert#` to be a foreign key referencing `Studio(presC#)`; we would first have to declare `presC#` to be `UNIQUE`, but that declaration makes sense if you assume a person cannot be the president of two studios at the same time.

Now, it is impossible to insert new studios with new presidents. We can't insert a tuple with a new value of `presC#` into `Studio`, because that tuple would violate the foreign-key constraint from `presC#` to `MovieExec(cert#)`. We can't insert a tuple with a new value of `cert#` into `MovieExec`, because that would violate the foreign-key constraint from `cert#` to `Studio(presC#)`. □

The problem of Example 3 can be solved as follows.

1. First, we must group the two insertions (one into `Studio` and the other into `MovieExec`) into a single transaction.
2. Then, we need a way to tell the DBMS not to check the constraints until after the whole transaction has finished its actions and is about to commit.

To inform the DBMS about point (2), the declaration of any constraint — key, foreign-key, or other constraint types we shall meet later in this chapter — may be followed by one of `DEFERRABLE` or `NOT DEFERRABLE`. The latter

is the default, and means that every time a database modification statement is executed, the constraint is checked immediately afterwards, if the modification could violate the foreign-key constraint. However, if we declare a constraint to be `DEFERRABLE`, then we have the option of having it wait until a transaction is complete before checking the constraint.

We follow the keyword `DEFERRABLE` by either `INITIALLY DEFERRED` or `INITIALLY IMMEDIATE`. In the former case, checking will be deferred to just before each transaction commits. In the latter case, the check will be made immediately after each statement.

Example 4: Figure 2 shows the declaration of `Studio` modified to allow the checking of its foreign-key constraint to be deferred until the end of each transaction. We have also declared `presC#` to be `UNIQUE`, in order that it may be referenced by other relations' foreign-key constraints.

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT UNIQUE
        REFERENCES MovieExec(cert#)
        DEFERRABLE INITIALLY DEFERRED
);
```

Figure 2: Making `presC#` unique and deferring the checking of its foreign-key constraint

If we made a similar declaration for the hypothetical foreign-key constraint from `MovieExec(cert#)` to `Studio(presC#)` mentioned in Example 3, then we could write transactions that inserted two tuples, one into each relation, and the two foreign-key constraints would not be checked until after both insertions had been done. Then, if we insert both a new studio and its new president, and use the same certificate number in each tuple, we would avoid violation of any constraint. □

There are two additional points about deferring constraints that we should bear in mind:

- Constraints of any type can be given names. We shall discuss how to do so in Section 3.1.
- If a constraint has a name, say `MyConstraint`, then we can change a deferrable constraint from immediate to deferred by the SQL statement

  ```
  SET CONSTRAINT MyConstraint DEFERRED;
  ```

  and we can reverse the process by replacing `DEFERRED` in the above to `IMMEDIATE`.

1.4 Exercises for Section 1

Exercise 1.1: Our movie database has keys defined for all its relations.

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

Declare the following referential integrity constraints for the movie database as in Exercise 1.1.

a) The producer of a movie must be someone mentioned in `MovieExec`. Modifications to `MovieExec` that violate this constraint are rejected.

b) Repeat (a), but violations result in the `producerC#` in `Movie` being set to `NULL`.

c) Repeat (a), but violations result in the deletion or update of the offending `Movie` tuple.

d) A movie that appears in `StarsIn` must also appear in `Movie`. Handle violations by rejecting the modification.

e) A star appearing in `StarsIn` must also appear in `MovieStar`. Handle violations by deleting violating tuples.

! Exercise 1.2: We would like to declare the constraint that every movie in the relation `Movie` must appear with at least one star in `StarsIn`. Can we do so with a foreign-key constraint? Why or why not?

2 Constraints on Attributes and Tuples

Within a SQL `CREATE TABLE` statement, we can declare two kinds of constraints:

1. A constraint on a single attribute.
2. A constraint on a tuple as a whole.

In Section 2.1 we shall introduce a simple type of constraint on an attribute's value: the constraint that the attribute not have a `NULL` value. Then in Section 2.2 we cover the principal form of constraints of type (1): *attribute-based* `CHECK` *constraints*. The second type, the tuple-based constraints, are covered in Section 2.3.

There are other, more general kinds of constraints that we shall meet in Sections 4 and 5. These constraints can be used to restrict changes to whole relations or even several relations, as well as to constrain the value of a single attribute or tuple.

2.1 Not-Null Constraints

One simple constraint to associate with an attribute is `NOT NULL`. The effect is to disallow tuples in which this attribute is `NULL`. The constraint is declared by the keywords `NOT NULL` following the declaration of the attribute in a `CREATE TABLE` statement.

Example 5: Suppose relation `Studio` required `presC#` not to be `NULL`, perhaps by changing line (4) of Fig. 1 to:

```
4)    presC# INT REFERENCES MovieExec(cert#) NOT NULL
```

This change has several consequences. For instance:

- We could not insert a tuple into `Studio` by specifying only the name and address, because the inserted tuple would have `NULL` in the `presC#` component.
- We could not use the set-null policy in situations like line (5) of Fig. 1, which tells the system to fix foreign-key violations by making `presC#` be `NULL`.

□

2.2 Attribute-Based `CHECK` Constraints

More complex constraints can be attached to an attribute declaration by the keyword `CHECK` and a parenthesized condition that must hold for every value of this attribute. In practice, an attribute-based `CHECK` constraint is likely to be a simple limit on values, such as an enumeration of legal values or an arithmetic inequality. However, in principle the condition can be anything that could follow `WHERE` in a SQL query. This condition may refer to the attribute being constrained, by using the name of that attribute in its expression. However, if the condition refers to any other relations or attributes of relations, then the relation must be introduced in the `FROM` clause of a subquery (even if the relation referred to is the one to which the checked attribute belongs).

An attribute-based `CHECK` constraint is checked whenever any tuple gets a new value for this attribute. The new value could be introduced by an update for the tuple, or it could be part of an inserted tuple. In the case of an update, the constraint is checked on the new value, not the old value. If the constraint is violated by the new value, then the modification is rejected.

It is important to understand that an attribute-based `CHECK` constraint is not checked if the database modification does not change the attribute with which the constraint is associated. This limitation can result in the constraint becoming violated, if other values involved in the constraint do change. First, let us consider a simple example of an attribute-based check. Then we shall see a constraint that involves a subquery, and also see the consequence of the fact that the constraint is only checked when its attribute is modified.

Example 6: Suppose we want to require that certificate numbers be at least six digits. We could modify line (4) of Fig. 1, a declaration of the schema for relation

```
Studio(name, address, presC#)
```

to be

```
4)    presC# INT REFERENCES MovieExec(cert#)
                  CHECK (presC# >= 100000)
```

For another example, the attribute `gender` of relation

```
MovieStar(name, address, gender, birthdate)
```

is declared to be of data type `CHAR(1)` — that is, a single character. However, we really expect that the only characters that will appear there are `'F'` and `'M'`. The following substitute for line (4) enforces the rule:

```
4) gender CHAR(1) CHECK (gender IN ('F', 'M')),
```

Note that the expression `('F' 'M')` describes a one-component relation with two tuples. The constraint says that the value of any `gender` component must be in this set. □

Example 7: We might suppose that we could simulate a referential integrity constraint by an attribute-based `CHECK` constraint that requires the existence of the referred-to value. The following is an *erroneous* attempt to simulate the requirement that the `presC#` value in a

```
Studio(name, address, presC#)
```

tuple must appear in the `cert#` component of some

```
MovieExec(name, address, cert#, netWorth)
```

tuple. Suppose line (4) of Fig. 1 were replaced by

```
4)     presC# INT CHECK
           (presC# IN (SELECT cert# FROM MovieExec))
```

This statement is a legal attribute-based `CHECK` constraint, but let us look at its effect. Modifications to `Studio` that introduce a `presC#` that is not also a `cert#` of `MovieExec` will be rejected. That is almost what the similar foreign-key constraint would do, except that the attribute-based check will also reject a `NULL` value for `presC#` if there is no `NULL` value for `cert#`. But far more importantly, if we change the `MovieExec` relation, say by deleting the tuple for the president of a studio, this change is invisible to the above `CHECK` constraint. Thus, the deletion is permitted, even though the attribute-based `CHECK` constraint on `presC#` is now violated. □

2.3 Tuple-Based `CHECK` Constraints

To declare a constraint on the tuples of a single table R, we may add to the list of attributes and key or foreign-key declarations, in R's `CREATE TABLE` statement, the keyword `CHECK` followed by a parenthesized condition. This condition can be anything that could appear in a `WHERE` clause. It is interpreted as a condition about a tuple in the table R, and the attributes of R may be referred to by name in this expression. However, as for attribute-based `CHECK` constraints, the condition may also mention, in subqueries, other relations or other tuples of the same relation R.

Limited Constraint Checking: Bug or Feature?

One might wonder why attribute- and tuple-based checks are allowed to be violated if they refer to other relations or other tuples of the same relation. The reason is that such constraints can be implemented much more efficiently than more general constraints can. With attribute- or tuple-based checks, we only have to evaluate that constraint for the tuple(s) that are inserted or updated. On the other hand, assertions must be evaluated every time any one of the relations they mention is changed. The careful database designer will use attribute- and tuple-based checks only when there is no possibility that they will be violated, and will use another mechanism, such as assertions (Section 4) or triggers (Section 5) otherwise.

The condition of a tuple-based `CHECK` constraint is checked every time a tuple is inserted into R and every time a tuple of R is updated. The condition is evaluated for the new or updated tuple. If the condition is false for that tuple, then the constraint is violated and the insertion or update statement that caused the violation is rejected. However, if the condition mentions some other relation in a subquery, and a change to that relation causes the condition to become false for some tuple of R, the check does not inhibit this change. That is, like an attribute-based `CHECK`, a tuple-based `CHECK` is invisible to other relations. In fact, even a deletion from R can cause the condition to become false, if R is mentioned in a subquery.

On the other hand, if a tuple-based check does not have subqueries, then we can rely on its always holding. Here is an example of a tuple-based `CHECK` constraint that involves several attributes of one tuple, but no subqueries.

Example 8: Recall the schema of table `MovieStar`. Figure 3 repeats the `CREATE TABLE` statement with the addition of a primary-key declaration and one other constraint, which is one of several possible "consistency conditions" that we might wish to check. This constraint says that if the star's gender is male, then his name must not begin with `'Ms.'`.

In line (2), `name` is declared the primary key for the relation. Then line (6) declares a constraint. The condition of this constraint is true for every female movie star and for every star whose name does not begin with `'Ms.'`. The only tuples for which it is *not* true are those where the gender is male and the name *does* begin with `'Ms.'`. Those are exactly the tuples we wish to exclude from `MovieStar`. □

```
1)  CREATE TABLE MovieStar (
2)     name CHAR(30) PRIMARY KEY,
3)     address VARCHAR(255),
4)     gender CHAR(1),
5)     birthdate DATE,
6)     CHECK (gender = 'F' OR name NOT LIKE 'Ms.%')
    );
```

Figure 3: A constraint on the table `MovieStar`

Writing Constraints Correctly

Many constraints are like Example 8, where we want to forbid tuples that satisfy two or more conditions. The expression that should follow the check is the `OR` of the negations, or opposites, of each condition; this transformation is one of "DeMorgan's laws": the negation of the `AND` of terms is the `OR` of the negations of the same terms. Thus, in Example 8 the first condition was that the star is male, and we used `gender = 'F'` as a suitable negation (although perhaps `gender <> 'M'` would be the more normal way to phrase the negation). The second condition is that the `name` begins with `'Ms.'`, and for this negation we used the `NOT LIKE` comparison. This comparison negates the condition itself, which would be `name LIKE 'Ms.%'` in SQL.

2.4 Comparison of Tuple- and Attribute-Based Constraints

If a constraint on a tuple involves more than one attribute of that tuple, then it must be written as a tuple-based constraint. However, if the constraint involves only one attribute of the tuple, then it can be written as either a tuple- or attribute-based constraint. In either case, we do not count attributes mentioned in subqueries, so even a attribute-based constraint can mention other attributes of the same relation in subqueries.

When only one attribute of the tuple is involved (not counting subqueries), then the condition checked is the same, regardless of whether a tuple- or attribute-based constraint is written. However, the tuple-based constraint will be checked more frequently than the attribute-based constraint — whenever any attribute of the tuple changes, rather than only when the attribute mentioned in the constraint changes.

2.5 Exercises for Section 2

Exercise 2.1: Write the following constraints for attributes of the relation

```
Movies(title, year, length, genre, studioName, producerC#)
```

a) The year cannot be before 1915.

b) The length cannot be less than 60 nor more than 250.

c) The studio name can only be Disney, Fox, MGM, or Paramount.

Exercise 2.2: Write the following constraints on attributes from this example schema:

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

a) The speed of a laptop must be at least 2.0.

b) The only types of printers are laser, ink-jet, and bubble-jet.

c) The only types of products are PC's, laptops, and printers.

! d) A model of a product must also be the model of a PC, a laptop, or a printer.

Exercise 2.3: Write the following constraints as tuple-based `CHECK` constraints on one of the relations of our running movies example:

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

If the constraint actually involves two relations, then you should put constraints in both relations so that whichever relation changes, the constraint will be checked on insertions and updates. Assume no deletions; it is not always possible to maintain tuple-based constraints in the face of deletions.

a) A star may not appear in a movie made before they were born.

! b) No two studios may have the same address.

! c) A name that appears in `MovieStar` must not also appear in `MovieExec`.

! d) A studio name that appears in `Studio` must also appear in at least one `Movies` tuple.

!! e) If a producer of a movie is also the president of a studio, then they must be the president of the studio that made the movie.

Exercise 2.4: Write the following as tuple-based `CHECK` constraints about our "PC" schema.

a) A PC with a processor speed less than 2.0 must not sell for more than $600.

b) A laptop with a screen size less than 15 inches must have at least a 40 gigabyte hard disk or sell for less than $1000.

Exercise 2.5: Write the following as tuple-based `CHECK` constraints about our "battleships" schema:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

a) No class of ships may have guns with larger than a 16-inch bore.

b) If a class of ships has more than 9 guns, then their bore must be no larger than 14 inches.

! c) No ship can be in battle before it is launched.

! Exercise 2.6: In Examples 6 and 8, we introduced constraints on the `gender` attribute of `MovieStar`. What restrictions, if any, do each of these constraints enforce if the value of `gender` is `NULL`?

3 Modification of Constraints

It is possible to add, modify, or delete constraints at any time. The way to express such modifications depends on whether the constraint involved is associated with an attribute, a table, or (as in Section 4) a database schema.

3.1 Giving Names to Constraints

In order to modify or delete an existing constraint, it is necessary that the constraint have a name. To do so, we precede the constraint by the keyword `CONSTRAINT` and a name for the constraint.

3.2 Altering Constraints on Tables

We mentioned in Section 1.3 that we can switch the checking of a constraint from immediate to deferred or vice-versa with a `SET CONSTRAINT` statement. Other changes to constraints are effected with an `ALTER TABLE` statement.

`ALTER TABLE` statements can affect constraints in several ways. You may drop a constraint with keyword `DROP` and the name of the constraint to be dropped. You may also add a constraint with the keyword `ADD`, followed by the constraint to be added. Note, however, that the added constraint must be of a kind that can be associated with tuples, such as tuple-based constraints, key, or foreign-key constraints. Also note that you cannot add a constraint to a table unless it holds at that time for every tuple in the table.

Name Your Constraints

Remember, it is a good idea to give each of your constraints a name, even if you do not believe you will ever need to refer to it. Once the constraint is created without a name, it is too late to give it one later, should you wish to alter it. However, should you be faced with a situation of having to alter a nameless constraint, you will find that your DBMS probably has a way for you to query it for a list of all your constraints, and that it has given your unnamed constraint an internal name of its own, which you may use to refer to the constraint.

3.3 Exercises for Section 3

Exercise 3.1: Show how to alter your relation schemas for the movie example:

```
Movie(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

in the following ways.

a) Make `title` and `year` the key for `Movie`.

b) Require the referential integrity constraint that the producer of every movie appear in `MovieExec`.

c) Require that no movie length be less than 60 nor greater than 250.

! d) Require that no name appear as both a movie star and movie executive (this constraint need not be maintained in the face of deletions).

! e) Require that no two studios have the same address.

Exercise 3.2: Show how to alter the schemas of the "battleships" database:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

to have the following tuple-based constraints.

a) `Class` and `country` form a key for relation `Classes`.

b) Require the referential integrity constraint that every ship appearing in `Battles` also appears in `Ships`.

c) Require the referential integrity constraint that every ship appearing in `Outcomes` appears in `Ships`.

d) Require that no ship has more than 14 guns.

! e) Disallow a ship being in battle before it is launched.

4 Assertions

The most powerful forms of active elements in SQL are not associated with particular tuples or components of tuples. These elements, called "triggers" and "assertions," are part of the database schema, on a par with tables.

- An assertion is a boolean-valued SQL expression that must be true at all times.
- A trigger is a series of actions that are associated with certain events, such as insertions into a particular relation, and that are performed whenever these events arise.

Assertions are easier for the programmer to use, since they merely require the programmer to state what must be true. However, triggers are the feature DBMS's typically provide as general-purpose, active elements. The reason is that it is very hard to implement assertions efficiently. The DBMS must deduce whether any given database modification could affect the truth of an assertion. Triggers, on the other hand, tell exactly when the DBMS needs to deal with them.

4.1 Creating Assertions

The SQL standard proposes a simple form of *assertion* that allows us to enforce any condition (expression that can follow `WHERE`). Like other schema elements, we declare an assertion with a `CREATE` statement. The form of an assertion is:

`CREATE ASSERTION` <assertion-name> `CHECK (`<condition>`)`

The condition in an assertion must be true when the assertion is created and must remain true; any database modification that causes it to become false will be rejected.[1] Recall that the other types of `CHECK` constraints we have covered can be violated under certain conditions, if they involve subqueries.

4.2 Using Assertions

There is a difference between the way we write tuple-based `CHECK` constraints and the way we write assertions. Tuple-based checks can refer directly to the attributes of that relation in whose declaration they appear. An assertion has no such privilege. Any attributes referred to in the condition must be introduced in the assertion, typically by mentioning their relation in a select-from-where expression.

Since the condition must have a boolean value, it is necessary to combine results in some way to make a single true/false choice. For example, we might write the condition as an expression producing a relation, to which `NOT EXISTS` is applied; that is, the constraint is that this relation is always empty. Alternatively, we might apply an aggregation operator like `SUM` to a column of a relation and compare it to a constant. For instance, this way we could require that a sum always be less than some limiting value.

Example 9: Suppose we wish to require that no one can become the president of a studio unless their net worth is at least $10,000,000. We declare an assertion to the effect that the set of movie studios with presidents having a net worth less than $10,000,000 is empty. This assertion involves the two relations

```
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

The assertion is shown in Fig. 4. □

```
CREATE ASSERTION RichPres CHECK
    (NOT EXISTS
         (SELECT Studio.name
          FROM Studio, MovieExec
          WHERE presC# = cert# AND netWorth < 10000000
         )
    );
```

Figure 4: Assertion guaranteeing rich studio presidents

[1] However, remember from Section 1.3 that it is possible to defer the checking of a constraint until just before its transaction commits. If we do so with an assertion, it may briefly become false until the end of a transaction.

Example 10: Here is another example of an assertion. It involves the relation

```
Movies(title, year, length, genre, studioName, producerC#)
```

and says the total length of all movies by a given studio shall not exceed 10,000 minutes.

```
CREATE ASSERTION SumLength CHECK (10000 >= ALL
    (SELECT SUM(length) FROM Movies GROUP BY studioName)
);
```

As this constraint involves only the relation `Movies`, it seemingly could have been expressed as a tuple-based `CHECK` constraint in the schema for `Movies` rather than as an assertion. That is, we could add to the definition of table `Movies` the tuple-based `CHECK` constraint

```
CHECK (10000 >= ALL
    (SELECT SUM(length) FROM Movies GROUP BY studioName));
```

Notice that in principle this condition applies to every tuple of table `Movies`. However, it does not mention any attributes of the tuple explicitly, and all the work is done in the subquery.

Also observe that if implemented as a tuple-based constraint, the check would not be made on deletion of a tuple from the relation `Movies`. In this example, that difference causes no harm, since if the constraint was satisfied before the deletion, then it is surely satisfied after the deletion. However, if the constraint were a lower bound on total length, rather than an upper bound as in this example, then we could find the constraint violated had we written it as a tuple-based check rather than an assertion. □

As a final point, it is possible to drop an assertion. The statement to do so follows the pattern for any database schema element:

`DROP ASSERTION` <assertion name>

4.3 Exercises for Section 4

Exercise 4.1: Write the following assertions. The database schema is as follows:

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

a) No manufacturer of PC's may also make laptops.

Comparison of Constraints

The following table lists the principal differences among attribute-based checks, tuple-based checks, and assertions.

Type of Constraint	Where Declared	When Activated	Guaranteed to Hold?
Attribute-based `CHECK`	With attribute	On insertion to relation or attribute update	Not if subqueries
Tuple-based `CHECK`	Element of relation schema	On insertion to relation or tuple update	Not if subqueries
Assertion	Element of database schema	On any change to any mentioned relation	Yes

b) A manufacturer of a PC must also make a laptop with at least as great a processor speed.

c) If a laptop has a larger main memory than a PC, then the laptop must also have a higher price than the PC.

d) If the relation `Product` mentions a model and its type, then this model must appear in the relation appropriate to that type.

Exercise 4.2: Write the following as assertions. The database schema is as follows:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

a) No class may have more than 2 ships.

! b) No country may have both battleships and battlecruisers.

! c) No ship with more than 9 guns may be in a battle with a ship having fewer than 9 guns that was sunk.

! d) No ship may be launched before the ship that bears the name of the first ship's class.

! e) For every class, there is a ship with the name of that class.

! Exercise 4.3 : The assertion of Example 9 can be written as two tuple-based constraints. Show how to do so.

5 Triggers

Triggers, sometimes called *event-condition-action rules* or *ECA rules*, differ from the kinds of constraints discussed previously in three ways.

1. Triggers are only awakened when certain *events*, specified by the database programmer, occur. The sorts of events allowed are usually insert, delete, or update to a particular relation. Another kind of event allowed in many SQL systems is a transaction end.

2. Once awakened by its triggering event, the trigger tests a *condition*. If the condition does not hold, then nothing else associated with the trigger happens in response to this event.

3. If the condition of the trigger is satisfied, the *action* associated with the trigger is performed by the DBMS. A possible action is to modify the effects of the event in some way, even aborting the transaction of which the event is part. However, the action could be any sequence of database operations, including operations not connected in any way to the triggering event.

5.1 Triggers in SQL

The SQL trigger statement gives the user a number of different options in the event, condition, and action parts. Here are the principal features.

1. The check of the trigger's condition and the action of the trigger may be executed either on the *state of the database* (i.e., the current instances of all the relations) that exists before the triggering event is itself executed or on the state that exists after the triggering event is executed.

2. The condition and action can refer to both old and/or new values of tuples that were updated in the triggering event.

3. It is possible to define update events that are limited to a particular attribute or set of attributes.

4. The programmer has an option of specifying that the trigger executes either:

   (a) Once for each modified tuple (a *row-level trigger*), or

   (b) Once for all the tuples that are changed in one SQL statement (a *statement-level trigger*; remember that one SQL modification statement can affect many tuples).

Before giving the details of the syntax for triggers, let us consider an example that will illustrate the most important syntactic as well as semantic points. Notice in the example trigger, Fig. 5, the key elements and the order in which they appear:

a) The `CREATE TRIGGER` statement (line 1).

b) A clause indicating the triggering event and telling whether the trigger uses the database state before or after the triggering event (line 2).

c) A `REFERENCING` clause to allow the condition and action of the trigger to refer to the tuple being modified (lines 3 through 5). In the case of an update, such as this one, this clause allows us to give names to the tuple both before and after the change.

d) A clause telling whether the trigger executes once for each modified row or once for all the modifications made by one SQL statement (line 6).

e) The condition, which uses the keyword `WHEN` and a boolean expression (line 7).

f) The action, consisting of one or more SQL statements (lines 8 through 10).

Each of these elements has options, which we shall discuss after working through the example.

Example 11: In Fig. 13 is a SQL trigger that applies to the

```
MovieExec(name, address, cert#, netWorth)
```

table. It is triggered by updates to the `netWorth` attribute. The effect of this trigger is to foil any attempt to lower the net worth of a movie executive.

```
 1)  CREATE TRIGGER NetWorthTrigger
 2)  AFTER UPDATE OF netWorth ON MovieExec
 3)  REFERENCING
 4)      OLD ROW AS OldTuple,
 5)      NEW ROW AS NewTuple
 6)  FOR EACH ROW
 7)  WHEN (OldTuple.netWorth > NewTuple.netWorth)
 8)      UPDATE MovieExec
 9)      SET netWorth = OldTuple.netWorth
10)      WHERE cert# = NewTuple.cert#;
```

Figure 5: A SQL trigger

Line (1) introduces the declaration with the keywords `CREATE TRIGGER` and the name of the trigger. Line (2) then gives the triggering event, namely the update of the `netWorth` attribute of the `MovieExec` relation. Lines (3) through (5) set up a way for the condition and action portions of this trigger to talk about both the old tuple (the tuple before the update) and the new tuple (the tuple after the update). These tuples will be referred to as `OldTuple` and `NewTuple`, according to the declarations in lines (4) and (5), respectively. In the condition and action, these names can be used as if they were tuple variables declared in the `FROM` clause of an ordinary SQL query.

Line (6), the phrase `FOR EACH ROW`, expresses the requirement that this trigger is executed once for each updated tuple. Line (7) is the condition part of the trigger. It says that we only perform the action when the new net worth is lower than the old net worth; i.e., the net worth of an executive has shrunk.

Lines (8) through (10) form the action portion. This action is an ordinary SQL update statement that has the effect of restoring the net worth of the executive to what it was before the update. Note that in principle, every tuple of `MovieExec` is considered for update, but the `WHERE`-clause of line (10) guarantees that only the updated tuple (the one with the proper `cert#`) will be affected. □

5.2 The Options for Trigger Design

Of course Example 11 illustrates only some of the features of SQL triggers. In the points that follow, we shall outline the options that are offered by triggers and how to express these options.

- Line (2) of Fig. 5 says that the condition test and action of the rule are executed on the database state that exists after the triggering event, as indicated by the keyword `AFTER`. We may replace `AFTER` by `BEFORE`, in which case the `WHEN` condition is tested on the database state that exists before the triggering event is executed. If the condition is true, then the action of the trigger is executed on that state. Finally, the event that awakened the trigger is executed, regardless of whether the condition is still true. There is a third option, `INSTEAD OF`, that we will not discuss here.

- Besides `UPDATE`, other possible triggering events are `INSERT` and `DELETE`. The `OF netWorth` clause in line (2) of Fig. 5 is optional for `UPDATE` events, and if present defines the event to be only an update of the attribute(s) listed after the keyword `OF`. An `OF` clause is not permitted for `INSERT` or `DELETE` events; these events make sense for entire tuples only.

- The `WHEN` clause is optional. If it is missing, then the action is executed whenever the trigger is awakened. If present, then the action is executed only if the condition following `WHEN` is true.

- While we showed a single SQL statement as an action, there can be any number of such statements, separated by semicolons and surrounded by `BEGIN...END`.
- When the triggering event of a row-level trigger is an update, then there will be old and new tuples, which are the tuple before the update and after, respectively. We give these tuples names by the `OLD ROW AS` and `NEW ROW AS` clauses seen in lines (4) and (5). If the triggering event is an insertion, then we may use a `NEW ROW AS` clause to give a name for the inserted tuple, and `OLD ROW AS` is disallowed. Conversely, on a deletion `OLD ROW AS` is used to name the deleted tuple and `NEW ROW AS` is disallowed.
- If we omit the `FOR EACH ROW` on line (6) or replace it by the default `FOR EACH STATEMENT`, then a row-level trigger such as Fig. 5 becomes a statement-level trigger. A statement-level trigger is executed once whenever a statement of the appropriate type is executed, no matter how many rows — zero, one, or many — it actually affects. For instance, if we update an entire table with a SQL update statement, a statement-level update trigger would execute only once, while a row-level trigger would execute once for each tuple to which an update was applied.
- In a statement-level trigger, we cannot refer to old and new tuples directly, as we did in lines (4) and (5). However, any trigger — whether row- or statement-level — can refer to the relation of *old tuples* (deleted tuples or old versions of updated tuples) and the relation of *new tuples* (inserted tuples or new versions of updated tuples), using declarations such as `OLD TABLE AS OldStuff` and `NEW TABLE AS NewStuff`.

Example 12: Suppose we want to prevent the average net worth of movie executives from dropping below $500,000. This constraint could be violated by an insertion, a deletion, or an update to the `netWorth` column of

```
MovieExec(name, address, cert#, netWorth)
```

The subtle point is that we might, in one statement insert, delete, or change many tuples of `MovieExec`. During the modification, the average net worth might temporarily dip below $500,000 and then rise above it by the time all the modifications are made. We only want to reject the entire set of modifications if the net worth is below $500,000 at the end of the statement.

It is necessary to write one trigger for each of these three events: insert, delete, and update of relation `MovieExec`. Figure 6 shows the trigger for the update event. The triggers for the insertion and deletion of tuples are similar.

Lines (3) through (5) declare that `NewStuff` and `OldStuff` are the names of relations containing the new tuples and old tuples that are involved in the database operation that awakened our trigger. Note that one database statement can modify many tuples of a relation, and if such a statement executes, there can be many tuples in `NewStuff` and `OldStuff`.

```
 1)  CREATE TRIGGER AvgNetWorthTrigger
 2)  AFTER UPDATE OF netWorth ON MovieExec
 3)  REFERENCING
 4)      OLD TABLE AS OldStuff,
 5)      NEW TABLE AS NewStuff
 6)  FOR EACH STATEMENT
 7)  WHEN (500000 > (SELECT AVG(netWorth) FROM MovieExec))
 8)  BEGIN
 9)      DELETE FROM MovieExec
10)      WHERE (name, address, cert#, netWorth) IN NewStuff;
11)      INSERT INTO MovieExec
12)          (SELECT * FROM OldStuff);
13)  END;
```

Figure 6: Constraining the average net worth

If the operation is an update, then tables `NewStuff` and `OldStuff` are the new and old versions of the updated tuples, respectively. If an analogous trigger were written for deletions, then the deleted tuples would be in `OldStuff`, and there would be no declaration of a relation name like `NewStuff` for `NEW TABLE` in this trigger. Likewise, in the analogous trigger for insertions, the new tuples would be in `NewStuff`, and there would be no declaration of `OldStuff`.

Line (6) tells us that this trigger is executed once for a statement, regardless of how many tuples are modified. Line (7) is the condition. This condition is satisfied if the average net worth *after* the update is less than $500,000.

The action of lines (8) through (13) consists of two statements that restore the old relation `MovieExec` if the condition of the `WHEN` clause is satisfied; i.e., the new average net worth is too low. Lines (9) and (10) remove all the new tuples, i.e., the updated versions of the tuples, while lines (11) and (12) restore the tuples as they were before the update. □

Example 13: An important use of `BEFORE` triggers is to fix up the inserted tuples in some way before they are inserted. Suppose that we want to insert movie tuples into

```
Movies(title, year, length, genre, studioName, producerC#)
```

but sometimes, we will not know the year of the movie. Since `year` is part of the primary key, we cannot have `NULL` for this attribute. However, we could make sure that `year` is not `NULL` with a trigger and replace `NULL` by some suitable value, perhaps one that we compute in a complex way. In Fig. 7 is a trigger that takes the simple expedient of replacing `NULL` by 1915 (something that could be handled by a default value, but which will serve as an example).

Line (2) says that the condition and action execute before the insertion event. In the referencing-clause of lines (3) through (5), we define names for

```
1)  CREATE TRIGGER FixYearTrigger
2)  BEFORE INSERT ON Movies
3)  REFERENCING
4)      NEW ROW AS NewRow
5)      NEW TABLE AS NewStuff
6)  FOR EACH ROW
7)  WHEN NewRow.year IS NULL
8)  UPDATE NewStuff SET year = 1915;
```

Figure 7: Fixing `NULL`'s in inserted tuples

both the new row being inserted and a table consisting of only that row. Even though the trigger executes once for each inserted tuple [because line (6) declares this trigger to be row-level], the condition of line (7) needs to be able to refer to an attribute of the inserted row, while the action of line (8) needs to refer to a table in order to describe an update. □

5.3 Exercises for Section 5

Exercise 5.1: Write the triggers analogous to Fig. 6 for the insertion and deletion events on `MovieExec`.

Exercise 5.2: Write the following as triggers. In each case, disallow or undo the modification if it does not satisfy the stated constraint. The database schema is as follows:

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

a) When updating the price of a PC, check that there is no lower priced PC with the same speed.

b) When inserting a new printer, check that the model number exists in `Product`.

! c) When making any modification to the `Laptop` relation, check that the average price of laptops for each manufacturer is at least $1500.

! d) When updating the RAM or hard disk of any PC, check that the updated PC has at least 100 times as much hard disk as RAM.

! e) When inserting a new PC, laptop, or printer, make sure that the model number did not previously appear in any of `PC`, `Laptop`, or `Printer`.

Exercise 5.3: Write the following as triggers. In each case, disallow or undo the modification if it does not satisfy the stated constraint. The database schema is as follows:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

a) When a new class is inserted into `Classes`, also insert a ship with the name of that class and a `NULL` launch date.

b) When a new class is inserted with a displacement greater than 35,000 tons, allow the insertion, but change the displacement to 35,000.

! c) If a tuple is inserted into `Outcomes`, check that the ship and battle are listed in `Ships` and `Battles`, respectively, and if not, insert tuples into one or both of these relations, with `NULL` components where necessary.

! d) When there is an insertion into `Ships` or an update of the `class` attribute of `Ships`, check that no country has more than 20 ships.

!! e) Check, under all circumstances that could cause a violation, that no ship fought in a battle that was at a later date than another battle in which that ship was sunk.

! Exercise 5.4: Write the following as triggers. In each case, disallow or undo the modification if it does not satisfy the stated constraint. The problems are based on our running movie example:

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

You may assume that the desired condition holds before any change to the database is attempted. Also, prefer to modify the database, even if it means inserting tuples with `NULL` or default values, rather than rejecting the attempted modification.

a) Assure that at all times, any star appearing in `StarsIn` also appears in `MovieStar`.

b) Assure that at all times every movie executive appears as either a studio president, a producer of a movie, or both.

c) Assure that every movie has at least one male and one female star.

d) Assure that the number of movies made by any studio in any year is no more than 100.

e) Assure that the average length of all movies made in any year is no more than 120.

6 Summary

- *Referential-Integrity Constraints*: We can declare that a value appearing in some attribute or set of attributes must also appear in the corresponding attribute(s) of some tuple of the same or another relation. To do so, we use a `REFERENCES` or `FOREIGN KEY` declaration in the relation schema.
- *Attribute-Based Check Constraints*: We can place a constraint on the value of an attribute by adding the keyword `CHECK` and the condition to be checked after the declaration of that attribute in its relation schema.
- *Tuple-Based Check Constraints*: We can place a constraint on the tuples of a relation by adding the keyword `CHECK` and the condition to be checked to the declaration of the relation itself.
- *Modifying Constraints*: A tuple-based check can be added or deleted with an `ALTER` statement for the appropriate table.
- *Assertions*: We can declare an assertion as an element of a database schema. The declaration gives a condition to be checked. This condition may involve one or more relations of the database schema, and may involve the relation as a whole, e.g., with aggregation, as well as conditions about individual tuples.
- *Invoking the Checks*: Assertions are checked whenever there is a change to one of the relations involved. Attribute- and tuple-based checks are only checked when the attribute or relation to which they apply changes by insertion or update. Thus, the latter constraints can be violated if they have subqueries.
- *Triggers*: The SQL standard includes triggers that specify certain events (e.g., insertion, deletion, or update to a particular relation) that awaken them. Once awakened, a condition can be checked, and if true, a specified sequence of actions (SQL statements such as queries and database modifications) will be executed.

7 References

References [5] and [4] survey all aspects of active elements in database systems. [1] discusses recent thinking regarding active elements in SQL-99 and future

standards. References [2] and [3] discuss HiPAC, an early prototype system that offered active database elements.

1. R. J. Cochrane, H. Pirahesh, and N. Mattos, "Integrating triggers and declarative constraints in SQL database systems," *Intl. Conf. on Very Large Database Systems*, pp. 567–579, 1996.

2. U. Dayal et al., "The HiPAC project: combining active databases and timing constraints," *SIGMOD Record* **17**:1, pp. 51–70, 1988.

3. D. R. McCarthy and U. Dayal, "The architecture of an active database management system," *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 215–224, 1989.

4. N. W. Paton and O. Diaz, "Active database systems," *Computing Surveys* **31**:1 (March, 1999), pp. 63–103.

5. J. Widom and S. Ceri, *Active Database Systems*, Morgan-Kaufmann, San Francisco, 1996.

Views and Indexes

We begin this chapter by introducing virtual views, which are relations that are defined by a query over other relations. Virtual views are not stored in the database, but can be queried as if they existed. The query processor will replace the view by its definition in order to execute the query.

Views can also be materialized, in the sense that they are constructed periodically from the database and stored there. The existence of these materialized views can speed up the execution of queries. A very important specialized type of "materialized view" is the index, a stored data structure whose sole purpose is to speed up the access to specified tuples of one of the stored relations. We introduce indexes here and consider the principal issues in selecting the right indexes for a stored table.

1 Virtual Views

Relations that are defined with a `CREATE TABLE` statement actually exist in the database. That is, a SQL system stores tables in some physical organization. They are persistent, in the sense that they can be expected to exist indefinitely and not to change unless they are explicitly told to change by a SQL modification statement.

There is another class of SQL relations, called (*virtual*) *views*, that do not exist physically. Rather, they are defined by an expression much like a query. Views, in turn, can be queried as if they existed physically, and in some cases, we can even modify views.

1.1 Declaring Views

The simplest form of view definition is:

`CREATE VIEW` <view-name> `AS` <view-definition>`;`

The view definition is a SQL query.

Relations, Tables, and Views

SQL programmers tend to use the term "table" instead of "relation." The reason is that it is important to make a distinction between stored relations, which are "tables," and virtual relations, which are "views." Now that we know the distinction between a table and a view, we shall use "relation" only where either a table or view could be used. When we want to emphasize that a relation is stored, rather than a view, we shall sometimes use the term "base relation" or "base table."

There is also a third kind of relation, one that is neither a view nor stored permanently. These relations are temporary results, as might be constructed for some subquery. Temporaries will also be referred to as "relations" subsequently.

Example 1: Suppose we want to have a view that is a part of the

```
Movies(title, year, length, genre, studioName, producerC#)
```

relation, specifically, the titles and years of the movies made by Paramount Studios. We can define this view by

```
1)  CREATE VIEW ParamountMovies AS
2)     SELECT title, year
3)     FROM Movies
4)     WHERE studioName = 'Paramount';
```

First, the name of the view is `ParamountMovies`, as we see from line (1). The attributes of the view are those listed in line (2), namely `title` and `year`. The definition of the view is the query of lines (2) through (4). □

Example 2: Let us consider a more complicated query used to define a view. Our goal is a relation `MovieProd` with movie titles and the names of their producers. The query defining the view involves two relations:

```
Movies(title, year, length, genre, studioName, producerC#)
MovieExec(name, address, cert#, netWorth)
```

The following view definition

```
CREATE VIEW MovieProd AS
    SELECT title, name
    FROM Movies, MovieExec
    WHERE producerC# = cert#;
```

joins the two relations and requires that the certificate numbers match. It then extracts the movie title and producer name from pairs of tuples that agree on the certificates. □

1.2 Querying Views

A view may be queried exactly as if it were a stored table. We mention its name in a `FROM` clause and rely on the DBMS to produce the needed tuples by operating on the relations used to define the virtual view.

Example 3: We may query the view `ParamountMovies` just as if it were a stored table, for instance:

```
SELECT title
FROM ParamountMovies
WHERE year = 1979;
```

finds the movies made by Paramount in 1979. □

Example 4: It is also possible to write queries involving both views and base tables. An example is:

```
SELECT DISTINCT starName
FROM ParamountMovies, StarsIn
WHERE title = movieTitle AND year = movieYear;
```

This query asks for the name of all stars of movies made by Paramount. □

The simplest way to interpret what a query involving virtual views means is to replace each view in a `FROM` clause by a subquery that is identical to the view definition. That subquery is followed by a tuple variable, so we can refer to its tuples. For instance, the query of Example 4 can be thought of as the query of Fig. 1.

```
SELECT DISTINCT starName
FROM (SELECT title, year
      FROM Movies
      WHERE studioName = 'Paramount'
     ) Pm, StarsIn
WHERE Pm.title = movieTitle AND Pm.year = movieYear;
```

Figure 1: Interpreting the use of a virtual view as a subquery

1.3 Renaming Attributes

Sometimes, we might prefer to give a view's attributes names of our own choosing, rather than use the names that come out of the query defining the view. We may specify the attributes of the view by listing them, surrounded by parentheses, after the name of the view in the `CREATE VIEW` statement. For instance, we could rewrite the view definition of Example 2 as:

```
CREATE VIEW MovieProd(movieTitle, prodName) AS
    SELECT title, name
    FROM Movies, MovieExec
    WHERE producerC# = cert#;
```

The view is the same, but its columns are headed by attributes `movieTitle` and `prodName` instead of `title` and `name`.

1.4 Exercises for Section 1

Exercise 1.1: From the following base tables of our running example

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

Construct the following views:

a) A view `RichExec` giving the name, address, certificate number and net worth of all executives with a net worth of at least $10,000,000.

b) A view `StudioPres` giving the name, address, and certificate number of all executives who are studio presidents.

c) A view `ExecutiveStar` giving the name, address, gender, birth date, certificate number, and net worth of all individuals who are both executives and stars.

Exercise 1.2: Write each of the queries below, using one or more of the views from Exercise 1.1 and no base tables.

a) Find the names of females who are both stars and executives.

b) Find the names of those executives who are both studio presidents and worth at least $10,000,000.

! c) Find the names of studio presidents who are also stars and are worth at least $50,000,000.

2 Modifying Views

In limited circumstances it is possible to execute an insertion, deletion, or update to a view. At first, this idea makes no sense at all, since the view does not exist the way a base table (stored relation) does. What could it mean, say, to insert a new tuple into a view? Where would the tuple go, and how would the database system remember that it was supposed to be in the view?

For many views, the answer is simply "you can't do that." However, for sufficiently simple views, called *updatable views*, it is possible to translate the

modification of the view into an equivalent modification on a base table, and the modification can be done to the base table instead. In addition, "instead-of" triggers can be used to turn a view modification into modifications of base tables. In that way, the programmer can force whatever interpretation of a view modification is desired.

2.1 View Removal

An extreme modification of a view is to delete it altogether. This modification may be done whether or not the view is updatable. A typical `DROP` statement is

```
DROP VIEW ParamountMovies;
```

Note that this statement deletes the definition of the view, so we may no longer make queries or issue modification commands involving this view. However dropping the view does not affect any tuples of the underlying relation `Movies`. In contrast,

```
DROP TABLE Movies
```

would not only make the `Movies` table go away. It would also make the view `ParamountMovies` unusable, since a query that used it would indirectly refer to the nonexistent relation `Movies`.

2.2 Updatable Views

SQL provides a formal definition of when modifications to a view are permitted. The SQL rules are complex, but roughly, they permit modifications on views that are defined by selecting (using `SELECT`, not `SELECT DISTINCT`) some attributes from one relation R (which may itself be an updatable view). Two important technical points:

- The `WHERE` clause must not involve R in a subquery.
- The `FROM` clause can only consist of one occurrence of R and no other relation.
- The list in the `SELECT` clause must include enough attributes that for every tuple inserted into the view, we can fill the other attributes out with `NULL` values or the proper default. For example, it is not permitted to project out an attribute that is declared `NOT NULL` and has no default.

An insertion on the view can be applied directly to the underlying relation R. The only nuance is that we need to specify that the attributes in the `SELECT` clause of the view are the only ones for which values are supplied.

Example 5: Suppose we insert into view `ParamountMovies` of Example 1 a tuple like:

```
INSERT INTO ParamountMovies
VALUES('Star Trek', 1979);
```

View `ParamountMovies` meets the SQL updatability conditions, since the view asks only for some components of some tuples of one base table:

```
Movies(title, year, length, genre, studioName, producerC#)
```

The insertion on `ParamountMovies` is executed as if it were the same insertion on `Movies`:

```
INSERT INTO Movies(title, year)
VALUES('Star Trek', 1979);
```

Notice that the attributes `title` and `year` had to be specified in this insertion, since we cannot provide values for other attributes of `Movies`.

The tuple inserted into `Movies` has values `'Star Trek'` for `title`, 1979 for `year`, and `NULL` for the other four attributes. Curiously, the inserted tuple, since it has `NULL` as the value of attribute `studioName`, will not meet the selection condition for the view `ParamountMovies`, and thus, the inserted tuple has no effect on the view. For instance, the query of Example 3 would not retrieve the tuple (`'Star Trek'`, 1979).

To fix this apparent anomaly, we could add `studioName` to the `SELECT` clause of the view, as:

```
CREATE VIEW ParamountMovies AS
   SELECT studioName, title, year
   FROM Movies
   WHERE studioName = 'Paramount';
```

Then, we could insert the *Star-Trek* tuple into the view by:

```
INSERT INTO ParamountMovies
VALUES('Paramount', 'Star Trek', 1979);
```

This insertion has the same effect on `Movies` as:

```
INSERT INTO Movies(studioName, title, year)
VALUES('Paramount', 'Star Trek', 1979);
```

Notice that the resulting tuple, although it has `NULL` in the attributes not mentioned, does yield the appropriate tuple for the view `ParamountMovies`. □

We may also delete from an updatable view. The deletion, like the insertion, is passed through to the underlying relation R. However, to make sure that only tuples that can be seen in the view are deleted, we add (using `AND`) the condition of the `WHERE` clause in the view to the `WHERE` clause of the deletion.

Example 6: Suppose we wish to delete from the updatable `ParamountMovies` view all movies with "Trek" in their titles. We may issue the deletion statement

```
DELETE FROM ParamountMovies
WHERE title LIKE '%Trek%';
```

This deletion is translated into an equivalent deletion on the `Movies` base table; the only difference is that the condition defining the view `ParamountMovies` is added to the conditions of the `WHERE` clause.

```
DELETE FROM Movies
WHERE title LIKE '%Trek%' AND studioName = 'Paramount';
```

is the resulting delete statement. □

Similarly, an update on an updatable view is passed through to the underlying relation. The view update thus has the effect of updating all tuples of the underlying relation that give rise in the view to updated view tuples.

Example 7: The view update

```
UPDATE ParamountMovies
SET year = 1979
WHERE title = 'Star Trek the Movie';
```

is equivalent to the base-table update

```
UPDATE Movies
SET year = 1979
WHERE title = 'Star Trek the Movie' AND
    studioName = 'Paramount';
```

□

2.3 Instead-Of Triggers on Views

When a trigger is defined on a view, we can use `INSTEAD OF` in place of `BEFORE` or `AFTER`. If we do so, then when an event awakens the trigger, the action of the trigger is done instead of the event itself. That is, an instead-of trigger intercepts attempts to modify the view and in its place performs whatever action the database designer deems appropriate. The following is a typical example.

Why Some Views Are Not Updatable

Consider the view `MovieProd` of Example 2, which relates movie titles and producers' names. This view is not updatable according to the SQL definition, because there are two relations in the `FROM` clause: `Movies` and `MovieExec`. Suppose we tried to insert a tuple like

```
('Greatest Show on Earth', 'Cecil B. DeMille')
```

We would have to insert tuples into both `Movies` and `MovieExec`. We could use the default value for attributes like `length` or `address`, but what could be done for the two equated attributes `producerC#` and `cert#` that both represent the unknown certificate number of DeMille? We could use `NULL` for both of these. However, when joining relations with `NULL`'s, SQL does not recognize two `NULL` values as equal. Thus, `'Greatest Show on Earth'` would not be connected with `'Cecil B. DeMille'` in the `MovieProd` view, and our insertion would not have been done correctly.

Example 8: Let us recall the definition of the view of all movies owned by Paramount:

```
CREATE VIEW ParamountMovies AS
   SELECT title, year
   FROM Movies
   WHERE studioName = 'Paramount';
```

from Example 1. As we discussed in Example 5, this view is updatable, but it has the unexpected flaw that when you insert a tuple into `ParamountMovies`, the system cannot deduce that the `studioName` attribute is surely Paramount, so `studioName` is `NULL` in the inserted `Movies` tuple.

A better result can be obtained if we create an instead-of trigger on this view, as shown in Fig. 2. Much of the trigger is unsurprising. We see the keyword `INSTEAD OF` on line (2), establishing that an attempt to insert into `ParamountMovies` will never take place.

Rather, lines (5) and (6) is the action that replaces the attempted insertion. There is an insertion into `Movies`, and it specifies the three attributes that we know about. Attributes `title` and `year` come from the tuple we tried to insert into the view; we refer to these values by the tuple variable `NewRow` that was declared in line (3) to represent the tuple we are trying to insert. The value of attribute `studioName` is the constant `'Paramount'`. This value is not part of the inserted view tuple. Rather, we assume it is the correct studio for the inserted movie, because the insertion came through the view `ParamountMovies`. □

```
1)  CREATE TRIGGER ParamountInsert
2)  INSTEAD OF INSERT ON ParamountMovies
3)  REFERENCING NEW ROW AS NewRow
4)  FOR EACH ROW
5)  INSERT INTO Movies(title, year, studioName)
6)  VALUES(NewRow.title, NewRow.year, 'Paramount');
```

Figure 2: Trigger to replace an insertion on a view by an insertion on the underlying base table

2.4 Exercises for Section 2

Exercise 2.1: Which of the views of Exercise 1.1 are updatable?

Exercise 2.2: Suppose we create the view:

```
CREATE VIEW DisneyComedies AS
    SELECT title, year, length FROM Movies
    WHERE studioName = 'Disney' AND genre = 'comedy';
```

a) Is this view updatable?

b) Write an instead-of trigger to handle an insertion into this view.

c) Write an instead-of trigger to handle an update of the length for a movie (given by title and year) in this view.

Exercise 2.3: Using the base tables

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
```

suppose we create the view:

```
CREATE VIEW NewPC AS
SELECT maker, model, speed, ram, hd, price
FROM Product, PC
WHERE Product.model = PC.model AND type = 'pc';
```

Notice that we have made a check for consistency: that the model number not only appears in the `PC` relation, but the `type` attribute of `Product` indicates that the product is a PC.

a) Is this view updatable?

b) Write an instead-of trigger to handle an insertion into this view.

c) Write an instead-of trigger to handle an update of the price.

d) Write an instead-of trigger to handle a deletion of a specified tuple from this view.

3 Indexes in SQL

An *index* on an attribute A of a relation is a data structure that makes it efficient to find those tuples that have a fixed value for attribute A. We could think of the index as a binary search tree of (key, value) pairs, in which a key a (one of the values that attribute A may have) is associated with a "value" that is the set of locations of the tuples that have a in the component for attribute A. Such an index may help with queries in which the attribute A is compared with a constant, for instance $A = 3$, or even $A \leq 3$. Note that the key for the index can be any attribute or set of attributes, and need not be the key for the relation on which the index is built. We shall refer to the attributes of the index as the *index key* when a distinction needs to be made.

The technology of implementing indexes on large relations is of central importance in the implementation of DBMS's. The most important data structure used by a typical DBMS is the "B-tree," which is a generalization of a balanced binary tree. We shall take up B-trees when we talk about DBMS implementation, but for the moment, thinking of indexes as binary search trees will suffice.

3.1 Motivation for Indexes

When relations are very large, it becomes expensive to scan all the tuples of a relation to find those (perhaps very few) tuples that match a given condition. For example, consider this query:

```
SELECT *
FROM Movies
WHERE studioName = 'Disney' AND year = 1990;
```

There might be 10,000 `Movies` tuples, of which only 200 were made in 1990.

The naive way to implement this query is to get all 10,000 tuples and test the condition of the `WHERE` clause on each. It would be much more efficient if we had some way of getting only the 200 tuples from the year 1990 and testing each of them to see if the studio was Disney. It would be even more efficient if we could obtain directly only the 10 or so tuples that satisfied both the conditions of the `WHERE` clause — that the studio is Disney and the year is 1990; see the discussion of "multiattribute indexes," in Section 3.2.

Indexes may also be useful in queries that involve a join. The following example illustrates the point.

Example 9: Examine the query

```
SELECT name
FROM Movies, MovieExec
WHERE title = 'Star Wars' AND producerC# = cert#;
```

that asks for the name of the producer of *Star Wars*. If there is an index on `title` of `Movies`, then we can use this index to get the tuple for *Star Wars*. From this tuple, we can extract the `producerC#` to get the certificate of the producer.

Now, suppose that there is also an index on `cert#` of `MovieExec`. Then we can use the `producerC#` with this index to find the tuple of `MovieExec` for the producer of *Star Wars*. From this tuple, we can extract the producer's name. Notice that with these two indexes, we look at only the two tuples, one from each relation, that are needed to answer the query. Without indexes, we have to look at every tuple of the two relations. □

3.2 Declaring Indexes

Although the creation of indexes is not part of any SQL standard up to and including SQL-99, most commercial systems have a way for the database designer to say that the system should create an index on a certain attribute for a certain relation. The following syntax is typical. Suppose we want to have an index on attribute `year` for the relation `Movies`. Then we say:

```
CREATE INDEX YearIndex ON Movies(year);
```

The result will be that an index whose name is `YearIndex` will be created on attribute `year` of the relation `Movies`. Henceforth, SQL queries that specify a year may be executed by the SQL query processor in such a way that only those tuples of `Movies` with the specified year are ever examined; there is a resulting decrease in the time needed to answer the query.

Often, a DBMS allows us to build a single index on multiple attributes. This type of index takes values for several attributes and efficiently finds the tuples with the given values for these attributes.

Example 10: Since `title` and `year` form a key for `Movies`, we might expect it to be common that values for both these attributes will be specified, or neither will. The following is a typical declaration of an index on these two attributes:

```
CREATE INDEX KeyIndex ON Movies(title, year);
```

Since (`title`, `year`) is a key, if follows that when we are given a title and year, we know the index will find only one tuple, and that will be the desired tuple. In contrast, if the query specifies both the title and year, but only `YearIndex` is available, then the best the system can do is retrieve all the movies of that year and check through them for the given title.

If, as is often the case, the key for the multiattribute index is really the concatenation of the attributes in some order, then we can even use this index to find all the tuples with a given value in the first of the attributes. Thus, part of the design of a multiattribute index is the choice of the order in which the attributes are listed. For instance, if we were more likely to specify a title

than a year for a movie, then we would prefer to order the attributes as above; if a year were more likely to be specified, then we would ask for an index on `(year, title)`. □

If we wish to delete the index, we simply use its name in a statement like:

```
DROP INDEX YearIndex;
```

3.3 Exercises for Section 3

Exercise 3.1: For our running movies example:

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

Declare indexes on the following attributes or combination of attributes:

a) `studioName`.

b) `address` of `MovieExec`.

c) `genre` and `length`.

4 Selection of Indexes

Choosing which indexes to create requires the database designer to analyze a trade-off. In practice, this choice is one of the principal factors that influence whether a database design gives acceptable performance. Two important factors to consider are:

- The existence of an index on an attribute may speed up greatly the execution of those queries in which a value, or range of values, is specified for that attribute, and may speed up joins involving that attribute as well.
- On the other hand, every index built for one or more attributes of some relation makes insertions, deletions, and updates to that relation more complex and time-consuming.

4.1 A Simple Cost Model

To understand how to choose indexes for a database, we first need to know where the time is spent answering a query. The details of how relations are stored will be taken up when we consider DBMS implementation. But for the moment, let us state that the tuples of a relation are normally distributed

among many pages of a disk.[1] One page, which is typically several thousand bytes at least, will hold many tuples.

To examine even one tuple requires that the whole page be brought into main memory. On the other hand, it costs little more time to examine all the tuples on a page than to examine only one. There is a great time saving if the page you want is already in main memory, but for simplicity we shall assume that never to be the case, and every page we need must be retrieved from the disk.

4.2 Some Useful Indexes

Often, the most useful index we can put on a relation is an index on its key. There are two reasons:

1. Queries in which a value for the key is specified are common. Thus, an index on the key will get used frequently.

2. Since there is at most one tuple with a given key value, the index returns either nothing or one location for a tuple. Thus, at most one page must be retrieved to get that tuple into main memory (although there may be other pages that need to be retrieved to use the index itself).

The following example shows the power of key indexes, even in a query that involves a join.

Example 11: Consider an exhaustive pairing of tuples of `Movies` and `MovieExec` to compute a join. Implementing the join this way requires us to read each of the pages holding tuples of `Movies` and each of the pages holding tuples of `MovieExec` at least once. In fact, since these pages may be too numerous to fit in main memory at the same time, we may have to read each page from disk many times. With the right indexes, the whole query might be done with as few as two page reads.

An index on the key `title` and `year` for `Movies` would help us find the one `Movies` tuple for *Star Wars* quickly. Only one page — the page containing that tuple — would be read from disk. Then, after finding the producer-certificate number in that tuple, an index on the key `cert#` for `MovieExec` would help us quickly find the one tuple for the producer in the `MovieExec` relation. Again, only one page with `MovieExec` tuples would be read from disk, although we might need to read a small number of other pages to use the `cert#` index. □

When the index is not on a key, it may or may not be able to improve the time spent retrieving from disk the tuples needed to answer a query. There are two situations in which an index can be effective, even if it is not on a key.

[1] Pages are usually referred to as "blocks" in discussion of databases, but if you are familiar with a paged-memory system from operating systems you should think of the disk as divided into pages.

1. If the attribute is almost a key; that is, relatively few tuples have a given value for that attribute. Even if each of the tuples with a given value is on a different page, we shall not have to retrieve many pages from disk.

2. If the tuples are "clustered" on that attribute. We *cluster* a relation on an attribute by grouping the tuples with a common value for that attribute onto as few pages as possible. Then, even if there are many tuples, we shall not have to retrieve nearly as many pages as there are tuples.

Example 12: As an example of an index of the first kind, suppose `Movies` had an index on `title` rather than `title` and `year`. Since `title` by itself is not a key for the relation, there would be titles such as *King Kong*, where several tuples matched the index key `title`. If we compared use of the index on `title` with what happens in Example 11, we would find that a search for movies with title *King Kong* would produce three tuples (because there are three movies with that title, from years 1933, 1976, and 2005). It is possible that these tuples are on three different pages, so all three pages would be brought into main memory, roughly tripling the amount of time this step takes. However, since the relation `Movies` probably is spread over many more than three pages, there is still a considerable time saving in using the index.

At the next step, we need to get the three `producerC#` values from these three tuples, and find in the relation `MovieExec` the producers of these three movies. We can use the index on `cert#` to find the three relevant tuples of `MovieExec`. Possibly they are on three different pages, but we still spend less time than we would if we had to bring the entire `MovieExec` relation into main memory. □

Example 13: Now, suppose the only index we have on `Movies` is one on `year`, and we want to answer the query:

```
SELECT *
FROM Movies
WHERE year = 1990;
```

First, suppose the tuples of `Movies` are not clustered by year; say they are stored alphabetically by title. Then this query gains little from the index on `year`. If there are, say, 100 movies per page, there is a good chance that any given page has at least one movie made in 1990. Thus, a large fraction of the pages used to hold the relation `Movies` will have to be brought to main memory.

However, suppose the tuples of `Movies` are clustered on `year`. Then we could use the index on `year` to find only the small number of pages that contained tuples with `year` = 1990. In this case, the `year` index will be of great help. In comparison, an index on the combination of `title` and `year` would be of little help, no matter what attribute or attributes we used to cluster `Movies`. □

4.3 Calculating the Best Indexes to Create

It might seem that the more indexes we create, the more likely it is that an index useful for a given query will be available. However, if modifications are the most frequent action, then we should be very conservative about creating indexes. Each modification on a relation R forces us to change any index on one or more of the modified attributes of R. Thus, we must read and write not only the pages of R that are modified, but also read and write certain pages that hold the index. But even when modifications are the dominant form of database action, it may be an efficiency gain to create an index on a frequently used attribute. In fact, since some modification commands involve querying the database (e.g., an `INSERT` with a select-from-where subquery or a `DELETE` with a condition) one must be very careful how one estimates the relative frequency of modifications and queries.

Remember that the typical relation is stored over many disk blocks (pages), and the principal cost of a query or modification is often the number of pages that need to be brought to main memory. Thus, indexes that let us find a tuple without examining the entire relation can save a lot of time. However, the indexes themselves have to be stored, at least partially, on disk, so accessing and modifying the indexes themselves cost disk accesses. In fact, modification, since it requires one disk access to read a page and another disk access to write the changed page, is about twice as expensive as accessing the index or the data in a query.

To calculate the new value of an index, we need to make assumptions about which queries and modifications are most likely to be performed on the database. Sometimes, we have a history of queries that we can use to get good information, on the assumption that the future will be like the past. In other cases, we may know that the database supports a particular application or applications, and we can see in the code for those applications all the SQL queries and modifications that they will ever do. In either situation, we are able to list what we expect are the most common query and modification forms. These forms can have variables in place of constants, but should otherwise look like real SQL statements. Here is a simple example of the process, and of the calculations that we need to make.

Example 14: Let us consider the relation

```
StarsIn(movieTitle, movieYear, starName)
```

Suppose that there are three database operations that we sometimes perform on this relation:

Q_1: We look for the title and year of movies in which a given star appeared. That is, we execute a query of the form:

```
SELECT movieTitle, movieYear
FROM StarsIn
WHERE starName = s;
```

for some constant s.

Q_2: We look for the stars that appeared in a given movie. That is, we execute a query of the form:

```
SELECT starName
FROM StarsIn
WHERE movieTitle = t AND movieYear = y;
```

for constants t and y.

I: We insert a new tuple into `StarsIn`. That is, we execute an insertion of the form:

```
INSERT INTO StarsIn VALUES(t, y, s);
```

for constants t, y, and s.

Let us make the following assumptions about the data:

1. `StarsIn` occupies 10 pages, so if we need to examine the entire relation the cost is 10.
2. On the average, a star has appeared in 3 movies and a movie has 3 stars.
3. Since the tuples for a given star or a given movie are likely to be spread over the 10 pages of `StarsIn`, even if we have an index on `starName` or on the combination of `movieTitle` and `movieYear`, it will take 3 disk accesses to find the (average of) 3 tuples for a star or movie. If we have no index on the star or movie, respectively, then 10 disk accesses are required.
4. One disk access is needed to read a page of the index every time we use that index to locate tuples with a given value for the indexed attribute(s). If an index page must be modified (in the case of an insertion), then another disk access is needed to write back the modified page.
5. Likewise, in the case of an insertion, one disk access is needed to read a page on which the new tuple will be placed, and another disk access is needed to write back this page. We assume that, even without an index, we can find some page on which an additional tuple will fit, without scanning the entire relation.

Figure 3 gives the costs of each of the three operations; Q_1 (query given a star), Q_2 (query given a movie), and I (insertion). If there is no index, then we must scan the entire relation for Q_1 or Q_2 (cost 10),[2] while an insertion requires

[2]There is a subtle point that we shall ignore here. In many situations, it is possible to store a relation on disk using consecutive pages or tracks. In that case, the cost of retrieving the entire relation may be significantly less than retrieving the same number of pages chosen randomly.

Action	No Index	Star Index	Movie Index	Both Indexes
Q_1	10	4	10	4
Q_2	10	10	4	4
I	2	4	4	6
Average	$2 + 8p_1 + 8p_2$	$4 + 6p_2$	$4 + 6p_1$	$6 - 2p_1 - 2p_2$

Figure 3: Costs associated with the three actions, as a function of which indexes are selected

merely that we access a page with free space and rewrite it with the new tuple (cost of 2, since we assume that page can be found without an index). These observations explain the column labeled "No Index."

If there is an index on stars only, then Q_2 still requires a scan of the entire relation (cost 10). However, Q_1 can be answered by accessing one index page to find the three tuples for a given star and then making three more accesses to find those tuples. Insertion I requires that we read and write both a page for the index and a page for the data, for a total of 4 disk accesses.

The case where there is an index on movies only is symmetric to the case for stars only. Finally, if there are indexes on both stars and movies, then it takes 4 disk accesses to answer either Q_1 or Q_2. However, insertion I requires that we read and write two index pages as well as a data page, for a total of 6 disk accesses. That observation explains the last column in Fig. 3.

The final row in Fig. 3 gives the average cost of an action, on the assumption that the fraction of the time we do Q_1 is p_1 and the fraction of the time we do Q_2 is p_2; therefore, the fraction of the time we do I is $1 - p_1 - p_2$.

Depending on p_1 and p_2, any of the four choices of index/no index can yield the best average cost for the three actions. For example, if $p_1 = p_2 = 0.1$, then the expression $2+8p_1+8p_2$ is the smallest, so we would prefer not to create any indexes. That is, if we are doing mostly insertion, and very few queries, then we don't want an index. On the other hand, if $p_1 = p_2 = 0.4$, then the formula $6 - 2p_1 - 2p_2$ turns out to be the smallest, so we would prefer indexes on both `starName` and on the (`movieTitle, movieYear`) combination. Intuitively, if we are doing a lot of queries, and the number of queries specifying movies and stars are roughly equally frequent, then both indexes are desired.

If we have $p_1 = 0.5$ and $p_2 = 0.1$, then an index on stars only gives the best average value, because $4+6p_2$ is the formula with the smallest value. Likewise, $p_1 = 0.1$ and $p_2 = 0.5$ tells us to create an index on only movies. The intuition is that if only one type of query is frequent, create only the index that helps that type of query. □

4.4 Automatic Selection of Indexes to Create

"Tuning" a database is a process that includes not only index selection, but the choice of many different parameters. We have not yet discussed much about

physical implementation of databases, but some examples of tuning issues are the amount of main memory to allocate to various processes and the rate at which backups and checkpoints are made (to facilitate recovery from a crash). There are a number of tools that have been designed to take the responsibility from the database designer and have the system tune itself, or at least advise the designer on good choices.

We shall mention some of these projects in the bibliographic notes for this chapter. However, here is an outline of how the index-selection portion of tuning advisors work.

1. The first step is to establish the query workload. Since a DBMS normally logs all operations anyway, we may be able to examine the log and find a set of representative queries and database modifications for the database at hand. Or it is possible that we know, from the application programs that use the database, what the typical queries will be.

2. The designer may be offered the opportunity to specify some constraints, e.g., indexes that must, or must not, be chosen.

3. The tuning advisor generates a set of possible *candidate* indexes, and evaluates each one. Typical queries are given to the query optimizer of the DBMS. The query optimizer has the ability to estimate the running times of these queries under the assumption that one particular set of indexes is available.

4. The index set resulting in the lowest cost for the given workload is suggested to the designer, or it is automatically created.

A subtle issue arises when we consider possible indexes in step (3). The existence of previously chosen indexes may influence how much *benefit* (improvement in average execution time of the query mix) another index offers. A "greedy" approach to choosing indexes has proven effective.

a) Initially, with no indexes selected, evaluate the benefit of each of the candidate indexes. If at least one provides positive benefit (i.e., it reduces the average execution time of queries), then choose that index.

b) Then, reevaluate the benefit of each of the remaining candidate indexes, assuming that the previously selected index is also available. Again, choose the index that provides the greatest benefit, assuming that benefit is positive.

c) In general, repeat the evaluation of candidate indexes under the assumption that all previously selected indexes are available. Pick the index with maximum benefit, until no more positive benefits can be obtained.

4.5 Exercises for Section 4

Exercise 4.1: Suppose that the relation `StarsIn` discussed in Example 14 required 100 pages rather than 10, but all other assumptions of that example continued to hold. Give formulas in terms of p_1 and p_2 to measure the cost of queries Q_1 and Q_2 and insertion I, under the four combinations of index/no index discussed there.

! Exercise 4.2: In this problem, we consider indexes for the relation

```
Ships(name, class, launched)
```

from our running battleships exercise. Assume:

i. `name` is the key.

ii. The relation `Ships` is stored over 50 pages.

iii. The relation is clustered on `class` so we expect that only one disk access is needed to find the ships of a given class.

iv. On average, there are 5 ships of a class, and 25 ships launched in any given year.

v. With probability p_1 the operation on this relation is a query of the form `SELECT * FROM Ships WHERE name =` n.

vi. With probability p_2 the operation on this relation is a query of the form `SELECT * FROM Ships WHERE class =` c.

vii. With probability p_3 the operation on this relation is a query of the form `SELECT * FROM Ships WHERE launched =` y.

viii. With probability $1 - p_1 - p_2 - p_3$ the operation on this relation is an insertion of a new tuple into `Ships`.

You can also make the assumptions about accessing indexes and finding empty space for insertions that were made in Example 14.

Consider the creation of indexes on `name`, `class`, and `launched`. For each combination of indexes, estimate the average cost of an operation. As a function of p_1, p_2, and p_3, what is the best choice of indexes?

5 Materialized Views

A view describes how a new relation can be constructed from base tables by executing a query on those tables. Until now, we have thought of views only as logical descriptions of relations. However, if a view is used frequently enough, it may even be efficient to *materialize* it; that is, to maintain its value at all times. As with maintaining indexes, there is a cost involved in maintaining a materialized view, since we must recompute parts of the materialized view each time one of the underlying base tables changes.

5.1 Maintaining a Materialized View

In principle, the DBMS needs to recompute a materialized view every time one of its base tables changes in any way. For simple views, it is possible to limit the number of times we need to consider changing the materialized view, and it is possible to limit the amount of work we do when we must maintain the view. We shall take up an example of a join view, and see that there are a number of opportunities to simplify our work.

Example 15: Suppose we frequently want to find the name of the producer of a given movie. We might find it advantageous to materialize a view:

```
CREATE MATERIALIZED VIEW MovieProd AS
    SELECT title, year, name
    FROM Movies, MovieExec
    WHERE producerC# = cert#
```

To start, the DBMS does not have to consider the effect on `MovieProd` of an update on any attribute of `Movies` or `MovieExec` that is not mentioned in the query that defines the materialized view. Surely any modification to a relation that is neither `Movies` nor `MovieExec` can be ignored as well. However, there are a number of other simplifications that enable us to handle other modifications to `Movies` or `MovieExec` more efficiently than a re-execution of the query that defines the materialized view.

1. Suppose we insert a new movie into `Movies`, say `title = 'Kill Bill'`, `year` = 2003, and `producerC#` = 23456. Then we only need to look up `cert#` = 23456 in `MovieExec`. Since `cert#` is the key for `MovieExec`, there can be at most one name returned by the query

   ```
   SELECT name FROM MovieExec
   WHERE cert# = 23456;
   ```

   As this query returns `name = 'Quentin Tarantino'`, the DBMS can insert the proper tuple into `MovieProd` by:

   ```
   INSERT INTO MovieProd
   VALUES('Kill Bill', 2003, 'Quentin Tarantino');
   ```

   Note that, since `MovieProd` is materialized, it is stored like any base table, and this operation makes sense; it does not have to be reinterpreted by an instead-of trigger or any other mechanism.

2. Suppose we delete a movie from `Movies`, say the movie with `title = 'Dumb & Dumber'` and `year` = 1994. The DBMS has only to delete this one movie from `MovieProd` by:

```
DELETE FROM MovieProd
WHERE title = 'Dumb & Dumber' AND year = 1994;
```

3. Suppose we insert a tuple into `MovieExec`, and that tuple has `cert#` = 34567 and `name = 'Max Bialystock'`. Then the DBMS may have to insert into `MovieProd` some movies that were not there because their producer was previously unknown. The operation is:

```
INSERT INTO MovieProd
    SELECT title, year, 'Max Bialystock'
    FROM Movies
    WHERE producerC# = 34567;
```

4. Suppose we delete the tuple with `cert#` = 45678 from `MovieExec`. Then the DBMS must delete from `MovieProd` all movies that have `producerC#` = 45678, because there now can be no matching tuple in `MovieExec` for their underlying `Movies` tuple. Thus, the DBMS executes:

```
DELETE FROM MovieProd
WHERE (title, year) IN
    (SELECT title, year FROM Movies
     WHERE producerC# = 45678);
```

   Notice that it is not sufficient to look up the `name` corresponding to 45678 in `MovieExec` and delete all movies from `MovieProd` that have that producer name. The reason is that, because `name` is not a key for `MovieExec`, there could be two producers with the same name.

We leave as an exercise the consideration of how updates to `Movies` that involve `title` or `year` are handled, and how updates to `MovieExec` involving `cert#` are handled. □

The most important thing to take away from Example 15 is that all the changes to the materialized view are *incremental*. That is, we never have to reconstruct the whole view from scratch. Rather, insertions, deletions, and updates to a base table can be implemented in a join view such as `MovieProd` by a small number of queries to the base tables followed by modification statements on the materialized view. Moreover, these modifications do not affect all the tuples of the view, but only those that have at least one attribute with a particular constant.

It is not possible to find rules such as those in Example 15 for any materialized view we could construct; some are just too complicated. However, many common types of materialized view *do* allow the view to be maintained incrementally. We shall explore another common type of materialized view — aggregation views — in the exercises.

5.2 Periodic Maintenance of Materialized Views

There is another setting in which we may use materialized views, yet not have to worry about the cost or complexity of maintaining them up-to-date as the underlying base tables change. For the moment let us remark that it is common for databases to serve two purposes. For example, a department store may use its database to record its current inventory; this data changes with every sale. The same database may be used by analysts to study buyer patterns and to predict when the store is going to need to restock an item.

The analysts' queries may be answered more efficiently if they can query materialized views, especially views that aggregate data (e.g., sum the inventories of different sizes of shirt after grouping by style). But the database is updated with each sale, so modifications are far more frequent than queries. When modifications dominate, it is costly to have materialized views, or even indexes, on the data.

What is usually done is to create materialized views, but not to try to keep them up-to-date as the base tables change. Rather, the materialized views are reconstructed periodically (typically each night), when other activity in the database is low. The materialized views are only used by analysts, and their data might be out of date by as much as 24 hours. However, in normal situations, the rate at which an item is bought by customers changes slowly. Thus, the data will be "good enough" for the analysts to predict items that are selling well and those that are selling poorly. Of course if Brad Pitt is seen wearing a Hawaiian shirt one morning, and every cool guy has to buy one by that evening, the analysts will not notice they are out of Hawaiian shirts until the next morning, but the risk of that sort of occurrence is low.

5.3 Rewriting Queries to Use Materialized Views

A materialized view can be referred to in the `FROM` clause of a query, just as a virtual view can (Section 1.2). However, because a materialized view is stored in the database, it is possible to rewrite a query to use a materialized view, even if that view was not mentioned in the query as written. Such a rewriting may enable the query to execute much faster, because the hard parts of the query, e.g., joining of relations, may have been carried out already when the materialized view was constructed.

However,we must be very careful to check that the query can be rewritten to use a materialized view. A complete set of rules that will let us use materialized views of any kind is beyond the scope of this book. However, we shall offer a relatively simple rule that applies to the view of Example 15 and similar views.

Suppose we have a materialized view V defined by a query of the form:

```
SELECT L_V
FROM R_V
WHERE C_V
```

where L_V is a list of attributes, R_V is a list of relations, and C_V is a condition. Similarly, suppose we have a query Q of the same form:

SELECT L_Q
FROM R_Q
WHERE C_Q

Here are the conditions under which we can replace part of the query Q by the view V.

1. The relations in list R_V all appear in the list R_Q.
2. The condition C_Q is equivalent to C_V AND C for some condition C. As a special case, C_Q could be equivalent to C_V, in which case the "AND C" is unnecessary.
3. If C is needed, then the attributes of relations on list R_V that C mentions are attributes on the list L_V.
4. Attributes on the list L_Q that come from relations on the list R_V are also on the list L_V.

If all these conditions are met, then we can rewrite Q to use V, as follows:

a) Replace the list R_Q by V and the relations that are on list R_Q but not on R_V.

b) Replace C_Q by C. If C is not needed (i.e., $C_V = C_Q$), then there is no WHERE clause.

Example 16: Suppose we have the materialized view MovieProd from Example 15. This view is defined by the query V:

```
SELECT title, year, name
FROM Movies, MovieExec
WHERE producerC# = cert#
```

Suppose also that we need to answer the query Q that asks for the names of the stars of movies produced by Max Bialystock. For this query we need the relations:

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieExec(name, address, cert#, netWorth)
```

The query Q can be written:

```
SELECT starName
FROM StarsIn, Movies, MovieExec
WHERE movieTitle = title AND movieYear = year AND
    producerC# = cert# AND name = 'Max Bialystock';
```

Let us compare the view definition V with the query Q, to see that they meet the conditions listed above.

1. The relations in the `FROM` clause of V are all in the `FROM` clause of Q.
2. The condition from Q can be written as the condition from V `AND` C, where $C =$

```
movieTitle = title AND movieYear = year AND
    name = 'Max Bialystock'
```

3. The attributes of C that come from relations of V (`Movies` and `MovieExec`) are `title`, `year`, and `name`. These attributes all appear in the `SELECT` clause of V.
4. No attribute from the `SELECT` list of Q is from a relation that appears in the `FROM` list of V.

We may thus use V in Q, yielding the rewritten query:

```
SELECT starName
FROM StarsIn, MovieProd
WHERE movieTitle = title AND movieYear = year AND
    name = 'Max Bialystock';
```

That is, we replaced `Movies` and `MovieExec` in the `FROM` clause by the materialized view `MovieProd`. We also removed the condition of the view from the `WHERE` clause, leaving only the condition C. Since the rewritten query involves the join of only two relations, rather than three, we expect the rewritten query to execute in less time than the original. □

5.4 Automatic Creation of Materialized Views

The ideas that were discussed in Section 4.4 for indexes can apply as well to materialized views. We first need to establish or approximate the query workload. An automated materialized-view-selection advisor needs to generate candidate views. This task can be far more difficult than generating candidate indexes. In the case of indexes, there is only one possibile index for each attribute of each relation. We could also consider indexes on small sets of attributes of a relation, but even if we do, generating all the candidate indexes is straightforward. However, with materialized views, any query could in principle define a view, so there is no limit on what views we need to consider.

The process can be limited if we remember that there is no point in creating a materialized view that does not help for at least one query of our expected workload. For example, suppose some or all of the queries in our workload have the form considered in Section 5.3. Then we can use the analysis of that section to find the views that can help a given query. We can limit ourselves to candidate materialized views that:

1. Have a list of relations in the `FROM` clause that is a subset of those in the `FROM` clause of at least one query of the workload.
2. Have a `WHERE` clause that is the `AND` of conditions that each appear in at least one query.
3. Have a list of attributes in the `SELECT` clause that is sufficient to be used in at least one query.

To evaluate the benefit of a materialized view, let the query optimizer estimate the running times of the queries, both with and without the materialized view. Of course, the optimizer must be designed to take advantage of materialized views; all modern optimizers know how to exploit indexes, but not all can exploit materialized views. Section 5.3 was an example of the reasoning that would be necessary for a query optimizer to perform, if it were to take advantage of such views.

There is another issue that comes up when we consider automatic choice of materialized views, but that did not surface for indexes. An index on a relation is generally smaller than the relation itself, and all indexes on one relation take roughly the same amount of space. However, materialized views can vary radically in size, and some — those involving joins — can be very much larger than the relation or relations on which they are built. Thus, we may need to rethink the definition of the "benefit" of a materialized view. For example, we might want to define the benefit to be the improvement in average running time of the query workload divided by the amount of space the view occupies.

5.5 Exercises for Section 5

Exercise 5.1: Complete Example 15 by considering updates to either of the base tables.

! Exercise 5.2: Suppose the view `NewPC` of Exercise 2.3 were a materialized view. What modifications to the base tables `Product` and `PC` would require a modification of the materialized view? How would you implement those modifications incrementally?

! Exercise 5.3: This exercise explores materialized views that are based on aggregation of data. Suppose we build a materialized view on the base tables

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
```

from our running battleships exercise, as follows:

```
CREATE MATERIALIZED VIEW ShipStats AS
    SELECT country, AVG(displacement), COUNT(*)
    FROM Classes, Ships
    WHERE Classes.class = Ships.class
    GROUP BY country;
```

What modifications to the base tables `Classes` and `Ships` would require a modification of the materialized view? How would you implement those modifications incrementally?

! Exercise 5.4: In Section 5.3 we gave conditions under which a materialized view of simple form could be used in the execution of a query of similar form. For the view of Example 15, describe all the queries of that form, for which this view could be used.

6 Summary

- *Virtual Views*: A virtual view is a definition of how one relation (the view) may be constructed logically from tables stored in the database or other views. Views may be queried as if they were stored relations. The query processor modifies queries about a view so the query is instead about the base tables that are used to define the view.

- *Updatable Views*: Some virtual views on a single relation are updatable, meaning that we can insert into, delete from, and update the view as if it were a stored table. These operations are translated into equivalent modifications to the base table over which the view is defined.

- *Instead-Of Triggers*: SQL allows a special type of trigger to apply to a virtual view. When a modification to the view is called for, the instead-of trigger turns the modification into operations on base tables that are specified in the trigger.

- *Indexes*: While not part of the SQL standard, commercial SQL systems allow the declaration of indexes on attributes; these indexes speed up certain queries or modifications that involve specification of a value, or range of values, for the indexed attribute(s).

- *Choosing Indexes*: While indexes speed up queries, they slow down database modifications, since the indexes on the modified relation must also be modified. Thus, the choice of indexes is a complex problem, depending on the actual mix of queries and modifications performed on the database.

- *Automatic Index Selection*: Some DBMS's offer tools that choose indexes for a database automatically. They examine the typical queries and modifications performed on the database and evaluate the cost trade-offs for different indexes that might be created.

- *Materialized Views*: Instead of treating a view as a query on base tables, we can use the query as a definition of an additional stored relation, whose value is a function of the values of the base tables.

- *Maintaining Materialized Views*: As the base tables change, we must make the corresponding changes to any materialized view whose value is affected by the change. For many common kinds of materialized views, it is possible to make the changes to the view incrementally, without recomputing the entire view.
- *Rewriting Queries to Use Materialized Views*: The conditions under which a query can be rewritten to use a materialized view are complex. However, if the query optimizer can perform such rewritings, then an automatic design tool can consider the improvement in performance that results from creating materialized views and can select views to materialize, automatically.

7 References

The technology behind materialized views is surveyed in [2] and [7]. Reference [3] introduces the greedy algorithm for selecting materialized views.

Two projects for automatically tuning databases are AutoAdmin at Microsoft and SMART at IBM. Current information on AutoAdmin can be found on-line at [8]. A description of the technology behind this system is in [1].

A survey of the SMART project is in [4]. The index-selection aspect of the project is described in [6].

Reference [5] surveys index selection, materialized views, automatic tuning, and related subjects covered in this chapter.

1. S. Agrawal, S. Chaudhuri, and V. R. Narasayya, "Automated selection of materialized views and indexes in SQL databases," *Intl. Conf. on Very Large Databases*, pp. 496–505, 2000.
2. A. Gupta and I. S. Mumick, *Materialized Views: Techniques, Implementations, and Applications*, MIT Press, Cambridge MA, 1999.
3. V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Implementing data cubes efficiently," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1996), pp. 205–216.
4. S. S. Lightstone, G. Lohman, and S. Zilio, "Toward autonomic computing with DB2 universal database," *SIGMOD Record* **31**:3, pp. 55–61, 2002.
5. S. S. Lightstone, T. Teorey, and T. Nadeau, *Physical Database Design*, Morgan-Kaufmann, San Francisco, 2007.
6. G. Lohman, G. Valentin, D. Zilio, M. Zuliani, and A. Skelley, "DB2 Advisor: an optimizer smart enough to recommend its own indexes," *Proc. Sixteenth IEEE Conf. on Data Engineering*, pp. 101–110, 2000.
7. D. Lomet and J. Widom (eds.), Special issue on materialized views and data warehouses, *IEEE Data Engineering Bulletin* **18**:2 (1995).

8. Microsoft on-line description of the AutoAdmin project.
`http://research.microsoft.com/dmx/autoadmin/`

SQL in a Server Environment

We now turn to the question of how SQL fits into a complete programming environment. The typical server environment is introduced in Section 1. Section 2 introduces the SQL terminology for client-server computing and connecting to a database.

Then, we turn to how programming is really done, when SQL must be used to access a database as part of a typical application. In Section 3 we see how to embed SQL in programs that are written in an ordinary programming language, such as C. A critical issue is how we move data between SQL relations and the variables of the surrounding, or "host," language. Section 4 considers another way to combine SQL with general-purpose programming: persistent stored modules, which are pieces of code stored as part of a database schema and executable on command from the user.

A third programming approach is a "call-level interface," where we program in some conventional language and use a library of functions to access the database. In Section 5 we discuss the SQL-standard library called SQL/CLI, for making calls from C programs. Then, in Section 6 we meet Java's JDBC (database connectivity), which is an alternative call-level interface. Finally, another popular call-level interface, PHP, is covered in Section 7.

1 The Three-Tier Architecture

Databases are used in many different settings, including small, standalone databases. For example, a scientist may run a copy of MySQL or Microsoft Access on a laboratory computer to store experimental data. However, there is a very common architecture for large database installations; this architecture motivates the discussion of the entire chapter. The architecture is called *three-tier* or *three-layer*, because it distinguishes three different, interacting functions:

1. *Web Servers.* These are processes that connect clients to the database system, usually over the Internet or possibly a local connection.

2. *Application Servers.* These processes perform the "business logic," whatever it is the system is intended to do.

3. *Database Servers.* These processes run the DBMS and perform queries and modifications at the request of the application servers.

The processes may all run on the same processor in a small system, but it is common to dedicate a large number of processors to each of the tiers. Figure 1 suggests how a large database installation would be organized.

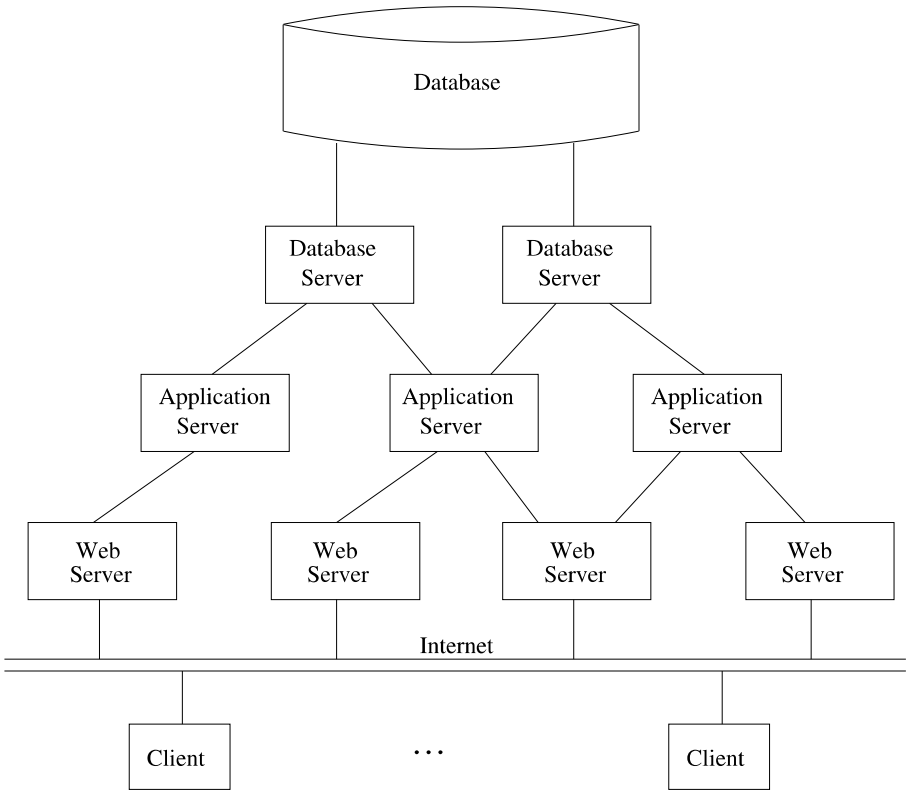

Figure 1: The Three-Tier Architecture

1.1 The Web-Server Tier

The web-server processes manage the interactions with the user. When a user makes contact, perhaps by opening a URL, a web server, typically running

Apache/Tomcat, responds to the request. The user then becomes a *client* of this web-server process. Typically, the client's actions are performed by the web-browser, e.g., managing of the filling of forms, which are then posted to the web server.

As an example, let us consider a site such as Amazon.com. A user (customer) opens a connection to the Amazon database system by entering the URL `www.amazon.com` into their browser. The Amazon web-server presents a "home page" to the user, which includes forms, menus, and buttons enabling the user to express what it is they want to do. For example, the user may set a menu to `Books` and enter into a form the title of the book they are interested in. The client web-browser transmits this information to the Amazon web-server, and that web-server must negotiate with the next tier — the application tier — to fulfill the client's request.

1.2 The Application Tier

The job of the application tier is to turn data, from the database, into a response to the request that it receives from the web-server. Each web-server process can invoke one or more application-tier processes to handle the request; these processes can be on one machine or many, and they may be on the same or different machines from the web-server processes.

The actions performed by the application tier are often referred to as the *business logic* of the organization operating the database. That is, one designs the application tier by reasoning out what the response to a request by the potential customer should be, and then implementing that strategy.

In the case of our example of a book at Amazon.com, this response would be the elements of the page that Amazon displays about a book. That data includes the title, author, price, and several other pieces of information about the book. It also includes links to more information, such as reviews, alternative sellers of the book, and similar books.

In a simple system, the application tier may issue database queries directly to the database tier, and assemble the results of those queries, perhaps in an HTML page. In a more complex system, there can be several subtiers to the application tier, and each may have its own processes. A common architecture is to have a subtier that supports "objects." These objects can contain data such as the title and price of a book in a "book object." Data for this object is obtained by a database query. The object may also have methods that can be invoked by the application-tier processes, and these methods may in turn cause additional queries to be issued to the database when and if they are invoked.

Another subtier may be present to support *database integration*. That is, there may be several quite independent databases that support operations, and it may not be possible to issue queries involving data from more than one database at a time. The results of queries to different sources may need to be combined at the integration subtier. To make integration more complex, the databases may not be compatable in a number of important ways. We shall

examine the technology of information integration elsewhere. However, for the moment, consider the following hypothetical example.

Example 1: The Amazon database containing information about a book may have a price in dollars. But the customer is in Europe, and their account information is in another database, located in Europe, with billing information in Euros. The integration subtier needs to know that there is a difference in currencies, when it gets a price from the books database and uses that price to enter data into a bill that is displayed to the customer. □

1.3 The Database Tier

Like the other tiers, there can be many processes in the database tier, and the processes can be distributed over many machines, or all be together on one. The database tier executes queries that are requested from the application tier, and may also provide some buffering of data. For example, a query that produces many tuples may be fed one-at-a-time to the requesting process of the application tier.

Since creating connections to the database takes significant time, we normally keep a large number of connections open and allow application processes to share these connections. Each application process must return the connection to the state in which it was found, to avoid unexpected interactions between application processes.

The balance of this chapter is about how we implement a database tier. Especially, we need to learn:

1. How do we enable a database to interact with "ordinary" programs that are written in a conventional language such as C or Java?

2. How do we deal with the differences in data-types supported by SQL and conventional languages? In particular, relations are the results of queries, and these are not directly supported by conventional languages.

3. How do we manage connections to a database when these connections are shared between many short-lived processes?

2 The SQL Environment

In this section we shall take the broadest possible view of a DBMS and the databases and programs it supports. We shall see how databases are defined and organized into clusters, catalogs, and schemas. We shall also see how programs are linked with the data they need to manipulate. Many of the details depend on the particular implementation, so we shall concentrate on the general ideas that are contained in the SQL standard. Sections 5, 6, and 7 illustrate how these high-level concepts appear in a "call-level interface," which requires the programmer to make explicit connections to databases.

2.1 Environments

A *SQL environment* is the framework under which data may exist and SQL operations on data may be executed. In practice, we should think of a SQL environment as a DBMS running at some installation. For example, ABC company buys a license for the Megatron 2010 DBMS to run on a collection of ABC's machines. The system running on these machines constitutes a SQL environment.

All the database elements we have discussed — tables, views, triggers, and so on — are defined within a SQL environment. These elements are organized into a hierarchy of structures, each of which plays a distinct role in the organization. The structures defined by the SQL standard are indicated in Fig. 2.

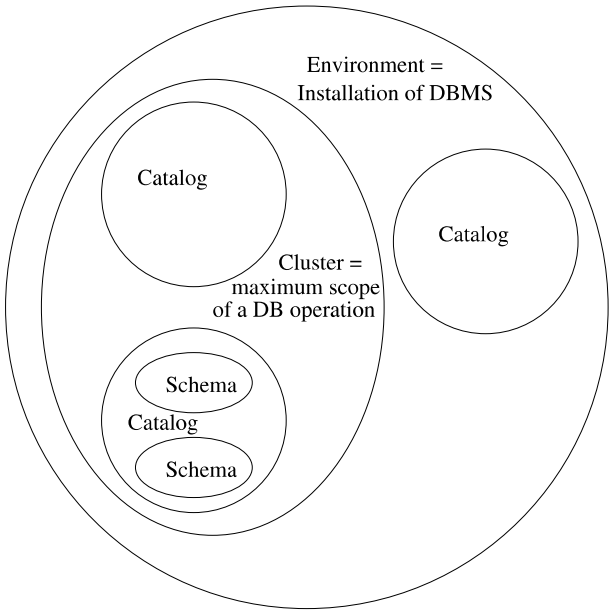

Figure 2: Organization of database elements within the environment

Briefly, the organization consists of the following structures:

1. *Schemas.* These are collections of tables, views, assertions, triggers, and some other types of information (see the box on "More Schema Elements" in Section 2.2). Schemas are the basic units of organization, close to what we might think of as a "database," but in fact somewhat less than a database as we shall see in point (3) below.

2. *Catalogs.* These are collections of schemas. They are the basic unit for supporting unique, accessible terminology. Each catalog has one or more schemas; the names of schemas within a catalog must be unique, and

each catalog contains a special schema called `INFORMATION_SCHEMA` that contains information about all the schemas in the catalog.

3. *Clusters.* These are collections of catalogs. Each user has an associated cluster: the set of all catalogs accessible to the user. A cluster is the maximum scope over which a query can be issued, so in a sense, a cluster is "the database" as seen by a particular user.

2.2 Schemas

The simplest form of schema declaration is:

`CREATE SCHEMA` <schema name> <element declarations>

The element declarations are of the forms discussed in various places, such as Section 4.1

Example 2: We could declare a schema that includes the five relations about movies that we have been using in our running example, plus some of the other elements we have introduced, such as views. Figure 3 sketches the form of such a declaration. □

```
CREATE SCHEMA MovieSchema
  CREATE TABLE MovieStar
    Create-table statements for the four other tables
  CREATE VIEW MovieProd
    Other view declarations
  CREATE ASSERTION RichPres
```

Figure 3: Declaring a schema

It is not necessary to declare the schema all at once. One can modify or add to the "current" schema using the appropriate `CREATE`, `DROP`, or `ALTER` statement, e.g., `CREATE TABLE` followed by the declaration of a new table for the schema. We change the "current" schema with a `SET SCHEMA` statement. For example,

```
SET SCHEMA MovieSchema;
```

makes the schema described in Fig. 3 the current schema. Then, any declarations of schema elements are added to that schema, and any `DROP` or `ALTER` statements refer to elements already in that schema.

More Schema Elements

Some schema elements that we have not already mentioned, but that occasionally are useful are:

- *Domains*: These are sets of values or simple data types. They are little used today, because object-relational DBMS's provide more powerful type-creation mechanisms.
- *Character sets*: These are sets of symbols and methods for encoding them. ASCII and Unicode are common options.
- *Collations*: A collation specifies which characters are "less than" which others. For example, we might use the ordering implied by the ASCII code, or we might treat lower-case and capital letters the same and not compare anything that isn't a letter.
- *Grant statements*: These concern who has access to schema elements.
- *Stored Procedures*: These are executable code; see Section 4.

2.3 Catalogs

Just as schema elements like tables are created within a schema, schemas are created and modified within a catalog. In principle, we would expect the process of creating and populating catalogs to be analogous to the process of creating and populating schemas. Unfortunately, SQL does not define a standard way to do so, such as a statement

`CREATE CATALOG` <catalog name>

followed by a list of schemas belonging to that catalog and the declarations of those schemas.

However, SQL does stipulate a statement

`SET CATALOG` <catalog name>

This statement allows us to set the "current" catalog, so new schemas will go into that catalog and schema modifications will refer to schemas in that catalog should there be a name ambiguity.

2.4 Clients and Servers in the SQL Environment

A SQL environment is more than a collection of catalogs and schemas. It contains elements whose purpose is to support operations on the database or

Complete Names for Schema Elements

Formally, the name for a schema element such as a table is its catalog name, its schema name, and its own name, connected by dots in that order. Thus, the table `Movies` in the schema `MovieSchema` in the catalog `MovieCatalog` can be referred to as

```
MovieCatalog.MovieSchema.Movies
```

If the catalog is the default or current catalog, then we can omit that component of the name. If the schema is also the default or current schema, then that part too can be omitted, and we are left with the element's own name, as is usual. However, we have the option to use the full name if we need to access something outside the current schema or catalog.

databases represented by those catalogs and schemas. According to the SQL standard, within a SQL environment are two special kinds of processes: SQL clients and SQL servers.

In terms of Fig. 1, a "SQL server" plays the role what we called a "database server there. A "SQL client" is like the application servers from that figure. The SQL standard does not define processes analogous to what we called "Web servers" or "clients" in Fig. 1.

2.5 Connections

If we wish to run some program involving SQL at a host where a SQL client exists, then we may open a connection between the client and server by executing a SQL statement

```
CONNECT TO <server name> AS <connection name>
    AUTHORIZATION <name and password>
```

The server name is something that depends on the installation. The word `DEFAULT` can substitute for a name and will connect the user to whatever SQL server the installation treats as the "default server." We have shown an authorization clause followed by the user's name and password. The latter is the typical method by which a user would be identified to the server, although other strings following `AUTHORIZATION` might be used.

The connection name can be used to refer to the connection later on. The reason we might have to refer to the connection is that SQL allows several connections to be opened by the user, but only one can be active at any time. To switch among connections, we can make `conn1` become the active connection by the statement:

```
SET CONNECTION conn1;
```

Whatever connection was currently active becomes *dormant* until it is reactivated with another `SET CONNECTION` statement that mentions it explicitly.

We also use the name when we drop the connection. We can drop connection `conn1` by

```
DISCONNECT conn1;
```

Now, `conn1` is terminated; it is not dormant and cannot be reactivated.

However, if we shall never need to refer to the connection being created, then `AS` and the connection name may be omitted from the `CONNECT TO` statement. It is also permitted to skip the connection statements altogether. If we simply execute SQL statements at a host with a SQL client, then a default connection will be established on our behalf.

2.6 Sessions

The SQL operations that are performed while a connection is active form a *session*. The session lasts as long as the connection that created it. For example, when a connection is made dormant, its session also becomes dormant, and reactivation of the connection by a `SET CONNECTION` statement also makes the session active. Thus, we have shown the session and connection as two aspects of the link between client and server in Fig. 4.

Each session has a current catalog and a current schema within that catalog. These may be set with statements `SET SCHEMA` and `SET CATALOG`, as discussed in Sections 2.2 and 2.3. There is also an authorized user for every session.

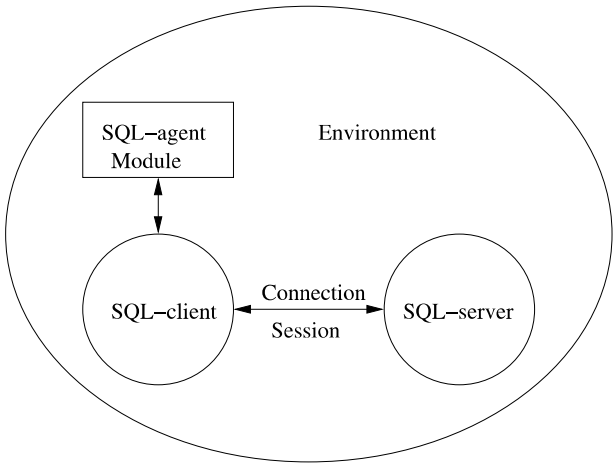

Figure 4: The SQL client-server interactions

The Languages of the SQL Standard

Implementations conforming to the SQL standard are required to support at least one of the following seven host languages: ADA, C, Cobol, Fortran, M (formerly called Mumps, and used primarily in the medical community), Pascal, and PL/I. We shall use C in our examples.

2.7 Modules

A *module* is the SQL term for an application program. The SQL standard suggests that there are three kinds of modules, but insists only that a SQL implementation offer the user at least one of these types.

1. *Generic SQL Interface.* The user may type SQL statements that are executed by a SQL server. In this mode, each query or other statement is a module by itself. It is this mode that we imagined for most of our examples in this book, although in practice it is rarely used.

2. *Embedded SQL.* This style will be discussed in Section 3. Typically, a preprocessor turns the embedded SQL statements into suitable function or procedure calls to the SQL system. The compiled host-language program, including these function calls, is a module.

3. *True Modules.* The most general style of modules envisioned by SQL is a collection of stored functions or procedures, some of which are host-language code and some of which are SQL statements. They communicate among themselves by passing parameters and perhaps via shared variables. PSM modules (Section 4) are an example of this type of module.

An execution of a module is called a *SQL agent.* In Fig. 4 we have shown both a module and an SQL agent, as one unit, calling upon a SQL client to establish a connection. However, we should remember that the distinction between a module and an SQL agent is analogous to the distinction between a program and a process; the first is code, the second is an execution of that code.

3 The SQL/Host-Language Interface

To this point, we have used the *generic SQL interface* in our examples. That is, we have assumed there is a SQL interpreter, which accepts and executes the sorts of SQL queries and commands that we have learned. Although provided as an option by almost all DBMS's, this mode of operation is actually rare. In real systems, such as those described in Section 1, there is a program in some

conventional *host* language such as C, but some of the steps in this program are actually SQL statements.

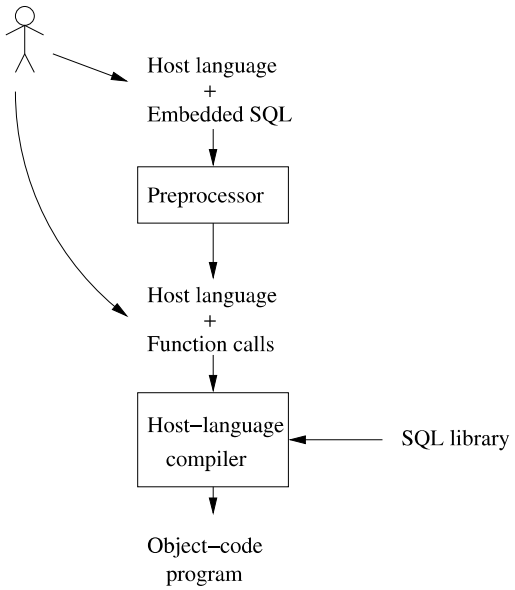

Figure 5: Processing programs with SQL statements embedded

A sketch of a typical programming system that involves SQL statements is in Fig. 5. There, we see the programmer writing programs in a host language, but with some special "embedded" SQL statements. There are two ways this embedding could take place.

1. *Call-Level Interface.* A library is provided, and the embedding of SQL in the host language is really calls to functions or methods in this library. SQL statements are usually string arguments of these methods. This approach, often referred to as a call-level interface or CLI, is discussed in Section 5 and is represented by the curved arrow in Fig. 5 from the user directly to the host language.

2. *Directly Embedded SQL.* The entire host-language program, with embedded SQL statements, is sent to a preprocessor, which changes the embedded SQL statements into something that makes sense in the host language. Typically, the SQL statements are replaced by calls to library functions or methods, so the difference between a CLI and direct embedding of SQL is more a matter of "look and feel" than of substance. The preprocessed host-language program is then compiled in the usual manner and operates on the database through execution of the library calls.

In this section, we shall learn the SQL standard for direct embedding in a host language — C in particular. We are also introduced to a number of concepts, such as cursors, that appear in all, or almost all, systems for embedding SQL.

3.1 The Impedance Mismatch Problem

The basic problem of connecting SQL statements with those of a conventional programming language is *impedance mismatch*: the fact that the data model of SQL differs so much from the models of other languages. As we know, SQL uses the relational data model at its core. However, C and similar languages use a data model with integers, reals, arithmetic, characters, pointers, record structures, arrays, and so on. Sets are not represented directly in C or these other languages, while SQL does not use pointers, loops and branches, or many other common programming-language constructs. As a result, passing data between SQL and other languages is not straightforward, and a mechanism must be devised to allow the development of programs that use both SQL and another language.

One might first suppose that it is preferable to use a single language. Either do all computation in SQL or forget SQL and do all computation in a conventional language. However, we can dispense with the idea of omitting SQL when there are database operations involved. SQL systems greatly aid the programmer in writing database operations that can be executed efficiently, yet that can be expressed at a very high level. SQL takes from the programmer's shoulders the need to understand how data is organized in storage or how to exploit that storage structure to operate efficiently on the database.

On the other hand, there are many important things that SQL cannot do at all. For example, one cannot write a SQL query to compute n factorial, something that is an easy exercise in C or similar languages.[1] As another example, SQL cannot format its output directly into a convenient form such as a graphic. Thus, real database programming requires both SQL and a host language.

3.2 Connecting SQL to the Host Language

When we wish to use a SQL statement within a host-language program, we warn the preprocessor that SQL code is coming with the keywords `EXEC SQL` in front of the statement. We transfer information between the database, which is accessed only by SQL statements, and the host-language program through *shared variables*, which are allowed to appear in both host-language statements

[1] We should be careful here. There are extensions to the basic SQL language, such as recursive SQL or SQL/PSM discussed in Section 4, that do offer "Turing completeness" — the ability to compute anything that can be computed in any other programming language. However, these extensions were never intended for general-purpose calculation, and we do not regard them as general-purpose languages.

and SQL statements. Shared variables are prefixed by a colon within a SQL statement, but they appear without the colon in host-language statements.

A special variable, called `SQLSTATE` in the SQL standard, serves to connect the host-language program with the SQL execution system. The type of `SQLSTATE` is an array of five characters. Each time a function of the SQL library is called, a code is put in the variable `SQLSTATE` that indicates any problems found during that call. The SQL standard also specifies a large number of five-character codes and their meanings.

For example, `'00000'` (five zeroes) indicates that no error condition occurred, and `'02000'` indicates that a tuple requested as part of the answer to a SQL query could not be found. The latter code is very important, since it allows us to create a loop in the host-language program that examines tuples from some relation one-at-a-time and to break the loop after the last tuple has been examined.

3.3 The DECLARE Section

To declare shared variables, we place their declarations between two embedded SQL statements:

```
EXEC SQL BEGIN DECLARE SECTION;
    ...
EXEC SQL END DECLARE SECTION;
```

What appears between them is called the *declare section*. The form of variable declarations in the declare section is whatever the host language requires. It only makes sense to declare variables to have types that both the host language and SQL can deal with, such as integers, reals, and character strings or arrays.

Example 3: The following statements might appear in a C function that updates the `Studio` relation:

```
EXEC SQL BEGIN DECLARE SECTION;
    char studioName[50], studioAddr[256];
    char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
```

The first and last statements are the required beginning and end of the declare section. In the middle is a statement declaring two shared variables, `studioName` and `studioAddr`. These are both character arrays and, as we shall see, they can be used to hold a name and address of a studio that are made into a tuple and inserted into the `Studio` relation. The third statement declares `SQLSTATE` to be a six-character array.[2] □

[2] We shall use six characters for the five-character value of `SQLSTATE` because in programs to follow we want to use the C function `strcmp` to test whether `SQLSTATE` has a certain value. Since `strcmp` expects strings to be terminated by `'\0'`, we need a sixth character for this endmarker. The sixth character must be set initially to `'\0'`, but we shall not show this assignment in programs to follow.

3.4 Using Shared Variables

A shared variable can be used in SQL statements in places where we expect or allow a constant. Recall that shared variables are preceded by a colon when so used. Here is an example in which we use the variables of Example 3 as components of a tuple to be inserted into relation `Studio`.

Example 4: In Fig. 6 is a sketch of a C function `getStudio` that prompts the user for the name and address of a studio, reads the responses, and inserts the appropriate tuple into `Studio`. Lines (1) through (4) are the declarations from Example 3. We omit the C code that prints requests and scans entered text to fill the two arrays `studioName` and `studioAddr`.

```
    void getStudio() {

1)      EXEC SQL BEGIN DECLARE SECTION;
2)          char studioName[50], studioAddr[256];
3)          char SQLSTATE[6];
4)      EXEC SQL END DECLARE SECTION;

        /* print request that studio name and address
           be entered and read response into variables
           studioName and studioAddr */

5)      EXEC SQL INSERT INTO Studio(name, address)
6)               VALUES (:studioName, :studioAddr);
    }
```

Figure 6: Using shared variables to insert a new studio

Then, in lines (5) and (6) is an embedded SQL `INSERT` statement. This statement is preceded by the keywords `EXEC SQL` to indicate that it is indeed an embedded SQL statement rather than ungrammatical C code. The values inserted by lines (5) and (6) are not explicit constants, as they were in all previous examples; rather, the values appearing in line (6) are shared variables whose current values become components of the inserted tuple. □

Any SQL statement that does not return a result (i.e., is not a query) can be embedded in a host-language program by preceding it with `EXEC SQL`. Examples of embeddable SQL statements include insert-, delete-, and update-statements and those statements that create, modify, or drop schema elements such as tables and views.

However, select-from-where queries are not embeddable directly into a host language, because of the "impedance mismatch." Queries produce bags of tuples as a result, while none of the major host languages support a set or

bag data type directly. Thus, embedded SQL must use one of two mechanisms for connecting the result of queries with a host-language program:

1. *Single-Row* `SELECT` *Statements.* A query that produces a single tuple can have that tuple stored in shared variables, one variable for each component of the tuple.

2. *Cursors.* Queries producing more than one tuple can be executed if we declare a *cursor* for the query. The cursor ranges over all tuples in the answer relation, and each tuple in turn can be fetched into shared variables and processed by the host-language program.

We shall consider each of these mechanisms in turn.

3.5 Single-Row Select Statements

The form of a single-row select is the same as an ordinary select-from-where statement, except that following the `SELECT` clause is the keyword `INTO` and a list of shared variables. These shared variables each are preceded by a colon, as is the case for all shared variables within a SQL statement. If the result of the query is a single tuple, this tuple's components become the values of these variables. If the result is either no tuple or more than one tuple, then no assignment to the shared variables is made, and an appropriate error code is written in the variable `SQLSTATE`.

Example 5: We shall write a C function to read the name of a studio and print the net worth of the studio's president. A sketch of this function is shown in Fig. 7. It begins with a declare section, lines (1) through (5), for the variables we shall need. Next, C statements that we do not show explicitly obtain a studio name from the standard input.

Lines (6) through (9) are the single-row select statement. It is quite similar to queries we have already seen. The two differences are that the value of variable `studioName` is used in place of a constant string in the condition of line (9), and there is an `INTO` clause at line (7) that tells us where to put the result of the query. In this case, we expect a single tuple, and tuples have only one component, that for attribute `netWorth`. The value of this one component of one tuple is placed in the shared variable `presNetWorth`. □

3.6 Cursors

The most versatile way to connect SQL queries to a host language is with a cursor that runs through the tuples of a relation. This relation can be a stored table, or it can be something that is generated by a query. To create and use a cursor, we need the following statements:

1. A cursor declaration, whose simplest form is:

```
    void printNetWorth() {

1)      EXEC SQL BEGIN DECLARE SECTION;
2)          char studioName[50];
3)          int presNetWorth;
4)          char SQLSTATE[6];
5)      EXEC SQL END DECLARE SECTION;

        /* print request that studio name be entered.
           read response into studioName */

6)      EXEC SQL SELECT netWorth
7)               INTO :presNetWorth
8)               FROM Studio, MovieExec
9)               WHERE presC# = cert# AND
                       Studio.name = :studioName;

        /* check that SQLSTATE has all 0's and if so, print
           the value of presNetWorth */
    }
```

Figure 7: A single-row select embedded in a C function

`EXEC SQL DECLARE` <cursor name> `CURSOR FOR` <query>

The query can be either an ordinary select-from-where query or a relation name. The cursor *ranges* over the tuples of the relation produced by the query.

2. A statement `EXEC SQL OPEN`, followed by the cursor name. This statement initializes the cursor to a position where it is ready to retrieve the first tuple of the relation over which the cursor ranges.

3. One or more uses of a *fetch statement*. The purpose of a fetch statement is to get the next tuple of the relation over which the cursor ranges. The fetch statement has the form:

   `EXEC SQL FETCH FROM` <cursor name> `INTO` <list of variables>

   There is one variable in the list for each attribute of the tuple's relation. If there is a tuple available to be fetched, these variables are assigned the values of the corresponding components from that tuple. If the tuples have been exhausted, then no tuple is returned, and the value of `SQLSTATE` is set to `'02000'`, a code that means "no tuple found."

4. The statement `EXEC SQL CLOSE` followed by the name of the cursor. This statement closes the cursor, which now no longer ranges over tuples of the relation. It can, however, be reinitialized by another `OPEN` statement, in which case it ranges anew over the tuples of this relation.

Example 6: Suppose we wish to determine the number of movie executives whose net worths fall into a sequence of bands of exponentially growing size, each band corresponding to a number of digits in the net worth. We shall design a query that retrieves the `netWorth` field of all the `MovieExec` tuples into a shared variable called `worth`. A cursor called `execCursor` will range over all these one-component tuples. Each time a tuple is fetched, we compute the number of digits in the integer `worth` and increment the appropriate element of an array `counts`.

The C function `worthRanges` begins in line (1) of Fig. 8. Line (2) declares some variables used only by the C function, not by the embedded SQL. The array `counts` holds the counts of executives in the various bands, `digits` counts the number of digits in a net worth, and `i` is an index ranging over the elements of array `counts`.

Lines (3) through (6) are a SQL declare section in which shared variable `worth` and the usual `SQLSTATE` are declared. Lines (7) and (8) declare `execCursor` to be a cursor that ranges over the values produced by the query on line (8). This query simply asks for the `netWorth` components of all the tuples in `MovieExec`. This cursor is then opened at line (9). Line (10) completes the initialization by zeroing the elements of array `counts`.

The main work is done by the loop of lines (11) through (16). At line (12) a tuple is fetched into shared variable `worth`. Since tuples produced by the query of line (8) have only one component, we need only one shared variable, although in general there would be as many variables as there are components of the retrieved tuples. Line (13) tests whether the fetch has been successful. Here, we use a macro `NO_MORE_TUPLES`, defined by

```
#define NO_MORE_TUPLES !(strcmp(SQLSTATE,"02000"))
```

Recall that `"02000"` is the `SQLSTATE` code that means no tuple was found. If there are no more tuples, we break out of the loop and go to line (17).

If a tuple has been fetched, then at line (14) we initialize the number of digits in the net worth to 1. Line (15) is a loop that repeatedly divides the net worth by 10 and increments `digits` by 1. When the net worth reaches 0 after division by 10, `digits` holds the correct number of digits in the value of `worth` that was originally retrieved. Finally, line (16) increments the appropriate element of the array `counts` by 1. We assume that the number of digits is no more than 14. However, should there be a net worth with 15 or more digits, line (16) will not increment any element of the `counts` array, since there is no appropriate range; i.e., enormous net worths are thrown away and do not affect the statistics.

Line (17) begins the wrap-up of the function. The cursor is closed, and lines (18) and (19) print the values in the `counts` array. □

```
 1)  void worthRanges() {

 2)      int i, digits, counts[15];
 3)      EXEC SQL BEGIN DECLARE SECTION;
 4)          int worth;
 5)          char SQLSTATE[6];
 6)      EXEC SQL END DECLARE SECTION;
 7)      EXEC SQL DECLARE execCursor CURSOR FOR
 8)          SELECT netWorth FROM MovieExec;

 9)      EXEC SQL OPEN execCursor;
10)      for(i=1; i<15; i++) counts[i] = 0;
11)      while(1) {
12)          EXEC SQL FETCH FROM execCursor INTO :worth;
13)          if(NO_MORE_TUPLES) break;
14)          digits = 1;
15)          while((worth /= 10) > 0) digits++;
16)          if(digits <= 14) counts[digits]++;
         }
17)      EXEC SQL CLOSE execCursor;
18)      for(i=0; i<15; i++)
19)          printf("digits = %d: number of execs = %d\n",
                 i, counts[i]);
     }
```

Figure 8: Grouping executive net worths into exponential bands

3.7 Modifications by Cursor

When a cursor ranges over the tuples of a base table (i.e., a relation that is stored in the database), then one can not only read the current tuple, but one can update or delete the current tuple. The `WHERE` clause may only be `WHERE CURRENT OF` followed by the name of the cursor. Of course it is possible for the host-language program reading the tuple to apply whatever condition it likes to the tuple before deciding whether or not to delete or update it.

Example 7: In Fig. 9 we see a C function that looks at each tuple of `MovieExec` and decides either to delete the tuple or to double the net worth. In lines (3) and (4) we declare variables that correspond to the four attributes of `MovieExec`, as well as the necessary `SQLSTATE`. Then, at line (6), `execCursor` is declared to range over the stored relation `MovieExec` itself.

Lines (8) through (14) are the loop, in which the cursor `execCursor` refers to each tuple of `MovieExec`, in turn. Line (9) fetches the current tuple into

```
 1)  void changeWorth() {

 2)      EXEC SQL BEGIN DECLARE SECTION;
 3)          int certNo, worth;
 4)          char execName[31], execAddr[256], SQLSTATE[6];
 5)      EXEC SQL END DECLARE SECTION;
 6)      EXEC SQL DECLARE execCursor CURSOR FOR MovieExec;

 7)      EXEC SQL OPEN execCursor;
 8)      while(1) {
 9)          EXEC SQL FETCH FROM execCursor INTO :execName,
                 :execAddr, :certNo, :worth;
10)          if(NO_MORE_TUPLES) break;
11)          if (worth < 1000)
12)              EXEC SQL DELETE FROM MovieExec
                          WHERE CURRENT OF execCursor;
13)          else
14)              EXEC SQL UPDATE MovieExec
                         SET netWorth = 2 * netWorth
                         WHERE CURRENT OF execCursor;
         }
15)      EXEC SQL CLOSE execCursor;
     }
```

Figure 9: Modifying executive net worths

the four variables used for this purpose; note that only `worth` is actually used. Line (10) tests whether we have exhausted the tuples of `MovieExec`. We have again used the macro `NO_MORE_TUPLES` for the condition that variable `SQLSTATE` has the "no more tuples" code `"02000"`.

In the test of line (11) we ask if the net worth is under $1000. If so, the tuple is deleted by the `DELETE` statement of line (12). Note that the `WHERE` clause refers to the cursor, so the current tuple of `MovieExec`, the one we just fetched, is deleted from `MovieExec`. If the net worth is at least $1000, then at line (14), the net worth in the same tuple is doubled, instead. □

3.8 Protecting Against Concurrent Updates

Suppose that as we examine the net worths of movie executives using the function `worthRanges` of Fig. 8, some other process is modifying the underlying `MovieExec` relation. What should we do about this possibility? Perhaps nothing. We might be happy with approximate statistics, and we don't care whether or not we count an executive who was in the process of being deleted, for example. Then, we simply accept what tuples we get through the cursor.

However, we may not wish to allow concurrent changes to affect the tuples we see through this cursor. Rather, we may insist on the statistics being taken on the relation as it exists at some point in time. We want the code that runs the cursor through the relation to be serializable with any other operations on the relation. To obtain this guarantee, we may declare the cursor *insensitive* to concurrent changes.

Example 8: We could modify lines (7) and (8) of Fig. 8 to be:

```
7)     EXEC SQL DECLARE execCursor INSENSITIVE CURSOR FOR
8)         SELECT netWorth FROM MovieExec;
```

If `execCursor` is so declared, then the SQL system will guarantee that changes to relation `MovieExec` made between one opening and closing of `execCursor` will not affect the set of tuples fetched. □

There are certain cursors ranging over a relation R about which we may say with certainty that they will not change R. Such a cursor can run simultaneously with an insensitive cursor for R, without risk of changing the relation R that the insensitive cursor sees. If we declare a cursor `FOR READ ONLY`, then the database system can be sure that the underlying relation will not be modified because of access to the relation through this cursor.

Example 9: We could append after line (8) of `worthRanges` in Fig. 8 a line

```
FOR READ ONLY;
```

If so, then any attempt to execute a modification through cursor `execCursor` would cause an error. □

3.9 Dynamic SQL

Our model of SQL embedded in a host language has been that of specific SQL queries and commands within a host-language program. An alternative style of embedded SQL has the statements themselves be computed by the host language. Such statements are not known at compile time, and thus cannot be handled by a SQL preprocessor or a host-language compiler.

An example of such a situation is a program that prompts the user for an SQL query, reads the query, and then executes that query. The generic interface for ad-hoc SQL queries is an example of just such a program. If queries are read and executed at run-time, there is nothing that can be done at compile-time. The query has to be parsed and a suitable way to execute the query found by the SQL system, immediately after the query is read.

The host-language program must instruct the SQL system to take the character string just read, to turn it into an executable SQL statement, and finally to execute that statement. There are two *dynamic SQL* statements that perform these two steps.

1. `EXEC SQL PREPARE` V `FROM` <expression>, where V is a SQL variable. The expression can be any host-language expression whose value is a string; this string is treated as a SQL statement. Presumably, the SQL statement is parsed and a good way to execute it is found by the SQL system, but the statement is not executed. Rather, the plan for executing the SQL statement becomes the value of V.

2. `EXEC SQL EXECUTE` V. This statement causes the SQL statement denoted by variable V to be executed.

Both steps can be combined into one, with the statement:

`EXEC SQL EXECUTE IMMEDIATE` <expression>

The disadvantage of combining these two parts is seen if we prepare a statement once and then execute it many times. With `EXECUTE IMMEDIATE` the cost of preparing the statement is paid each time the statement is executed, rather than paid only once, when we prepare it.

Example 10: In Fig. 10 is a sketch of a C program that reads text from standard input into a variable `query`, prepares it, and executes it. The SQL variable `SQLquery` holds the prepared query. Since the query is only executed once, the line:

```
EXEC SQL EXECUTE IMMEDIATE :query;
```

could replace lines (6) and (7) of Fig. 10. □

```
1)  void readQuery() {

2)      EXEC SQL BEGIN DECLARE SECTION;
3)          char *query;
4)      EXEC SQL END DECLARE SECTION;

5)      /* prompt user for a query, allocate space (e.g.,
           use malloc) and make shared variable :query point
           to the first character of the query */
6)      EXEC SQL PREPARE SQLquery FROM :query;
7)      EXEC SQL EXECUTE SQLquery;
    }
```

Figure 10: Preparing and executing a dynamic SQL query

3.10 Exercises for Section 3

Exercise 3.1: Write the following embedded SQL queries, based on this database schema:

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

You may use any host language with which you are familiar, and details of host-language programming may be replaced by clear comments if you wish.

a) Ask the user for a price and find the PC whose price is closest to the desired price. Print the maker, model number, and speed of the PC.

b) Ask the user for minimum values of the speed, RAM, hard-disk size, and screen size that they will accept. Find all the laptops that satisfy these requirements. Print their specifications (all attributes of `Laptop`) and their manufacturer.

! c) Ask the user for a manufacturer. Print the specifications of all products by that manufacturer. That is, print the model number, product-type, and all the attributes of whichever relation is appropriate for that type.

!! d) Ask the user for a "budget" (total price of a PC and printer), and a minimum speed of the PC. Find the cheapest "system" (PC plus printer) that is within the budget and minimum speed, but make the printer a color printer if possible. Print the model numbers for the chosen system.

e) Ask the user for a manufacturer, model number, speed, RAM, hard-disk size, and price of a new PC. Check that there is no PC with that model number. Print a warning if so, and otherwise insert the information into tables `Product` and `PC`.

Exercise 3.2: Write the following embedded SQL queries, based on this database schema:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

a) The firepower of a ship is roughly proportional to the number of guns times the cube of the bore of the guns. Find the class with the largest firepower.

! b) Ask the user for the name of a battle. Find the countries of the ships involved in the battle. Print the country with the most ships sunk and the country with the most ships damaged.

c) Ask the user for the name of a class and the other information required for a tuple of table `Classes`. Then ask for a list of the names of the ships of that class and their dates launched. However, the user need not give the first name, which will be the name of the class. Insert the information gathered into `Classes` and `Ships`.

! d) Examine the `Battles`, `Outcomes`, and `Ships` relations for ships that were in battle before they were launched. Prompt the user when there is an error found, offering the option to change the date of launch or the date of the battle. Make whichever change is requested.

4 Stored Procedures

In this section, we introduce you to *Persistent, Stored Modules* (SQL/PSM, or just PSM). PSM is part of the latest revision to the SQL standard, called SQL:2003. It allows us to write procedures in a simple, general-purpose language and to store them in the database, as part of the schema. We can then use these procedures in SQL queries and other statements to perform computations that cannot be done with SQL alone. Each commercial DBMS offers its own extension of PSM. In this book, we shall describe the SQL/PSM standard, which captures the major ideas of these facilities, and which should help you understand the language associated with any particular system. References to PSM extensions provided with several major commercial systems are in the bibliographic notes.

4.1 Creating PSM Functions and Procedures

In PSM, you define *modules*, which are collections of function and procedure definitions, temporary relation declarations, and several other optional declarations. The major elements of a procedure declaration are:

```
CREATE PROCEDURE <name> (<parameters>)
    <local declarations>
    <procedure body>;
```

This form should be familiar from a number of programming languages; it consists of a procedure name, a parenthesized list of parameters, some optional local-variable declarations, and the executable body of code that defines the procedure. A function is defined in almost the same way, except that the keyword `FUNCTION` is used, and there is a return-value type that must be specified. That is, the elements of a function definition are:

```
CREATE FUNCTION <name> (<parameters>) RETURNS <type>
   <local declarations>
   <function body>;
```

The parameters of a PSM procedure are mode-name-type triples. That is, the parameter name is not only followed by its declared type, as usual in programming languages, but it is preceded by a "mode," which is either `IN`, `OUT`, or `INOUT`. These three keywords indicate that the parameter is input-only, output-only, or both input and output, respectively. `IN` is the default, and can be omitted.

Function parameters, on the other hand, may only be of mode `IN`. That is, PSM forbids side-effects in functions, so the only way to obtain information from a function is through its return-value. We shall not specify the `IN` mode for function parameters, although we do so in procedure definitions.

Example 11: While we have not yet learned the variety of statements that can appear in procedure and function bodies, one kind should not surprise us: an SQL statement. The limitation on these statements is the same as for embedded SQL, as we introduced in Section 3.4: only single-row-select statements and cursor-based accesses are permitted as queries. In Fig. 11 is a PSM procedure that takes two addresses — an old address and a new address — as parameters and replaces the old address by the new everywhere it appears in `MovieStar`.

```
1)  CREATE PROCEDURE Move(
2)      IN oldAddr VARCHAR(255),
3)      IN newAddr VARCHAR(255)
    )
4)  UPDATE MovieStar
5)  SET address = newAddr
6)  WHERE address = oldAddr;
```

Figure 11: A procedure to change addresses

Line (1) introduces the procedure and its name, `Move`. Lines (2) and (3) declare two input parameters, both of whose types are `VARCHAR(255)`. Lines (4) through (6) are a conventional `UPDATE` statement. However, notice that the parameter names can be used as if they were constants. Unlike host-language variables, which require a colon prefix when used in SQL (see Section 3.2), parameters and other local variables of PSM procedures and functions require no colon. □

4.2 Some Simple Statement Forms in PSM

Let us begin with a potpourri of statement forms that are easy to master.

1. *The call-statement*: The form of a procedure call is:

   CALL <procedure name> (<argument list>);

   That is, the keyword CALL is followed by the name of the procedure and a parenthesized list of arguments, as in most any language. This call can, however, be made from a variety of places:

   i. From a host-language program, in which it might appear as

   ```
   EXEC SQL CALL Foo(:x, 3);
   ```

   for instance.

   ii. As a statement of another PSM function or procedure.

   iii. As a SQL command issued to the generic SQL interface. For example, we can issue a statement such as

   ```
   CALL Foo(1, 3);
   ```

   to such an interface, and have stored procedure Foo executed with its two parameters set equal to 1 and 3, respectively.

   Note that it is not permitted to call a function. You invoke functions in PSM as you do in C: use the function name and suitable arguments as part of an expression.

2. *The return-statement*: Its form is

   RETURN <expression>;

   This statement can only appear in a function. It evaluates the expression and sets the return-value of the function equal to that result. However, at variance with common programming languages, the return-statement of PSM does *not* terminate the function. Rather, control continues with the following statement, and it is possible that the return-value will be changed before the function completes.

3. *Declarations of local variables*: The statement form

   DECLARE <name> <type>;

   declares a variable with the given name to have the given type. This variable is local, and its value is not preserved by the DBMS after a running of the function or procedure. Declarations must precede executable statements in the function or procedure body.

4. *Assignment Statements*: The form of an assignment is:

SET <variable> = <expression>;

Except for the introductory keyword SET, assignment in PSM is quite like assignment in other languages. The expression on the right of the equal-sign is evaluated, and its value becomes the value of the variable on the left. NULL is a permissible expression. The expression may even be a query, as long as it returns a single value.

5. *Statement groups*: We can form a list of statements ended by semicolons and surrounded by keywords BEGIN and END. This construct is treated as a single statement and can appear anywhere a single statement can. In particular, since a procedure or function body is expected to be a single statement, we can put any sequence of statements in the body by surrounding them by BEGIN...END.

6. *Statement labels*: We label a statement by prefixing it with a name (the label) and a colon.

4.3 Branching Statements

For our first complex PSM statement type, let us consider the if-statement. The form is only a little strange; it differs from C or similar languages in that:

1. The statement ends with keywords END IF.

2. If-statements nested within the else-clause are introduced with the single word ELSEIF.

Thus, the general form of an if-statement is as suggested by Fig. 12. The condition is any boolean-valued expression, as can appear in the WHERE clause of SQL statements. Each statement list consists of statements ended by semicolons, but does not need a surrounding BEGIN...END. The final ELSE and its statement(s) are optional; i.e., IF...THEN...END IF alone or with ELSEIF's is acceptable.

Example 12: Let us write a function to take a year y and a studio s, and return a boolean that is TRUE if and only if studio s produced at least one comedy in year y or did not produce any movies at all in that year. The code appears in Fig. 13.

Line (1) introduces the function and includes its arguments. We do not need to specify a mode for the arguments, since that can only be IN for a function. Lines (2) and (3) test for the case where there are no movies at all by studio s in year y, in which case we set the return-value to TRUE at line (4). Note that line (4) does not cause the function to return. Technically, it is the flow of control dictated by the if-statements that causes control to jump from line (4) to line (9), where the function completes and returns.

```
IF <condition> /tt THEN
   <statement list>
ELSEIF <condition> /tt THEN
   <statement list>
ELSEIF
     ...
ELSE
  <statement list>
END IF;
```

Figure 12: The form of an if-statement

```
1)  CREATE FUNCTION BandW(y INT, s CHAR(15)) RETURNS BOOLEAN

2)  IF NOT EXISTS(
3)      SELECT * FROM Movies WHERE year = y AND
            studioName = s)
4)  THEN RETURN TRUE;
5)  ELSEIF 1 <=
6)      (SELECT COUNT(*) FROM Movies WHERE year = y AND
            studioName = s AND genre = 'comedy')
7)  THEN RETURN TRUE;
8)  ELSE RETURN FALSE;
9)  END IF;
```

Figure 13: If there are any movies at all, then at least one has to be a comedy

If studio s made movies in year y, then lines (5) and (6) test if at least one of them was a comedy. If so, the return-value is again set to true, this time at line (7). In the remaining case, studio s made movies but only in color, so we set the return-value to `FALSE` at line (8). □

4.4 Queries in PSM

There are several ways that select-from-where queries are used in PSM.

1. Subqueries can be used in conditions, or in general, any place a subquery is legal in SQL. We saw two examples of subqueries in lines (3) and (6) of Fig. 13, for instance.

2. Queries that return a single value can be used as the right sides of assignment statements.

3. A single-row select statement is a legal statement in PSM. Recall this statement has an `INTO` clause that specifies variables into which the

components of the single returned tuple are placed. These variables could be local variables or parameters of a PSM procedure. The general form was discussed in the context of embedded SQL in Section 3.5.

4. We can declare and use a cursor, essentially as it was described in Section 3.6 for embedded SQL. The declaration of the cursor, `OPEN`, `FETCH`, and `CLOSE` statements are all as described there, with the exceptions that:
   (a) No `EXEC SQL` appears in the statements, and
   (b) The variables do not use a colon prefix.

```
CREATE PROCEDURE SomeProc(IN studioName CHAR(15))

DECLARE presNetWorth INTEGER;

SELECT netWorth
INTO presNetWorth
FROM Studio, MovieExec
WHERE presC# = cert# AND Studio.name = studioName;
    ...
```

Figure 14: A single-row select in PSM

Example 13: In Fig. 14 is the single-row select of Fig. 7, redone for PSM and placed in the context of a hypothetical procedure definition. Note that, because the single-row select returns a one-component tuple, we could also get the same effect from an assignment statement, as:

```
SET presNetWorth = (SELECT netWorth
    FROM Studio, MovieExec
    WHERE presC# = cert# AND Studio.name = studioName);
```

We shall defer examples of cursor use until we learn the PSM loop statements in the next section. □

4.5 Loops in PSM

The basic loop construct in PSM is:

```
LOOP
   <statement list>
END LOOP;
```

One often labels the `LOOP` statement, so it is possible to break out of the loop, using a statement:

`LEAVE` <loop label>`;`

In the common case that the loop involves the fetching of tuples via a cursor, we often wish to leave the loop when there are no more tuples. It is useful to declare a *condition* name for the `SQLSTATE` value that indicates no tuple found (`'02000'`, recall); we do so with:

```
DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
```

More generally, we can declare a condition with any desired name corresponding to any `SQLSTATE` value by

`DECLARE` <name> `CONDITION FOR SQLSTATE` <value>`;`

We are now ready to take up an example that ties together cursor operations and loops in PSM.

Example 14: Figure 15 shows a PSM procedure that takes a studio name s as an input argument and produces in output arguments `mean` and `variance` the mean and variance of the lengths of all the movies owned by studio s. Lines (1) through (4) declare the procedure and its parameters.

Lines (5) through (8) are local declarations. We define `Not_Found` to be the name of the condition that means a `FETCH` failed to return a tuple at line (5). Then, at line (6), the cursor `MovieCursor` is defined to return the set of the lengths of the movies by studio s. Lines (7) and (8) declare two local variables that we'll need. Integer `newLength` holds the result of a `FETCH`, while `movieCount` counts the number of movies by studio s. We need `movieCount` so that, at the end, we can convert a sum of lengths into an average (mean) of lengths and a sum of squares of the lengths into a variance.

The rest of the lines are the body of the procedure. We shall use `mean` and `variance` as temporary variables, as well as for "returning" the results at the end. In the major loop, `mean` actually holds the sum of the lengths, and `variance` actually holds the sum of the squares of the lengths. Thus, lines (9) through (11) initialize these variables and the count of the movies to 0. Line (12) opens the cursor, and lines (13) through (19) form the loop labeled `movieLoop`.

Line (14) performs a fetch, and at line (15) we check that another tuple was found. If not, we leave the loop. Lines (16) through (18) accumulate values; we add 1 to `movieCount`, add the length to `mean` (which, recall, is really computing the sum of lengths), and we add the square of the length to `variance`.

When all movies by studio s have been seen, we leave the loop, and control passes to line (20). At that line, we turn `mean` into its correct value by dividing the sum of lengths by the count of movies. At line (21), we make `variance` truly hold the variance by dividing the sum of squares of the lengths by the number of movies and subtracting the square of the mean. See Exercise 4.4 for a discussion of why this calculation is correct. Line (22) closes the cursor, and we are done. □

```
 1)  CREATE PROCEDURE MeanVar(
 2)      IN s CHAR(15),
 3)      OUT mean REAL,
 4)      OUT variance REAL
     )
 5)  DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
 6)  DECLARE MovieCursor CURSOR FOR
         SELECT length FROM Movies WHERE studioName = s;
 7)  DECLARE newLength INTEGER;
 8)  DECLARE movieCount INTEGER;

     BEGIN
 9)      SET mean = 0.0;
10)      SET variance = 0.0;
11)      SET movieCount = 0;
12)      OPEN MovieCursor;
13)      movieLoop: LOOP
14)          FETCH FROM MovieCursor INTO newLength;
15)          IF Not_Found THEN LEAVE movieLoop END IF;
16)          SET movieCount = movieCount + 1;
17)          SET mean = mean + newLength;
18)          SET variance = variance + newLength * newLength;
19)      END LOOP;
20)      SET mean = mean/movieCount;
21)      SET variance = variance/movieCount - mean * mean;
22)      CLOSE MovieCursor;
     END;
```

Figure 15: Computing the mean and variance of lengths of movies by one studio

4.6 For-Loops

There is also in PSM a for-loop construct, but it is used only to iterate over a cursor. The form of the statement is shown in Fig. 16. This statement not only declares a cursor, but it handles for us a number of "grubby details": the opening and closing of the cursor, the fetching, and the checking whether there are no more tuples to be fetched. However, since we are not fetching tuples for ourselves, we can not specify the variable(s) into which component(s) of a tuple are placed. Thus, the names used for the attributes in the result of the query are also treated by PSM as local variables of the same type.

Example 15: Let us redo the procedure of Fig. 15 using a for-loop. The code is shown in Fig. 17. Many things have not changed. The declaration of the procedure in lines (1) through (4) of Fig. 17 are the same, as is the

Other Loop Constructs

PSM also allows while- and repeat-loops, which have the expected meaning, as in C. That is, we can create a loop of the form

```
WHILE <condition> DO
   <statement list>
END WHILE;
```

or a loop of the form

```
REPEAT
   <statement list>
UNTIL <condition>
END REPEAT;
```

Incidentally, if we label these loops, or the loop formed by a loop-statement or for-statement, then we can place the label as well after the `END LOOP` or other ender. The advantage of doing so is that it makes clearer where each loop ends, and it allows the PSM compiler to catch some syntactic errors involving the omission of an `END`.

```
FOR <loop name> AS <cursor name> CURSOR FOR
   <query>
DO
   <statement list>
END FOR;
```

Figure 16: The PSM for-statement

declaration of local variable `movieCount` at line (5).

However, we no longer need to declare a cursor in the declaration portion of the procedure, and we do not need to define the condition `Not_Found`. Lines (6) through (8) initialize the variables, as before. Then, in line (9) we see the for-loop, which also defines the cursor `MovieCursor`. Lines (11) through (13) are the body of the loop. Notice that in lines (12) and (13), we refer to the length retrieved via the cursor by the attribute name `length`, rather than by the local variable name `newLength`, which does not exist in this version of the procedure. Lines (15) and (16) compute the correct values for the output variables, exactly as in the earlier version of this procedure. □

```
 1)  CREATE PROCEDURE MeanVar(
 2)      IN s CHAR(15),
 3)      OUT mean REAL,
 4)      OUT variance REAL
     )
 5)  DECLARE movieCount INTEGER;

     BEGIN
 6)      SET mean = 0.0;
 7)      SET variance = 0.0;
 8)      SET movieCount = 0;
 9)      FOR movieLoop AS MovieCursor CURSOR FOR
             SELECT length FROM Movies WHERE studioName = s;
10)      DO
11)          SET movieCount = movieCount + 1;
12)          SET mean = mean + length;
13)          SET variance = variance + length * length;
14)      END FOR;
15)      SET mean = mean/movieCount;
16)      SET variance = variance/movieCount - mean * mean;
     END;
```

Figure 17: Computing the mean and variance of lengths using a for-loop

4.7 Exceptions in PSM

A SQL system indicates error conditions by setting a nonzero sequence of digits in the five-character string `SQLSTATE`. We have seen one example of these codes: `'02000'` for "no tuple found." For another example, `'21000'` indicates that a single-row select has returned more than one row.

PSM allows us to declare a piece of code, called an *exception handler*, that is invoked whenever one of a list of these error codes appears in `SQLSTATE` during the execution of a statement or list of statements. Each exception handler is associated with a block of code, delineated by `BEGIN...END`. The handler appears within this block, and it applies only to statements within the block.

The components of the handler are:

1. A list of exception conditions that invoke the handler when raised.
2. Code to be executed when one of the associated exceptions is raised.
3. An indication of where to go after the handler has finished its work.

The form of a handler declaration is:

`DECLARE` <where to go next> `HANDLER FOR` <condition list>
  <statement>

Why Do We Need Names in For-Loops?

Notice that `movieLoop` and `MovieCursor`, although declared at line (9) of Fig. 17, are never used in that procedure. Nonetheless, we have to invent names, both for the for-loop itself and for the cursor over which it iterates. The reason is that the PSM interpreter will translate the for-loop into a conventional loop, much like the code of Fig. 15, and in this code, there is a need for both names.

The choices for "where to go" are:

a) `CONTINUE`, which means that after executing the statement in the handler declaration, we execute the statement after the one that raised the exception.

b) `EXIT`, which means that after executing the handler's statement, control leaves the `BEGIN...END` block in which the handler is declared. The statement after this block is executed next.

c) `UNDO`, which is the same as `EXIT`, except that any changes to the database or local variables that were made by the statements of the block executed so far are undone. That is, the block is a transaction, which is aborted by the exception.

The "condition list" is a comma-separated list of conditions, which are either declared conditions, like `Not_Found` in line (5) of Fig. 15, or expressions of the form `SQLSTATE` and a five-character string.

Example 16: Let us write a PSM function that takes a movie title as argument and returns the year of the movie. If there is no movie of that title or more than one movie of that title, then `NULL` must be returned. The code is shown in Fig. 18.

Lines (2) and (3) declare symbolic conditions; we do not have to make these definitions, and could as well have used the SQL states for which they stand in line (4). Lines (4), (5), and (6) are a block, in which we first declare a handler for the two conditions in which either zero tuples are returned, or more than one tuple is returned. The action of the handler, on line (5), is simply to set the return-value to `NULL`.

Line (6) is the statement that does the work of the function `GetYear`. It is a `SELECT` statement that is expected to return exactly one integer, since that is what the function `GetYear` returns. If there is exactly one movie with title t (the input parameter of the function), then this value will be returned. However, if an exception is raised at line (6), either because there is no movie with title t or several movies with that title, then the handler is invoked, and `NULL` instead

```
1)  CREATE FUNCTION GetYear(t VARCHAR(255)) RETURNS INTEGER

2)  DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
3)  DECLARE Too_Many CONDITION FOR SQLSTATE '21000';

    BEGIN
4)      DECLARE EXIT HANDLER FOR Not_Found, Too_Many
5)          RETURN NULL;
6)      RETURN (SELECT year FROM Movies WHERE title = t);
    END;
```

Figure 18: Handling exceptions in which a single-row select returns other than one tuple

becomes the return-value. Also, since the handler is an `EXIT` handler, control next passes to the point after the `END`. Since that point is the end of the function, `GetYear` returns at that time, with the return-value `NULL`. □

4.8 Using PSM Functions and Procedures

As we mentioned in Section 4.2, we can call a PSM procedure anywhere SQL statements can appear, e.g., as embedded SQL, from PSM code itself, or from SQL issued to the generic interface. We invoke a procedure by preceding it by the keyword `CALL`. In addition, a PSM function can be used as part of an expression, e.g., in a `WHERE` clause. Here is an example of how a function can be used within an expression.

Example 17: Suppose that our schema includes a module with the function `GetYear` of Fig. 18. Imagine that we are sitting at the generic interface, and we want to enter the fact that Denzel Washington was a star of *Remember the Titans*. However, we forget the year in which that movie was made. As long as there was only one movie of that name, and it is in the `Movies` relation, we don't have to look it up in a preliminary query. Rather, we can issue to the generic SQL interface the following insertion:

```
INSERT INTO StarsIn(movieTitle, movieYear, starName)
VALUES('Remember the Titans', GetYear('Remember the Titans'),
    'Denzel Washington');
```

Since `GetYear` returns `NULL` if there is not a unique movie by the name of *Remember the Titans*, it is possible that this insertion will have `NULL` in the middle component. □

4.9 Exercises for Section 4

Exercise 4.1: Using our running movie database:

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

write PSM procedures or functions to perform the following tasks:

a) Given the name of a movie studio, produce the net worth of its president.

b) Given a name and address, return 1 if the person is a movie star but not an executive, 2 if the person is an executive but not a star, 3 if both, and 4 if neither.

! c) Given a studio name, assign to output parameters the titles of the two longest movies by that studio. Assign `NULL` to one or both parameters if there is no such movie (e.g., if there is only one movie by a studio, there is no "second-longest").

! d) Given a star name, find the earliest (lowest year) movie of more than 120 minutes length in which they appeared. If there is no such movie, return the year 0.

e) Given an address, find the name of the unique star with that address if there is exactly one, and return `NULL` if there is none or more than one.

f) Given the name of a star, delete them from `MovieStar` and delete all their movies from `StarsIn` and `Movies`.

Exercise 4.2: Write the following PSM functions or procedures, based on this database schema:

```
Product(maker, model, type)
PC(model, speed, ram, hd, price)
Laptop(model, speed, ram, hd, screen, price)
Printer(model, color, type, price)
```

a) Take a price as argument and return the model number of the PC whose price is closest.

b) Take a maker and model as arguments, and return the price of whatever type of product that model is.

! c) Take model, speed, ram, hard-disk, and price information as arguments, and insert this information into the relation `PC`. However, if there is already a PC with that model number (tell by assuming that violation of a key constraint on insertion will raise an exception with `SQLSTATE` equal to `'23000'`), then keep adding 1 to the model number until you find a model number that is not already a PC model number.

! d) Given a price, produce the number of PC's, the number of laptops, and the number of printers selling for more than that price.

Exercise 4.3: Write the following PSM functions or procedures, based on this database schema:

```
Classes(class, type, country, numGuns, bore, displacement)
Ships(name, class, launched)
Battles(name, date)
Outcomes(ship, battle, result)
```

a) The firepower of a ship is roughly proportional to the number of guns times the cube of the bore. Given a class, find its firepower.

! b) Given the name of a battle, produce the two countries whose ships were involved in the battle. If there are more or fewer than two countries involved, produce `NULL` for both countries.

c) Take as arguments a new class name, type, country, number of guns, bore, and displacement. Add this information to `Classes` and also add the ship with the class name to `Ships`.

! d) Given a ship name, determine if the ship was in a battle with a date before the ship was launched. If so, set the date of the battle and the date the ship was launched to 0.

! **Exercise 4.4:** In Fig. 15, we used a tricky formula for computing the variance of a sequence of numbers $x_1, x_2, \ldots, x_n$. Recall that the variance is the average square of the deviation of these numbers from their mean. That is, the variance is $\left(\sum_{i=1}^{n}(x_i - \overline{x})^2\right)/n$, where the mean $\overline{x}$ is $\left(\sum_{i=1}^{n} x_i\right)/n$. Prove that the formula for the variance used in Fig. 15, which is

$$\left(\sum_{i=1}^{n}(x_i)^2\right)/n - \left(\left(\sum_{i=1}^{n} x_i\right)/n\right)^2$$

yields the same value.

5 Using a Call-Level Interface

When using a *call-level interface* (CLI), we write ordinary host-language code, and we use a library of functions that allow us to connect to and access a database, passing SQL statements to that database. The differences between this approach and embedded SQL programming are, in one sense, cosmetic, since the preprocessor replaces embedded SQL by calls to library functions much like the functions in the standard SQL/CLI.

We shall give three examples of call-level interfaces. In this section, we cover the standard SQL/CLI, which is an adaptation of ODBC (Open Database Connectivity). We cover JDBC, which is a collection of classes that support database access from Java programs. Then, we explore PHP, which is a way to embed database access in Web pages described by HTML.

5.1 Introduction to SQL/CLI

A program written in C and using SQL/CLI (hereafter, just CLI) will include the header file `sqlcli.h`, from which it gets a large number of functions, type definitions, structures, and symbolic constants. The program is then able to create and deal with four kinds of records (structs, in C):

1. *Environments.* A record of this type is created by the application (client) program in preparation for one or more connections to the database server.

2. *Connections.* One of these records is created to connect the application program to the database. Each connection exists within some environment.

3. *Statements.* An application program can create one or more statement records. Each holds information about a single SQL statement, including an implied cursor if the statement is a query. At different times, the same CLI statement can represent different SQL statements. Every CLI statement exists within some connection.

4. *Descriptions.* These records hold information about either tuples or parameters. The application program or the database server, as appropriate, sets components of description records to indicate the names and types of attributes and/or their values. Each statement has several of these created implicitly, and the user can create more if needed. In our presentation of CLI, description records will generally be invisible.

Each of these records is represented in the application program by a *handle*, which is a pointer to the record. The header file `sqlcli.h` provides types for the handles of environments, connections, statements, and descriptions: `SQLHENV`, `SQLHDBC`, `SQLHSTMT`, and `SQLHDESC`, respectively, although we may think of them as pointers or integers. We shall use these types and also some other defined types with obvious interpretations, such as `SQL_CHAR` and `SQL_INTEGER`, that are provided in `sqlcli.h`.

We shall not go into detail about how descriptions are set and used. However, (handles for) the other three types of records are created by the use of a function

`SQLAllocHandle(`*hType, hIn, hOut*`)`

Here, the three arguments are:

1. *hType* is the type of handle desired. Use `SQL_HANDLE_ENV` for a new environment, `SQL_HANDLE_DBC` for a new connection, or `SQL_HANDLE_STMT` for a new statement.

2. *hIn* is the handle of the higher-level element in which the newly allocated element lives. This parameter is `SQL_NULL_HANDLE` if you want an environment; the latter name is a defined constant telling `SQLAllocHandle` that there is no relevant value here. If you want a connection handle, then *hIn* is the handle of the environment within which the connection will exist, and if you want a statement handle, then *hIn* is the handle of the connection within which the statement will exist.

3. *hOut* is the address of the handle that is created by `SQLAllocHandle`.

`SQLAllocHandle` also returns a value of type `SQLRETURN` (an integer). This value is 0 if no errors occurred, and there are certain nonzero values returned in the case of errors.

Example 18: Let us see how the function `worthRanges` of Fig. 8, which we used as an example of embedded SQL, would begin in CLI. Recall this function examines all the tuples of `MovieExec` and breaks their net worths into ranges. The initial steps are shown in Fig. 19.

```
 1)  #include sqlcli.h
 2)  SQLHENV myEnv;
 3)  SQLHDBC myCon;
 4)  SQLHSTMT execStat;
 5)  SQLRETURN errorCode1, errorCode2, errorCode3;

 6)  errorCode1 = SQLAllocHandle(SQL_HANDLE_ENV,
         SQL_NULL_HANDLE, &myEnv);
 7)  if(!errorCode1) {
 8)      errorCode2 = SQLAllocHandle(SQL_HANDLE_DBC,
             myEnv, &myCon);
 9)  if(!errorCode2)
10)      errorCode3 = SQLAllocHandle(SQL_HANDLE_STMT,
             myCon, &execStat); }
```

Figure 19: Declaring and creating an environment, a connection, and a statement

Lines (2) through (4) declare handles for an environment, connection, and statement, respectively; their names are `myEnv`, `myCon`, and `execStat`, respectively. We plan that `execStat` will represent the SQL statement

```
SELECT netWorth FROM MovieExec;
```

much as did the cursor `execCursor` in Fig. 8, but as yet there is no SQL statement associated with `execStat`. Line (5) declares three variables into which function calls can place their response and indicate an error. A value of 0 indicates no error occurred in the call.

Line (6) calls `SQLAllocHandle`, asking for an environment handle (the first argument), providing a null handle in the second argument (because none is needed when we are requesting an environment handle), and providing the address of `myEnv` as the third argument; the generated handle will be placed there. If line (6) is successful, lines (7) and (8) use the environment handle to get a connection handle in `myCon`. Assuming that call is also successful, lines (9) and (10) get a statement handle for `execStat`. □

5.2 Processing Statements

At the end of Fig. 19, a statement record whose handle is `execStat`, has been created. However, there is as yet no SQL statement with which that record is associated. The process of associating and executing SQL statements with statement handles is analogous to the dynamic SQL described in Section 3.9. There, we associated the text of a SQL statement with what we called a "SQL variable," using `PREPARE`, and then executed it using `EXECUTE`.

The situation in CLI is quite analogous, if we think of the "SQL variable" as a statement handle. There is a function

`SQLPrepare(`*sh, st, sl*`)`

that takes:

1. A statement handle *sh*,
2. A pointer to a SQL statement *st*, and
3. A length *sl* for the character string pointed to by *st*. If we don't know the length, a defined constant `SQL_NTS` tells `SQLPrepare` to figure it out from the string itself. Presumably, the string is a "null-terminated string," and it is sufficient for `SQLPrepare` to scan it until encountering the endmarker `'\0'`.

The effect of this function is to arrange that the statement referred to by the handle *sh* now represents the particular SQL statement *st*.

Another function

`SQLExecute(`*sh*`)`

causes the statement to which handle *sh* refers to be executed. For many forms of SQL statement, such as insertions or deletions, the effect of executing this statement on the database is obvious. Less obvious is what happens when the SQL statement referred to by *sh* is a query. As we shall see in Section 5.3, there

is an implicit cursor for this statement that is part of the statement record itself. The statement is in principle executed, so we can imagine that all the answer tuples are sitting somewhere, ready to be accessed. We can fetch tuples one at a time, using the implicit cursor, much as we did with real cursors in Sections 3 and 4.

Example 19: Let us continue with the function `worthRanges` that we began in Fig. 19. The following two function calls associate the query

```
SELECT netWorth FROM MovieExec;
```

with the statement referred to by handle `execStat`:

```
11)  SQLPrepare(execStat, "SELECT netWorth FROM MovieExec",
         SQL_NTS);
12)  SQLExecute(execStat);
```

These lines could appear right after line (10) of Fig. 19. Remember that `SQL_NTS` tells `SQLPrepare` to determine the length of the null-terminated string to which its second argument refers. □

As with dynamic SQL, the prepare and execute steps can be combined into one if we use the function `SQLExecDirect`. An example that combines lines (11) and (12) above is:

```
SQLExecDirect(execStat, "SELECT netWorth FROM MovieExec",
    SQL_NTS);
```

5.3 Fetching Data From a Query Result

The function that corresponds to a `FETCH` command in embedded SQL or PSM is

`SQLFetch(`*sh*`)`

where *sh* is a statement handle. We presume the statement referred to by *sh* has been executed already, or the fetch will cause an error. `SQLFetch`, like all CLI functions, returns a value of type `SQLRETURN` that indicates either success or an error. The return value `SQL_NO_DATA` tells us tuples were left in the query result. As in our previous examples of fetching, this value will be used to get us out of a loop in which we repeatedly fetch new tuples from the result.

However, if we follow the `SQLExecute` of Example 19 by one or more `SQLFetch` calls, where does the tuple appear? The answer is that its components go into one of the description records associated with the statement whose handle appears in the `SQLFetch` call. We can extract the same component at each fetch by binding the component to a host-language variable, before we begin fetching. The function that does this job is:

`SQLBindCol(`*sh, colNo, colType, pVar, varSize, varInfo*`)`

The meanings of these six arguments are:

1. *sh* is the handle of the statement involved.
2. *colNo* is the number of the component (within the tuple) whose value we obtain.
3. *colType* is a code for the type of the variable into which the value of the component is to be placed. Examples of codes provided by `sqlcli.h` are `SQL_CHAR` for character arrays and strings, and `SQL_INTEGER` for integers.
4. *pVar* is a pointer to the variable into which the value is to be placed.
5. *varSize* is the length in bytes of the value of the variable pointed to by *pVar*.
6. *varInfo* is a pointer to an integer that can be used by `SQLBindCol` to provide additional information about the value produced.

Example 20: Let us redo the entire function `worthRanges` from Fig. 8, using CLI calls instead of embedded SQL. We begin as in Fig. 19, but for the sake of succinctness, we skip all error checking except for the test whether `SQLFetch` indicates that no more tuples are present. The code is shown in Fig. 20.

Line (3) declares the same local variables that the embedded-SQL version of the function uses, and lines (4) through (7) declare additional local variables using the types provided in `sqlcli.h`; these are variables that involve SQL in some way. Lines (4) through (6) are as in Fig. 19. New are the declarations on line (7) of `worth` (which corresponds to the shared variable of that name in Fig. 8) and `worthInfo`, which is required by `SQLBindCol`, but not used.

Lines (8) through (10) allocate the needed handles, as in Fig. 19, and lines (11) and (12) prepare and execute the SQL statement, as discussed in Example 19. In line (13), we see the binding of the first (and only) column of the result of this query to the variable `worth`. The first argument is the handle for the statement involved, and the second argument is the column involved, 1 in this case. The third argument is the type of the column, and the fourth argument is a pointer to the place where the value will be placed: the variable `worth`. The fifth argument is the size of that variable, and the final argument points to `worthInfo`, a place for `SQLBindCol` to put additional information (which we do not use here).

The balance of the function resembles closely lines (11) through (19) of Fig. 8. The while-loop begins at line (14) of Fig. 20. Notice that we fetch a tuple and check that we are not out of tuples, all within the condition of the while-loop, on line (14). If there is a tuple, then in lines (15) through (17) we determine the number of digits the integer (which is bound to `worth`) has and increment the appropriate count. After the loop finishes, i.e., all tuples returned

```
 1)  #include sqlcli.h
 2)  void worthRanges() {

 3)      int i, digits, counts[15];
 4)      SQLHENV myEnv;
 5)      SQLHDBC myCon;
 6)      SQLHSTMT execStat;
 7)      SQLINTEGER worth, worthInfo;

 8)      SQLAllocHandle(SQL_HANDLE_ENV,
             SQL_NULL_HANDLE, &myEnv);
 9)      SQLAllocHandle(SQL_HANDLE_DBC, myEnv, &myCon);
10)      SQLAllocHandle(SQL_HANDLE_STMT, myCon, &execStat);
11)      SQLPrepare(execStat,
             "SELECT netWorth FROM MovieExec", SQL_NTS);
12)      SQLExecute(execStat);
13)      SQLBindCol(execStat, 1, SQL_INTEGER, &worth,
             sizeof(worth), &worthInfo);
14)      while(SQLFetch(execStat) != SQL_NO_DATA) {
15)          digits = 1;
16)          while((worth /= 10) > 0) digits++;
17)          if(digits <= 14) counts[digits]++;
         }
18)      for(i=0; i<15; i++)
19)          printf("digits = %d: number of execs = %d\n",
                 i, counts[i]);
     }
```

Figure 20: Grouping executive net worths: CLI version

by the statement execution of line (12) have been examined, the resulting counts are printed out at lines (18) and (19). □

5.4 Passing Parameters to Queries

Embedded SQL gives us the ability to execute a SQL statement, part of which consists of values determined by the current contents of shared variables. There is a similar capability in CLI, but it is rather more complicated. The steps needed are:

1. Use `SQLPrepare` to prepare a statement in which some portions, called *parameters*, are replaced by a question-mark. The ith question-mark represents the ith parameter.

Extracting Components with `SQLGetData`

An alternative to binding a program variable to an output of a query's result relation is to fetch tuples without any binding and then transfer components to program variables as needed. The function to use is `SQLGetData`, and it takes the same arguments as `SQLBindCol`. However, it only copies data once, and it must be used after each fetch in order to have the same effect as initially binding the column to a variable.]

2. Use function `SQLBindParameter` to bind values to the places where the question-marks are found. This function has ten arguments, of which we shall explain only the essentials.

3. Execute the query with these bindings, by calling `SQLExecute`. Note that if we change the values of one or more parameters, we need to call `SQLExecute` again.

The following example will illustrate the process, as well as indicate the important arguments needed by `SQLBindParameter`.

Example 21: Let us reconsider the embedded SQL code of Fig. 6, where we obtained values for two variables `studioName` and `studioAddr` and used them as the components of a tuple, which we inserted into `Studio`. Figure 21 sketches how this process would work in CLI. It assumes that we have a statement handle `myStat` to use for the insertion statement.

```
        /* get values for studioName and studioAddr */

1)  SQLPrepare(myStat,
        "INSERT INTO Studio(name, address) VALUES(?, ?)",
        SQL_NTS);
2)  SQLBindParameter(myStat, 1,..., studioName,...);
3)  SQLBindParameter(myStat, 2,..., studioAddr,...);
4)  SQLExecute(myStat);
```

Figure 21: Inserting a new studio by binding parameters to values

The code begins with steps (not shown) to give `studioName` and `studioAddr` values. Line (1) shows statement `myStat` being prepared to be an insertion statement with two parameters (the question-marks) in the `VALUE` clause. Then, lines (2) and (3) bind the first and second question-marks, to the current contents of `studioName` and `studioAddr`, respectively. Finally, line (4) executes the insertion. If the entire sequence of steps in Fig. 21, including the unseen work to obtain new values for `studioName` and `studioAddr`, are placed in a loop,

then each time around the loop, a new tuple, with a new name and address for a studio, is inserted into `Studio`. □

5.5 Exercises for Section 5

Exercise 5.1: Repeat the problems of Exercise 3.1, but write the code in C with CLI calls.

Exercise 5.2: Repeat the problems of Exercise 3.2, but write the code in C with CLI calls.

6 JDBC

Java Database Connectivity, or JDBC, is a facility similar to CLI for allowing Java programs to access SQL databases. The concepts resemble those of CLI, although Java's object-oriented flavor is evident in JDBC.

6.1 Introduction to JDBC

The first steps we must take to use JDBC are:

1. include the line:

   ```
   import java.sql.*;
   ```

   to make the JDBC classes available to your Java program.

2. Load a "driver" for the database system we shall use. The driver we need depends on which DBMS is available to us, but we load the needed driver with the statement:

   `Class.forName(`<driver name>`);`

   For example, to get the driver for a MySQL database, execute:

   ```
   Class.forName("com.mysql.jdbc.Driver");
   ```

   The effect is that a class called `DriverManager` is available. This class is analogous in many ways to the environment whose handle we get as the first step in using CLI.

3. Establish a connection to the database. A variable of class Connection is created if we apply the method `getConnection` to `DriverManager`.

The Java statement to establish a connection looks like:

```
Connection myCon = DriverManager.getConnection(<URL>,
        <user name>, <password>);
```

That is, the method `getConnection` takes as arguments the URL for the database to which you wish to connect, your user name, and your password. It returns an object of class Connection, which we have chosen to call `myCon`.

Example 22: Each DBMS has its own way of specifying the URL in the `getConnection` method. For instance, if you want to connect to a MySQL database, the form of the URL is

`jdbc:mysql://`<host name>`/`<database name>

□

A JDBC Connection object is quite analogous to a CLI connection, and it serves the same purpose. By applying the appropriate methods to a Connection like `myCon`, we can create statement objects, place SQL statements "in" those objects, bind values to SQL statement parameters, execute the SQL statements, and examine results a tuple at a time.

6.2 Creating Statements in JDBC

There are two methods we can apply to a Connection object in order to create statements:

1. `createStatement()` returns a Statement object. This object has no associated SQL statement yet, so method `createStatement()` may be thought of as analogous to the CLI call to `SQLAllocHandle` that takes a connection handle and returns a statement handle.

2. `prepareStatement(`Q`)`, where Q is a SQL query passed as a string argument, returns a PreparedStatement object. Thus, we may draw an analogy between executing `prepareStatement(`Q`)` in JDBC with the two CLI steps in which we get a statement handle with `SQLAllocHandle` and then apply `SQLPrepare` to that handle and the query Q.

There are four different methods that execute SQL statements. Like the methods above, they differ in whether or not they take a SQL statement as an argument. However, these methods also distinguish between SQL statements that are queries and other statements, which are collectively called "updates." Note that the SQL `UPDATE` statement is only one small example of what JDBC terms an "update." The latter include all modification statements, such as inserts, and all schema-related statements such as `CREATE TABLE`. The four "execute" methods are:

a) `executeQuery(`Q`)` takes a statement Q, which must be a query, and is applied to a Statement object. This method returns a ResultSet object, which is the set (bag, to be precise) of tuples produced by the query Q. We shall see how to access these tuples in Section 6.3.

b) `executeQuery()` is applied to a PreparedStatement object. Since a prepared statement already has an associated query, there is no argument. This method also returns a ResultSet object.

c) `executeUpdate(`U`)` takes a nonquery statement U and, when applied to a Statement object, executes U. The effect is felt on the database only; no ResultSet object is returned.

d) `executeUpdate()`, with no argument, is applied to a PreparedStatement object. In that case, the SQL statement associated with the prepared statement is executed. This SQL statement must not be a query, of course.

Example 23: Suppose we have a Connection object `myCon`, and we wish to execute the query

```
SELECT netWorth FROM MovieExec;
```

One way to do so is to create a Statement object `execStat`, and then use it to execute the query directly.

```
Statement execStat = myCon.createStatement();
ResultSet worths = execStat.executeQuery(
    "SELECT netWorth FROM MovieExec");
```

The result of the query is a ResultSet object, which we have named `worths`. We'll see in Section 6.3 how to extract the tuples from `worths` and process them.

An alternative is to prepare the query immediately and later execute it. This approach would be preferable should we want to execute the same query repeatedly. Then, it makes sense to prepare it once and execute it many times, rather than having the DBMS prepare the same query many times. The JDBC steps needed to follow this approach are:

```
PreparedStatement execStat = myCon.prepareStatement(
    "SELECT netWorth FROM MovieExec");
ResultSet worths = execStat.executeQuery();
```

The result of executing the query is again a ResultSet object, which we have called `worths`. □

Example 24: If we want to execute a parameterless nonquery, we can perform analogous steps in both styles. There is no result set, however. For instance, suppose we want to insert into `StarsIn` the fact that Denzel Washington starred in *Remember the Titans* in the year 2000. We may create and use a statement `starStat` in either of the following ways:

```
Statement starStat = myCon.createStatement();
starStat.executeUpdate("INSERT INTO StarsIn VALUES(" +
    "'Remember the Titans', 2000, 'Denzel Washington')");
```

or

```
PreparedStatement starStat = myCon.prepareStatement(
    "INSERT INTO StarsIn VALUES('Remember the Titans'," +
    "2000, 'Denzel Washington')");
starStat.executeUpdate();
```

Notice that each of these sequences of Java statements takes advantage of the fact that + is the Java operator that concatenates strings. Thus, we are able to extend SQL statements over several lines of Java, as needed. □

6.3 Cursor Operations in JDBC

When we execute a query and obtain a result-set object, we may, in effect, run a cursor through the tuples of the result set. To do so, the `ResultSet` class provides the following useful methods:

1. `next()`, when applied to a ResultSet object, causes an implicit cursor to move to the next tuple (to the first tuple the first time it is applied). This method returns `FALSE` if there is no next tuple.

2. `getString(`i`)`, `getInt(`i`)`, `getFloat(`i`)`, and analogous methods for the other types that SQL values can take, each return the ith component of the tuple currently indicated by the cursor. The method appropriate to the type of the ith component must be used.

Example 25: Having obtained the result set `worths` as in Example 23, we may access its tuples one at a time. Recall that these tuples have only one component, of type integer. The form of the loop is:

```
while(worths.next()) {
    int worth = worths.getInt(1);
    /* process this net worth */
};
```

□

6.4 Parameter Passing

As in CLI, we can use a question-mark in place of a portion of a query, and then bind values to those *parameters*. To do so in JDBC, we need to create a prepared statement, and we need to apply to that PreparedStatement object methods such as `setString(`i`, `v`)` or `setInt(`i`, `v`)` that bind the value v, which must be of the appropriate type for the method, to the ith parameter in the query.

Example 26: Let us mimic the CLI code in Example 21, where we prepared a statement to insert a new studio into relation `Studio`, with parameters for the name and address of that studio. The Java code to prepare this statement, set its parameters, and execute it is shown in Fig. 22. We continue to assume that connection object `myCon` is available to us.

```
1)  PreparedStatement studioStat = myCon.prepareStatement(
2)      "INSERT INTO Studio(name, address) VALUES(?, ?)");
    /* get values for variables studioName and studioAddr
       from the user */
3)  studioStat.setString(1, studioName);
4)  studioStat.setString(2, studioAddr);
5)  studioStat.executeUpdate();
```

Figure 22: Setting and using parameters in JDBC

In lines (1) and (2), we create and prepare the insertion statement. It has parameters for each of the values to be inserted. After line (2), we could begin a loop in which we repeatedly ask the user for a studio name and address, and place these strings in the variables `studioName` and `studioAddr`. This assignment is not shown, but represented by a comment. Lines (3) and (4) set the first and second parameters to the strings that are the current values of `studioName` and `studioAddr`, respectively. Finally, at line (5), we execute the insertion statement with the current values of its parameters. After line (5), we could go around the loop again, beginning with the steps represented by the comment. □

6.5 Exercises for Section 6

Exercise 6.1: Repeat Exercise 3.1, but write the code in Java using JDBC.

Exercise 6.2: Repeat Exercise 3.2, but write the code in Java using JDBC.

7 PHP

PHP is a scripting language for helping to create HTML Web pages. It provides support for database operations through an available library, much as JDBC

What Does PHP Stand For?

Originally, PHP was an acronym for "Personal Home Page." More recently, it is said to be the recursive acronym "PHP: Hypertext Preprocessor" in the spirit of other recursive acronyms such as GNU (= "GNU is Not Unix").

does. In this section we shall give a brief overview of PHP and show how database operations are performed in this language.

7.1 PHP Basics

All PHP code is intended to exist inside HTML text. A browser will recognize that text is PHP code by placing it inside a special tag, which looks like:

```
<?php
   PHP code goes here
?>
```

Many aspects of PHP, such as assignment statements, branches, and loops, will be familiar to the C or Java programmer, and we shall not cover them explicitly. However, there are some interesting features of PHP of which we should be aware.

Variables

Variables are untyped and need not be declared. All variable names begin with $.

Often, a variable will be declared to be a member of a "class," in which case certain *functions* (analogous to methods in Java) may be applied to that variable. The function-application operator is `->`, comparable to the dot in Java or C++.

Strings

String values in PHP can be surrounded by either single or double quotes, but there is an important difference. Strings surrounded by single quotes are treated literally, just like SQL strings. However, when a string has double quotes around it, any variable names within the string are replaced by their values.

Example 27: In the following code:

```
$foo = 'bar';
$x = 'Step up to the $foo';
```

the value of `$x` is `Step up to the $foo`. However, if the following code is executed instead:

```
$foo = "bar";
$x = "Step up to the $foo";
```

the value of `$x` is `Step up to the bar`. It doesn't matter whether `bar` has single or double quotes, since it contains no dollar-signs and therefore no variables. However, the variable `$foo` is replaced only when surrounded by double quotes, as in the second example. □

Concatenation of strings is denoted by a dot. Thus,

```
$y = "$foo" . 'bar';
```

gives `$y` the value `barbar`.

7.2 Arrays

PHP has ordinary arrays (called *numeric*), which are indexed $0, 1, \ldots$. It also has arrays that are really mappings, called *associative arrays*. The indexes (*keys*) of an associative array can be any strings, and the array associates a single value with each key. Both kinds of arrays use the conventional square brackets for indexing, but for associative arrays, an array element is represented by:

<key> `=>` <value>

Example 28: The following line:

```
$a = array(30,20,10,0);
```

sets `$a` to be a numeric array of length four, with `$a[0]` equal to 30, `$a[1]` equal to 20, and so on. □

Example 29: The following line:

```
$seasons = array('spring' => 'warm',  'summer' => 'hot',
                 'fall' => 'warm', 'winter' => 'cold');
```

makes `$seasons` be an array of length four, but it is an associative array. For instance, `$seasons['summer']` has the value `'hot'`. □

7.3 The PEAR DB Library

PHP has a collection of libraries called PEAR (PHP Extension and Application Repository). One of these libraries, DB, has generic functions that are analogous to the methods of JDBC. We tell the function `DB::connect` which vendor's DBMS we wish to access, but none of the other functions of DB need to know about which DBMS we are using. Note that the double colon in `DB::connect` is PHP's way of saying "the function `connect` in the DB library." We make the DB library available to our PHP program with the statement:

```
include(DB.php);
```

7.4 Creating a Database Connection Using DB

The form of an invocation of the `connect` function is:

```
$myCon = DB::connect(<vendor>://<user name>:<password>
            <host name>/<database name>);
```

The components of this call are like those in the analogous JDBC statement that creates a connection (see Section 6.1). The one exception is the vendor, which is a code used by the DB library. For example, `mysqli` is the code for recent versions of the MySQL database.

After executing this statement, the variable `$myCon` is a connection. Like all PHP variables, `$myCon` can change its type. But as long as it is a connection, we may apply to it a number of useful functions that enable us to manipulate the database to which the connection was made. For example, we can disconnect from the database by

```
$myCon->disconnect();
```

Remember that `->` is the PHP way of applying a function to an "object."

7.5 Executing SQL Statements

All SQL statements are referred to as "queries" and are executed by the function `query`, which takes the statement as an argument and is applied to the connection variable.

Example 30: Let us duplicate the insertion statement of Example 24, where we inserted Denzel Washington and *Remember the Titans* into the `StarsIn` table. Assuming that `$myCon` has connected to our movie database, We can simply say:

```
$result = $myCon->query("INSERT INTO StarsIn VALUES(" .
    "'Denzel Washington', 2000, 'Remember the Titans')");
```

Note that the dot concatenates the two strings that form the query. We only broke the query into two strings because it was necessary to break it over two lines.

The variable `$result` will hold an error code if the insert-statement failed to execute. If the "query" were really a SQL query, then `$result` is a cursor to the tuples of the result (see Section 7.6). □

PHP allows SQL to have parameters, denoted by question-marks, as we shall discuss in Section 7.7. However, the ability to expand variables in doubly quoted strings gives us another easy way to execute SQL statements that depend on user input. In particular, since PHP is used within Web pages, there are built-in ways to exploit HTML's capabilities.

We often get information from a user of a Web page by showing them a form and having their answers "posted." PHP provides an associative array called `$_POST` with all the information provided by the user. Its keys are the names of the form elements, and the associated values are what the user has entered into the form.

Example 31: Suppose we ask the user to fill out a form whose elements are `title`, `year`, and `starName`. These three values will form a tuple that we may insert into the table `StarsIn`. The statement:

```
$result = $myCon->query("INSERT INTO StarsIn VALUES(
    $_POST['title'], $_POST['year'], $_POST['starName'])");
```

will obtain the posted values for these three form elements. Since the query argument is a double-quoted string, PHP evaluates terms like `$_POST['title']` and replaces them by their values. □

7.6 Cursor Operations in PHP

When the `query` function gets a true query as argument, it returns a result object, that is, a list of tuples. Each tuple is a numeric array, indexed by integers starting at 0. The essential function that we can apply to a result object is `fetchRow()`, which returns the next row, or 0 (false) if there is no next row.

```
1) $worths = $myCon->query("SELECT netWorth FROM MovieExec");
2) while ($tuple = $worths->fetchRow()) {
3)     $worth = $tuple[0];
          // process this value of $worth
      }
```

Figure 23: Finding and processing net worths in PHP

Example 32: In Fig. 23 is PHP code that is the equivalent of the JDBC in Examples 23 and 25. It assumes that connection `$myCon` is available, as before.

Line (1) passes the query to the connection `$myCon`, and the result object is assigned to the variable `$worths`. We then enter a loop, in which we repeatedly get a tuple from the result and assign this tuple to the variable `$tuple`, which technically becomes an array of length 1, with only a component for the column `netWorth`. As in C, the value returned by `fetchRow()` becomes the value of the condition in the while-statement. Thus, if no tuple is found, this value, 0, terminates the loop. At line (3), the value of the tuple's first (and only) component is extracted and assigned to the variable `$worth`. We do not show the processing of this value. □

7.7 Dynamic SQL in PHP

As in JDBC, PHP allows a SQL query to contain question-marks. These question-marks are placeholders for values that can be filled in later, during the execution of the statement. The process of doing so is as follows.

We may apply `prepare` and `execute` functions to a connection; these functions are analogous to similarly named functions discussed in Section 3.9 and elsewhere. Function `prepare` takes a SQL statement as argument and returns a prepared version of that statement. Function `execute` takes two arguments: the prepared statement and an array of values to substitute for the question-marks in the statement. If there is only one question-mark, a simple variable, rather than an array, suffices.

Example 33: Let us again look at the problem of Example 26, where we prepared to insert many name-address pairs into relation `Studio`. To begin, we prepare the query, with parameters, by:

```
$prepQuery = $myCon->prepare("INSERT INTO Studio(name, " .
                  "address) VALUES(?,?)");
```

Now, `$prepQuery` is a "prepared query." We can use it as an argument to `execute` along with an array of two values, a studio name and address. For example, we could perform the following statements:

```
$args = array('MGM', 'Los Angeles');
$result = $myCon->execute($prepQuery, $args);
```

The advantage of this arrangement is the same as for all implementations of dynamic SQL. If we insert many different tuples this way, we only have to prepare the insertion statement once and can execute it many times. □

7.8 Exercises for Section 7

Exercise 7.1: Repeat Exercise 3.1, but write the code using PHP.

Exercise 7.2: Repeat Exercise 3.2, but write the code using PHP.

! Exercise 7.3: In Example 31 we exploited the feature of PHP that strings in double-quotes have variables expanded. How essential is this feature? Could we have done something analogous in JDBC? If so, how?

8 Summary

- *Three-Tier Architectures*: Large database installations that support large-scale user interactions over the Web commonly use three tiers of processes: web servers, application servers, and database servers. There can be many processes active at each tier, and these processes can be at one processor or distributed over many processors.

- *Client-Server Systems in the SQL Standard*: The standard talks of SQL clients connecting to SQL servers, creating a connection (link between the two processes) and a session (sequence of operations). The code executed during the session comes from a module, and the execution of the module is called a SQL agent.

- *The Database Environment*: An installation using a SQL DBMS creates a SQL environment. Within the environment, database elements such as relations are grouped into (database) schemas, catalogs, and clusters. A catalog is a collection of schemas, and a cluster is the largest collection of elements that one user may see.

- *Impedance Mismatch*: The data model of SQL is quite different from the data models of conventional host languages. Thus, information passes between SQL and the host language through shared variables that can represent components of tuples in the SQL portion of the program.

- *Embedded SQL*: Instead of using a generic query interface to express SQL queries and modifications, it is often more effective to write programs that embed SQL queries in a conventional host language. A preprocessor converts the embedded SQL statements into suitable function calls of the host language.

- *Cursors*: A cursor is a SQL variable that indicates one of the tuples of a relation. Connection between the host language and SQL is facilitated by having the cursor range over each tuple of the relation, while the components of the current tuple are retrieved into shared variables and processed using the host language.

- ✦ *Dynamic SQL*: Instead of embedding particular SQL statements in a host-language program, the host program may create character strings that are interpreted by the SQL system as SQL statements and executed.
- ✦ *Persistent Stored Modules*: We may create collections of procedures and functions as part of a database schema. These are written in a special language that has all the familiar control primitives, as well as SQL statements.
- *The Call-Level Interface*: There is a standard library of functions, called SQL/CLI or ODBC, that can be linked into any C program. These functions give capabilities similar to embedded SQL, but without the need for a preprocessor.
- ✦ *JDBC*: Java Database Connectivity is a collection of Java classes analogous to CLI for connecting Java programs to a database.
- ✦ *PHP*: Another popular system for implementing a call-level interface is PHP. This language is found embedded in HTML pages and enables these pages to interact with a database.

9 References

The PSM standard is [4], and [5] is a comprehensive book on the subject. Oracle's version of PSM is called PL/SQL; a summary can be found in [2]. SQL Server has a version called Transact-SQL [6]. IBM's version is SQL PL [1].

[3] is a popular reference on JDBC. [7] is one on PHP, which was originally developed by one of the book's authors, R. Lerdorf.

1. D. Bradstock et al., *DB2 SQL Procedure Language for Linux, Unix, and Windows*, IBM Press, 2005.
2. Y.-M. Chang et al., "Using Oracle PL/SQL"
   `http://infolab.stanford.edu/~ullman/fcdb/oracle/or-plsql.html`
3. M. Fisher, J. Ellis, and J. Bruce, *JDBC API Tutorial and Reference*, Prentice-Hall, Upper Saddle River, NJ, 2003.
4. ISO/IEC Report 9075-4, 2003.
5. J. Melton, *Understanding SQL's Stored Procedures: A Complete Guide to SQL/PSM*, Morgan-Kaufmann, San Francisco, 1998.
6. Microsoft Corp., "Transact-SQL Reference"
   `http://msdn2.microsoft.com/en-us/library/ms189826.aspx`
7. K. Tatroe, R. Lerdorf, and P. MacIntyre, *Programming PHP*, O'Reilly Media, Cambridge, MA, 2006.