*! 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 7.3.2: 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.

7.4 Schema-Level Constraints and Triggers

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 the relations and views themselves.

- 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.

While assertions are easier for the programmer to use, since they merely require the programmer to state what must be true, 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.

7.4.1 Assertions

The SQL standard proposes a simple form of *assertion* (also called a "general constraint") 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:

1. The keywords `CREATE ASSERTION`,
2. The name of the assertion,
3. The keyword `CHECK`, and
4. A parenthesized condition.

That is, the form of this statement is

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

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

There is a difference between the way we write tuple-based `CHECK` constraints and the way we write assertions. Tuple-based checks can refer to the attributes of that relation in whose declaration they appear. For instance, in line (6) of Fig. 7.5 we used attributes `gender` and `name` without saying where they came from. They refer to components of a tuple being inserted or updated in the table `MovieStar`, because that table is the one being declared in the `CREATE TABLE` statement.

The condition of 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 normal to aggregate the results of the condition 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 aggregate 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 7.13: 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#)
```

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

Figure 7.6: Assertion guaranteeing rich studio presidents

The assertion is shown in Fig. 7.6.

Incidentally, it is worth noting that even though this constraint involves two relations, we could write it as tuple-based CHECK constraints on the two relations rather than as a single assertion. For instance, we can add to the CREATE TABLE statement of Example 7.3 a constraint on Studio as shown in Fig. 7.7.

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presC# INT REFERENCES MovieExec(cert#),
    CHECK (presC# NOT IN
        (SELECT cert# FROM MovieExec
         WHERE netWorth < 10000000)
    )
);
```

Figure 7.7: A constraint on Studio mirroring an assertion

Note, however, that the constraint of Fig. 7.7 will only be checked when a change to its relation, Studio occurs. It would not catch a situation where the net worth of some studio president, as recorded in relation MovieExec, dropped below \$10,000,000. To get the full effect of the assertion, we would have to add another constraint to the declaration of the table MovieExec, requiring that the net worth be at least \$10,000,000 if that executive is the president of a studio. □

Example 7.14: Here is another example of an assertion. It involves the relation

```
Movie(title, year, length, inColor, studioName, producerC#)
```

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

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 Movie GROUP BY studioName)
);
```

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

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

Notice that in principle this condition applies to every tuple of table Movie. 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 Movie. 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>
```

7.4.2 Event-Condition-Action Rules

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 (we mentioned transactions briefly in Section 7.1.6 and cover them with more detail in Section 8.6).

2. Instead of immediately preventing the event that awakened it, a 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. The action may then prevent the event from taking place, or it could undo the event (e.g., delete the tuple inserted). In fact, the action could be any sequence of database operations, perhaps even operations not connected in any way to the triggering event.

7.4.3 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 action may be executed either before or after the triggering event.

2. The action can refer to both old and/or new values of tuples that were inserted, deleted, or updated in the event that triggered the action.

3. Update events may be limited to a particular attribute or set of attributes.

4. A condition may be specified by a `WHEN` clause; the action is executed only if the rule is triggered *and* the condition holds when the triggering event occurs.

5. The programmer has an option of specifying that the action is performed either:

   (a) Once for each modified tuple, or

   (b) Once for all the tuples that are changed in one database operation.

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. In this example, the trigger executes once for each tuple that is updated.

Example 7.15 : We shall write an 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. The trigger declaration appears in Fig. 7.8.

```
 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 7.8: An 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. If this phrase is missing or it is replaced by the default `FOR EACH STATEMENT`, then the triggering would occur once for an SQL statement, no matter how many triggering-event changes to tuples it made. We would not then declare alias for old and new rows, but we might use `OLD TABLE` and `NEW TABLE`, introduced below.

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. □

Of course Example 7.15 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. 7.8 says that the action of the rule is executed after the triggering event, as indicated by the keyword `AFTER`. We may replace `AFTER` by `BEFORE`, in which case the `WHEN` condition is tested before the triggering event, that is, before the modification that awakened the trigger has been made to the database. If the condition is true, then the action of the trigger is executed. Then, the event that awakened the trigger is executed, regardless of whether the condition is true.

- Besides `UPDATE`, other possible triggering events are `INSERT` and `DELETE`. The `OF netWorth` clause in line (2) of Fig. 7.8 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.

- 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 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), then a *row-level trigger* such as Fig. 7.8 becomes a *statement-level trigger*. A statement-level trigger is executed once whenever a statment 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 an SQL update statement, a statement-level update trigger would execute only once, while a tuple-level trigger would execute once for each tuple to which an update is 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 7.16: 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 `INSERT` or `UPDATE` statement insert or change many tuples of `MovieExec`, and 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 7.9 shows the trigger for the update event. The triggers for the insertion and deletion of tuples are similar but slightly simpler.

```
 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 7.9: Constraining the average net worth

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`.

If the operation is an update, then `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. □

7.4.4 Instead-Of Triggers

There is a useful feature of triggers that did not make the SQL-99 standard, but figured into the discussion of the standard and is supported by some commercial systems. This extension allows BEFORE or AFTER to be replaced by INSTEAD OF; the meaning is that when an event awakens a trigger, the action of the trigger is done instead of the event itself.

This capability offers little when the trigger is on a stored table, but it is very powerful when used on a view. The reason is that we cannot really modify a view (see Section 6.7.4). 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.

Example 7.17: Let us recall the definition of the view of all movies owned by Paramount:

```
CREATE VIEW ParamountMovie AS
   SELECT title, year
   FROM Movie
   WHERE studioName = 'Paramount';
```

from Example 6.45. As we discussed in Example 6.49, this view is updatable, but it has the unexpected flaw that when you insert a tuple into Paramount-Movie, the system cannot deduce that the studioName attribute is surely Paramount, so that attribute is NULL in the inserted Movie tuple.

A better result can be obtained if we create an instead-of trigger on this view, as shown in Fig. 7.10. Much of the trigger is unsurprising. We see the keyword INSTEAD OF on line (2), establishing that an attempt to insert into ParamountMovie will never take place.

Rather, we see in lines (5) and (6) the action that replaces the attempted insertion. There is an insertion into Movie, 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

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

Figure 7.10: Trigger to replace an insertion on a view by an insertion on the underlying base table

value of attribute studioName is the constant 'Paramount'. This value is not part of the inserted tuple. Rather, we assume it is the correct studio for the inserted movie, because the insertion came through the view ParamountMovie. □

7.4.5 Exercises for Section 7.4

Exercise 7.4.1: Write the triggers analogous to Fig. 7.9 for the insertion and deletion events on MovieExec.

Exercise 7.4.2: Write the following as triggers or assertions. In each case, disallow or undo the modification if it does not satisfy the stated constraint. The database schema is from the "PC" example of Exercise 5.2.1:

```
Product(maker, model, type)
PC(model, speed, ram, hd, rd, 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) No manufacturer of PC's may also make laptops.

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

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

! e) When making any modification to the Laptop relation, check that the average price of laptops for each manufacturer is at least $2000.

! f) 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.

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

! h) 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`.

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

Exercise 7.4.3: Write the following as triggers or assertions. In each case, disallow or undo the modification if it does not satisfy the stated constraint. The database schema is from the battleships example of Exercise 5.2.4.

```
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) No class may have more than 2 ships.

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

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

! f) 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.

! g) 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.

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

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

!! j) 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 7.4.4:** Write the following as triggers or assertions. 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:

```
Movie(title, year, length, inColor, 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.

7.5 Summary of Chapter 7

✦ *Key Constraints*: We can declare an attribute or set of attributes to be a key with a `UNIQUE` or `PRIMARY KEY` declaration in a relation schema.

✦ *Referential Integrity Constraints*: We can declare that a value appearing in some attribute or set of attributes must also appear in the corresponding attributes of some tuple of another relation with a `REFERENCES` or `FOREIGN KEY` declaration in a 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 with the keyword CHECK and the 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, these 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.

7.6 References for Chapter 7

The reader should go to the bibliographic notes for Chapter 6 for information about how to get the SQL2 or SQL-99 standards documents. 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. Cochrane, R. J., 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. Dayal, U., et al., "The HiPAC project: combining active databases and timing constraints," *SIGMOD Record* **17**:1, pp. 51–70, 1988.

3. McCarthy, D. R., 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. Widom, J. and S. Ceri, *Active Database Systems*, Morgan-Kaufmann, San Francisco, 1996.

Chapter 8

System Aspects of SQL

We now turn to the question of how SQL fits into a complete programming environment. In Section 8.1 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 8.2 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. Section 8.3 covers additional system issues, such as support for a client-server model of computing.

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 8.4 we discuss the SQL-standard library called SQL/CLI, for making calls from C programs. Then, in Section 8.5 we meet Java's JDBC (database connectivity), which is an alternative call-level interface.

Then, Section 8.6 introduces us to the "transaction," an atomic unit of work. Many database applications, such as banking, require that operations on the data appear atomic, or indivisible, even though a large number of concurrent operations may be in progress at once. SQL provides features to allow us to specify transactions, and SQL systems have mechanisms to make sure that what we call a transaction is indeed executed atomically. Finally, Section 8.7 discusses how SQL controls unauthorized access to data, and how we can tell the SQL system what accesses are authorized.

8.1 SQL in a Programming Environment

To this point, we have used the *generic SQL interface* in our examples. That is, we have assumed there is an 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