**UNIT- II**

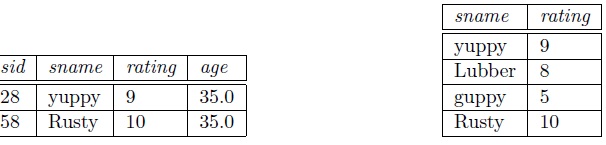
**RELATIONAL ALGEBRA**

Relational algebra is one of the two formal query languages associated with the relational model. Queries in algebra are composed using a collection of operators. A fundamental property is that every operator in the algebra accepts (one or two) relation instances as arguments and returns a relation instance as the result. This property makes it easy to *compose* operators to form a complex query—a **Relational algebra expression** is recursively defined to be a relation, a unary algebra operator applied to a single expression, or a binary algebra operator applied to two expressions. We describe the basic operators of the algebra (selection, projection, union, cross-product, and difference), as well as some additional operators that can be defined in terms of the basic operators but arise frequently enough to warrant special attention, in the following sections.

**Selection and Projection**

Relational algebra includes operators to *select* rows from a relation (*σ*) and to *project* columns (*π*). These operations allow us to manipulate data in a single relation. Consider the instance of the Sailors relation denoted as *S2*. We can retrieve rows corresponding to expert sailors by using the *σ* operator.

The expression *σrating>*8(*S*2) evaluates to the relation



**Set Operations**

The following standard operations on sets are also available in relational algebra: *union* (*∪*), *intersection* (*∩*), *set-difference* (*−*), and *cross-product* (*×*).

**Union:** *R∪S* returns a relation instance containing all tuples that occur in *either* relation instance *R* or relation instance *S* (or both). *R* and *S* must be *unioncompatible*, and the schema of the result is defined to be identical to the schema of *R*.

Two relation instances are said to be **union-compatible** if the following conditions hold:

**–** they have the same number of the fields, and

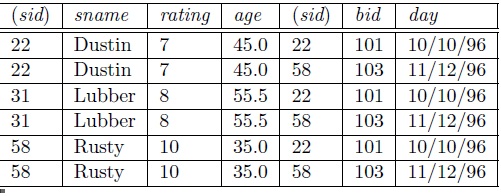
**–** corresponding fields, taken in order from left to right, have the same *domains*.

**Intersection:** *R∩S* returns a relation instance containing all tuples that occur in *both R* and *S*. The relations *R* and *S* must be union-compatible, and the schema of the result is defined to be identical to the schema of *R*.

**Set-difference:** *R−S* returns a relation instance containing all tuples that occur in *R* but not in *S*. The relations *R* and *S* must be union-compatible, and the schema of the result is defined to be identical to the schema of *R*.

**Cross-product:** *R×S* returns a relation instance whose schema contains all the fields of *R* (in the same order as they appear in *R*) followed by all the fields of *S* (in the same order as they appear in *S*). The result of *R × S* contains one tuple *\_r, s\_* (the concatenation of tuples *r* and *s*) for each pair of tuples *r ∈ R, s ∈ S*. The cross-product opertion is sometimes called **Cartesian product**.

We will use the convention that the fields of *R × S* inherit names from the corresponding fields of *R* and *S*. It is possible for both *R* and *S* to contain one or more fields having the same name; this situation creates a *naming conflict*. The corresponding fields in *R × S* are unnamed and are referred to solely by position.



**Renaming:**

We introduce a **renaming** operator *ρ* for this purpose. The expression *ρ*(*R*(*F*)*,E*) takes an arbitrary relational algebra expression *E* and returns an instance of a (new) relation called *R*. *R* contains the same tuples as the result of *E*, and has the same schema as *E*, but some fields are renamed. The field names in relation *R* are the same as in *E*, except for fields renamed in the *renaming list F*, which is a list ofterms having the form *oldname → newname* or *position → newname*. For *ρ* to be well-defined, references to fields (in the form of *oldname*s or *position*s in the renaming list) may be unambiguous, and no two fields in the result must have the same name. Sometimes we only want to rename fields or to (re)name the relation; we will therefore treat both *R* and *F* as optional in the use of *ρ*. (Of course, it is meaningless to omit both.)

For example, the expression *ρ*(*C*(1 *→ sid*1*,* 5 *→ sid*2)*, S*1 *× R*1) returns a relation that contains the tuples shown in Figure 4.11 and has the following schema: C(*sid1:* integer, *sname:* string, *rating:* integer, *age:* real, *sid2:* integer, *bid:* integer, *day:* dates).

It is customary to include some additional operators in the algebra, but they can all be defined in terms of the operators that we have defined thus far. (In fact, the renaming operator is only needed for syntactic convenience, and even the *∩* operator is redundant; *R ∩ S* can be defined as *R −* (*R − S*).) We will consider these additional operators, and their definition in terms of the basic operators, in the next two subsections.

**Joins:**

The *join* operation is one of the most useful operations in relational algebra and is the most commonly used way to combine information from two or more relations. Although a join can be defined as a cross-product followed by selections and projections, joins arise much more frequently in practice than plain cross-products. Further, the result of a cross-product is typically much larger than the result of a join, and it is very important to recognize joins and implement them without materializing the underlying cross-product (by applying the selections and projections ‘on-the-fly’). For these reasons, joins have received a lot of attention, and there are several variants of the join operation.

**Condition Joins**

The most general version of the join operation accepts a *join condition c* and a pair of relation instances as arguments, and returns a relation instance. The *join condition* is identical to a *selection condition* in form. The operation is defined as follows:

*R \_\_c S* = *σc*(*R × S*)

Thus *\_\_* is defined to be a cross-product followed by a selection. Note that the condition *c* can (and typically *does*) refer to attributes of both *R* and *S*. The reference to an attribute of a relation, say *R*, can be by position (of the form *R.i*) or by name (of the form *R.name*).

As an example, the result of *S*1 *\_\_S*1*.sid<R*1*.sid R*1. Because *sid* appears in both *S*1 and *R*1, the corresponding fields in the result of the cross-product *S*1 *× R*1 (and therefore in the result of *S*1 *\_\_S*1*.sid<R*1*.sid R*1) are unnamed. Domains are inherited from the corresponding fields of *S*1 and *R*1.

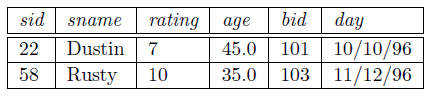


**Equijoin**

A common special case of the join operation *R \_\_ S* is when the *join condition* consists solely of equalities (connected by *∧*) of the form *R.name*1 = *S.name*2, that is, equalities between two fields in *R* and *S*. In this case, obviously, there is some redundancy in retaining both attributes in the result. For join conditions that contain only such equalities, the join operation is refined by doing an additional projection in which *S.name*2 is dropped. The join operation with this refinement is called **equijoin**.

The schema of the result of an equijoin contains the fields of *R* (with the same names and domains as in *R*) followed by the fields of *S* that do not appear in the join conditions. If this set of fields in the result relation includes two fields that inherit the same name from *R* and *S*, they are unnamed in the result relation.

We illustrate *S*1 *\_\_R.sid*=*S.sid R*1 as shown below Notice that only one field called *sid* appears in the result.



**RELATIONAL CALCULUS:**

Relational calculus is an alternative to relational algebra. In contrast to the algebra, which is procedural, the calculus is nonprocedural, or *declarative*, in that it allows us to describe the set of answers without being explicit about how they should be computed. Relational calculus has had a big influence on the design of commercial query languages such as SQL and, especially, Query-by-Example (QBE). The variant of the calculus that we present in detail is called the tuple relational calculus (TRC). Variables in TRC take on tuples as values. In another variant, called the domain relational calculus (DRC), the variables range over field values. TRC has had more of an influence on SQL, while DRC has strongly influenced QBE.

**Tuple Relational Calculus**

A **tuple variable** is a variable that takes on tuples of a particular relation schema as values. That is, every value assigned to a given tuple variable has the same number and type of fields. A tuple relational calculus query has the form *{ T | p(T) }*, where *T* is a tuple variable and *p*(*T*) denotes a *formula* that describes *T*; we will shortly define formulas and queries rigorously. The result of this query is the set of all tuples *t* for which the formula *p*(*T*) evaluates to true with *T* = *t*. The language for writing formulas *p*(*T*) is thus at the heart of TRC and is essentially a simple subset of *first-order logic*. As a simple example, consider the following query.

*(Q11) Find all sailors with a rating above 7.*

*{S | S ∈ Sailors ∧ S.rating >* 7*}*

When this query is evaluated on an instance of the Sailors relation, the tuple variable *S* is instantiated successively with each tuple, and the test *S.rating>7* is applied. The answer contains those instances of *S* that pass this test. On instance *S*3 of Sailors, the answer contains Sailors tuples with *sid* 31, 32, 58, 71, and 74.

**Syntax of TRC Queries**

Let *Rel* be a relation name, *R* and *S* be tuple variables, *a* an attribute of *R*, and *b* an attribute of *S*. Let op denote an operator in the set *{<, >,*=*,≤,≥, \_*=*}*. An **atomic formula** is one of the following:

*R ∈ Rel*

*R.a* op *S.b*

*R.a* op *constant*, or *constant* op *R.a*

A **formula** is recursively defined to be one of the following, where *p* and *q* are themselves

formulas, and *p*(*R*) denotes a formula in which the variable *R* appears:

**Semantics of TRC Queries**

What does a TRC query mean? More precisely, what is the set of answer tuples for a given TRC query? The **answer** to a TRC query *{T | p(T)}*, as we noted earlier, is the set of all tuples *t* for which the formula *p*(*T*) evaluates to true with variable *T* assigned the tuple value *t*. To complete this definition, we must state which assignments of tuple values to the free variables in a formula make the formula evaluate to true.

A query is evaluated on a given instance of the database. Let each free variable in a formula *F* be bound to a tuple value. For the given assignment of tuples to variables, with respect to the given database instance, *F* evaluates to (or simply ‘is’) true if one of the following holds:

* *F* is an atomic formula *R ∈ Rel*, and *R* is assigned a tuple in the instance of relation *Rel*.
* *F* is a comparison *R.a* op *S.b*, *R.a* op *constant*, or *constant* op *R.a*, and the tuples assigned to *R* and *S* have field values *R.a* and *S.b* that make the comparison true.
* *F* is of the form *￢p*, and *p* is not true; or of the form *p ∧ q*, and both *p* and *q* are true; or of the form *p ∨ q*, and one of them is true, or of the form *p ⇒ q* and *q* is true whenever4 *p* is true.
* *F* is of the form *∃R*(*p*(*R*)), and there is some assignment of tuples to the free variables in *p*(*R*), including the variable *R*,5 that makes the formula *p*(*R*) true.
* *F* is of the form *∀R*(*p*(*R*)), and there is some assignment of tuples to the free variables in *p*(*R*)

**The Domain Relational Calculus**

1. Domain variables take on values from an attribute's domain, rather than values for an entire tuple.
2. An expression is of the form

_7092_displaymath1601

where the _7092_tex2html_wrap1603 represent domain variables, and _7092_tex2html_wrap1579 is a **formula**.

1. An atom in the domain relational calculus is of the following forms
   1. _7092_tex2html_wrap1607 where _7092_tex2html_wrap1345 is a relation on _7092_tex2html_wrap1611 attributes, and _7092_tex2html_wrap1613, are domain variables or constants.
   2. _7092_tex2html_wrap1615, where _7092_tex2html_wrap1653 and _7092_tex2html_wrap1619 are domain variables, and _7092_tex2html_wrap1621 is a comparison operator.
   3. _7092_tex2html_wrap1623, where c is a constant.
2. **Formulae** are built up from atoms using the following rules:
   1. An atom is a formula.
   2. If _7092_tex2html_wrap1579 is a formula, then so are _7092_tex2html_wrap1551 and _7092_tex2html_wrap1553.
   3. If _7092_tex2html_wrap1555 and _7092_tex2html_wrap1633 are formulae, then so are _7092_tex2html_wrap1635, _7092_tex2html_wrap1561 and _7092_tex2html_wrap1563.
   4. If _7092_tex2html_wrap1641 is a formula where x is a domain variable, then so are _7092_tex2html_wrap1643 and _7092_tex2html_wrap1645.
3. Find branch name, loan number, customer name and amount for loans of over $1200.

_7092_displaymath1647

1. Find all customers who have a loan for an amount > than $1200.

_7092_displaymath1648

1. Find all customers having a loan from the SFU branch, and the city in which they live.

_7092_eqnarray822

1. Find all customers having a loan, an account or both at the SFU branch.

_7092_eqnarray824

1. Find all customers who have an account at **all** branches located in Brooklyn.

_7092_eqnarray826

If you find this example difficult to understand, try rewriting this expression using implication, as in the tuple relational calculus example. Here's my attempt:

_7092_eqnarray828

I've used two letter variable names to get away from the problem of having to remember what _7092_tex2html_wrap1653 stands for.

**EXPRESSIVE POWER OF ALGEBRA AND CALCULUS :**

We have presented two formal query languages for the relational model. Are they equivalent in power? Can every query that can be expressed in relational algebra also be expressed in relational calculus? The answer is yes, it can. Can every query that can be expressed in relational calculus also be expressed in relational algebra? Before we answer this question, we consider a major problem with the calculus as we have presented it.

Consider the query *{S | ￢*(*S ∈ Sailors*)*}*. This query is syntactically correct. However, it asks for all tuples *S* such that *S* is not in (the given instance of) Sailors. The set of such *S* tuples is obviously infinite, in the context of infinite domains such as the set of all integers. This simple example illustrates an *unsafe* query. It is desirable to restrict relational calculus to disallow unsafe queries.

We now sketch how calculus queries are restricted to be safe. Consider a set *I* of relation instances, with one instance per relation that appears in the query *Q*. Let *Dom*(*Q, I*) be the set of all constants that appear in these relation instances *I* or in the formulation of the query *Q* itself. Since we only allow finite instances *I*, *Dom*(*Q, I*) is also finite.

We therefore define a *safe* TRC formula *Q* to be a formula such that:

1.For any given *I*, the set of answers for *Q* contains only values that are in *Dom*(*Q, I*).

2. For each subexpression of the form *∃R*(*p*(*R*)) in *Q*, if a tuple *r* (assigned to variable

*R*) makes the formula true, then *r* contains only constants in *Dom*(*Q, I*).

3. For each subexpression of the form *∀R*(*p*(*R*)) in *Q*, if a tuple *r* (assigned to variable

*R*) contains a constant that is not in *Dom*(*Q, I*), then *r* must make the formula true.

Note that this definition is not *constructive*, that is, it does not tell us how to check if

a query is safe.

The query *Q* = *{S | ￢*(*S ∈ Sailors*)*}* is unsafe by this definition. Dom(Q,I) is the set of all values that appear in (an instance *I* of) Sailors. The answer to this query obviously includes values that do not appear in *Dom*(*Q, S*1).

Returning to the question of expressiveness, we can show that every query that can be

expressed using a *safe* relational calculus query can also be expressed as a relational algebra query. The expressive power of relational algebra is often used as a metric of how powerful a relational database query language is. If a query language can express all the queries that we can express in relational algebra, it is said to be **relationally complete**. A practical query language is expected to be relationally complete; in addition, commercial query languages typically support features that allow us to express some queries that cannot be expressed in relational algebra.

**Simple Queries in SQL**

Simple query, like almost all SQL queries, uses the three keywords, SELECT, FROM, and WHERE that characterize SQL.

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#)

Example database schema, repeated In this and subsequent examples, we shall use the movie database schema from Section 2.2.8. For reference, these relation schemas are the ones.

As our first query, let us ask about the relation Movies(title, year, length, genre, studioName, producerC#) for all movies produced by Disney Studios in 1990. In SQL, we say

SELECT \*

FROM Movies

WHERE studioName = ’Disney’ AND year = 1990;

This query exhibits the characteristic select-from-where form of most SQL queries.

SQL is followed by a unique set of rules and guidelines called Syntax. This tutorial gives you a quick start with SQL by listing all the basic SQL Syntax.

All the SQL statements start with any of the keywords like SELECT, INSERT, UPDATE, DELETE, ALTER, DROP, CREATE, USE, SHOW and all the statements end with a semicolon (;).

The most important point to be noted here is that SQL is case insensitive, which means SELECT and select have same meaning in SQL statements. Whereas, MySQL makes difference in table names. So, if you are working with MySQL, then you need to give table names as they exist in the database.

## Various Syntax in SQL

All the examples given in this tutorial have been tested with a MySQL server.

### SQL SELECT Statement

SELECT column1, column2....columnN

FROM table\_name;

### SQL DISTINCT Clause

SELECT DISTINCT column1, column2....columnN

FROM table\_name;

### SQL WHERE Clause

SELECT column1, column2....columnN

FROM table\_name

WHERE CONDITION;

### SQL AND/OR Clause

SELECT column1, column2....columnN

FROM table\_name

WHERE CONDITION-1 {AND|OR} CONDITION-2;

### SQL IN Clause

SELECT column1, column2....columnN

FROM table\_name

WHERE column\_name IN (val-1, val-2,...val-N);

### SQL BETWEEN Clause

SELECT column1, column2....columnN

FROM table\_name

WHERE column\_name BETWEEN val-1 AND val-2;

### SQL LIKE Clause

SELECT column1, column2....columnN

FROM table\_name

WHERE column\_name LIKE { PATTERN };

### SQL ORDER BY Clause

SELECT column1, column2....columnN

FROM table\_name

WHERE CONDITION

ORDER BY column\_name {ASC|DESC};

### SQL GROUP BY Clause

SELECT SUM(column\_name)

FROM table\_name

WHERE CONDITION

GROUP BY column\_name;

### SQL COUNT Clause

SELECT COUNT(column\_name)

FROM table\_name

WHERE CONDITION;

### SQL HAVING Clause

SELECT SUM(column\_name)

FROM table\_name

WHERE CONDITION

GROUP BY column\_name

HAVING (arithematic function condition);

A SQL nested query is a SELECT query that is nested inside a SELECT, UPDATE, INSERT, or DELETE SQL query. Here is a simple example of SQL nested query:

*SELECT Model FROM Product WHERE ManufacturerID IN (SELECT ManufacturerID FROM Manufacturer WHERE Manufacturer = 'Dell')*

The nested query above will select all models from the Product table manufactured by Dell:

|  |
| --- |
| **Model** |
| Inspiron B120 |
| Inspiron B130 |
| Inspiron E1705 |
|  |

You can have more than one level of nesting in one single query.

Correlated Sub Queries are also similar to sub queries but here the outer query is executed first and inner query is executed for each records of outer query. That is inner query is executed as many times as the outer query results.   
Suppose we need to select the students whose marks have been entered into MARKS table. (We can write much simpler query than below by not writing WHERE clause in the sub query, but below query is written to show how correlated sub query is written and executed)

SELECT \*

FROM STUDENT s

WHERE STD\_ID IN (SELECT STD\_ID FROM MARKS m WHERE s.STD\_ID = m.STD\_ID);

 Here we can see that outer query column and inner query column are joined to get the result. This query fetches all the records from STUDENT table and joins with the STD\_ID in MARKS table. It returns the records only if there is a matching STD\_ID in MARKS.

When we observe above query, we see that the STD\_ID in the SELECT statement of sub query is not really required. It is simply used to maintain the structure of the SELECT statement. The main task of sub query here is to certify if the STD\_ID exists in MARKS table or not by checking it in the WHERE clause and then display the result. Hence STD\_ID used in the WHERE clause of outer query and in the SELECT statement of inner query does not have any significance.

Hence DBMS provides another clause EXISTS to use in such cases. The above correlated sub query can be rewritten as below (in either of way):

SELECT \*

FROM STUDENT s

WHERE EXISTS

(SELECT 1

FROM MARKS m

WHERE s.STD\_ID = m.STD\_ID);

SELECT \*

FROM STUDENT s

WHERE EXISTS

(SELECT STD\_ID

FROM MARKS m

WHERE s.STD\_ID = m.STD\_ID);

 We can even use NOT EXISTS clause which performs opposite of EXISTS. Below query returns the students who didn’t get the marks yet.

SELECT \*

FROM STUDENT s

WHERE NOT EXISTS

(SELECT 1

FROM MARKS m

WHERE s.STD\_ID = m.STD\_ID);

SELECT \*

FROM STUDENT s

WHERE NOT EXISTS

|  |  |
| --- | --- |
| **Sub Query** | **Correlated Sub Query** |
| Inner Query is executed First. | Outer Query is executed first. |
| Inner query is executed only once and its result is used by outer query. | Inner query is executed for each of the records that outer query returns. |
| Uses using =, <, >, >=, <=, IN, BETWEEN operators. | Can use using =, <, >, >=, <=, IN, BETWEEN operators, but it mainly uses EXISTS and NOT EXISTS clause. |
| Always outer query columns are compared with inner query but there are no explicit joins in the inner query with outer query columns. | There should be some joins between the outer and inner query columns in the inner query. |
| Is always used in the WHERE clause. | Is used in WHERE clause as well as columns of SELECT statement. |
| Depending on the number of columns returned by inner query, operators should be used in the outer query. | There is no restrictions on the operators if EXISTS or NOT EXISTS are used. The condition inside the inner query should be correct. |
| Performance is better as inner query is executed only once and outer query is executed based on the result of inner query. | It will be bit slow if the outer table has large number of records. This is because, when each record of outer query is retrieved, the inner query is executed. The number of execution of inner query depends on the number of records returned by the outer query. |

(SELECT STD\_ID

FROM MARKS m

WHERE s.STD\_ID = m.STD\_ID);

We can use correlated sub queries as a column in the SELECT statement too. In below query AVG\_MARK is a column got from correlated sub query which finds the average mark of each student.

SELECT

STD\_ID,

STD\_NAME,

(SELECT AVG (mark)

FROM MARKS m

WHERE m.STD\_ID = s.STD\_ID) AS AVG\_MARK

FROM STUDENT s;

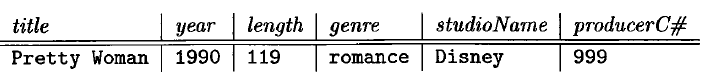
## 

The FROM clause gives the relation or relations to which the query refers. In our example, the query is about the relation Movies.

• The WHERE clause is a condition, much like a selection-condition in relational algebra. Tuples must satisfy the condition in order to match the query. Here, the condition is that the studioName attribute of the tuple has the value ’D isney’ and the year attribute of the tuple has the value 1990. All tuples meeting both stipulations satisfy the condition; other tuples do not.

• The SELECT clause tells which attributes of the tuples matching the condition are produced as part of the answer. The \* in this example indicates that the entire tuple is produced. The result of the query is the relation consisting of all tuples produced by this process.

One way to interpret this query is to consider each tuple of the relation mentioned in the FROM clause. The condition in the WHERE clause is applied to the tuple. More precisely, any attributes mentioned in the WHERE clause are replaced by the value in the tuple’s component for that attribute. The condition is then evaluated, and if true, the components appearing in the SELECT clause are produced as one tuple of the answer. Thus, the result of the query is the Movies tuples for those movies produced by Disney in 1990, for example, *Pretty Woman.*

In detail, when the SQL query processor encounters the Movies tuple****

(here, 999 is the imaginary certificate number for the producer of the movie), the value ’D isney’ is substituted for attribute studioName and value 1990 is substituted for attribute year in the condition of the WHERE clause, because these are the values for those attributes in the tuple in question. The WHERE clause thus becomes

WHERE ’Disney' = ’Disney’ AND 1990 = 1990

Since this condition is evidently true, the tuple for *Pretty Woman* passes the test of the WHERE clause and the tuple becomes part of the result of the query.

**Projection in SQL**

We can, if we wish, eliminate some of the components of the chosen tuples; that is, we can project the relation produced by a SQL query onto some of its attributes. In place of the \* of the SELECT clause, we may list some of the attributes of the relation mentioned in the FROM clause. The result will be projected onto the attributes listed.1

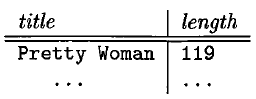
E x am p le: Suppose we wish to modify the query of Example to produce only the movie title and length. We may write

SELECT title, length

FROM Movies

WHERE studioName = ’Disney’ AND year = 1990;

The result is a table with two columns, headed title and length. The tuples in this table are pairs, each consisting of a movie title and its length, such that the movie was produced by Disney in 1990. For instance, the relation schema and one of its tuples looks like:

****

**Queries Involving More Than One Relation:**

Much of the power of relational algebra comes from its ability to combine two or more relations through joins, products, unions, intersections, and differences. We get all of these operations in SQL. The set-theoretic operations — union, intersection, and difference — appear directly in SQL, as we shall learn in Section 6.2.5. First, we shall learn how the select-from-where statement of SQL allows us to perform products and joins.

**Products and Joins in SQL**

SQL has a simple way to couple relations in one query: list each relation in the FROM clause. Then, the SELECT and WHERE clauses can refer to the attributes of any of the relations in the FROM clause.

Example: Suppose we want to know the name of the producer of *Star Wars.* To answer this question we need the following two relations from our running example:

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

MovieExec(name, address, cert#, netWorth)

The producer certificate number is given in the Movies relation, so we can do a simple query on Movies to get this number. We could then do a second query on the relation MovieExec to find the name of the person with that certificate number.

However, we can phrase both these steps as one query about the pair of relations Movies and MovieExec as follows:

SELECT name

FROM Movies, MovieExec

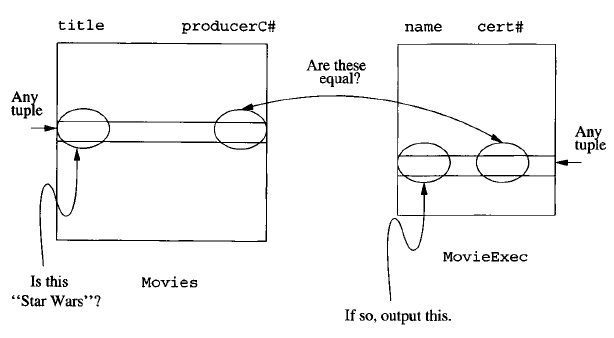
WHERE title = ’Star Weirs’ AND producerC# = cert#;

This query asks us to consider all pairs of tuples, one from Movies and the other from MovieExec. The conditions on this pair are stated in the WHERE clause:

1. The t i t l e component of the tuple from Movies must have value ’S ta r Wars’ .

2. The producerC# attribute of the Movies tuple must be the same certificate number as the c e rt# attribute in the MovieExec tuple. That is, these two tuples must refer to the same producer.

Whenever we find a pair of tuples satisfying both conditions, we produce the name attribute of the tuple from MovieExec as part of the answer. If the data is what we expect, the only time both conditions will be met is when the tuple from Movies is for *Star Wars,* and the tuple from MovieExec is for George Lucas. Then and only then will the title be correct and the certificate numbers agree.



**Disambiguating Attributes**

Sometimes we ask a query involving several relations, and among these relations are two or more attributes with the same name. If so, we need a way to indicate which of these attributes is meant by a use of their shared name. SQL solves this problem by allowing us to place a relation name and a dot in front of an attribute. Thus *R.A* refers to the attribute *A* of relation *R.*

Example: The two relations

MovieStar(name, address, gender, birthdate)

MovieExec(name, address, cert#, netWorth)

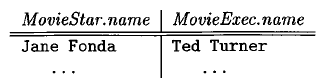
each have attributes name and address. Suppose we wish to find pairs consisting of a star and an executive with the same address. The following query does the job.

SELECT MovieStar.name, MovieExec.name

FROM MovieStar, MovieExec

WHERE MovieStar.address = MovieExec.address;

In this query, we look for a pair of tuples, one from MovieStar and the other from MovieExec, such that their address components agree. The WHERE clause enforces the requirement that the address attributes from each of the two tuples agree. Then, for each matching pair of tuples, we extract the two name attributes, first from the MovieStar tuple and then from the other. The result would be a set of pairs such as



The relation name, followed by a dot, is permissible even in situations where there is no ambiguity. For instance, we are free to write the query of Example as

SELECT MovieExec.name

FROM Movies, MovieExec

WHERE Movie.title = ’Star Wars’

AND Movie.producerC# = MovieExec.cert#;

Alternatively, we may use relation names and dots in front of any subset of the attributes in this query.

**Tuple Variables**

Disambiguating attributes by prefixing the relation name works as long as the query involves combining several different relations. However, sometimes we need to ask a query that involves two or more tuples from the same relation.

We may list a relation *R* as many times as we need to in the FROM clause, but we need a way to refer to each occurrence of *R.* SQL allows us to define, for each occurrence of *R* in the FROM clause, an “alias” which we shall refer to as a *tuple variable.* Each use of *R* in the FROM clause is followed by the (optional) keyword AS and the name of the tuple variable; we shall generally omit the AS in this context.

In the SELECT and WHERE clauses, we can disambiguate attributes of *R* by preceding them by the appropriate tuple variable and a dot. Thus, the tuple variable serves as another name for relation *R* and can be used in its place when

we wish for a star and an executive sharing an address, we might similarly want to know about two stars who share an address. The query is essentially the same, but now we must think of two tuples chosen from relation M ovieStar, rather than tuples from each of MovieStar and MovieExec. Using tuple variables as aliases for two uses of MovieStar, we can

write the query as

SELECT Starl.name, Star2.name

FROM MovieStar Starl, MovieStar Star2

WHERE Starl.address = Star2.address

AND Starl.name < Star2.name;

**Subqueries:**

In SQL, one query can be used in various ways to help in the evaluation of another. A query that is part of another is called a *subquery.* Subqueries can have subqueries, and so on, down as many levels as we wish. We already saw one example of the use of subqueries; built a union, intersection, or difference query by connecting two subqueries to form the whole query. There are a number of other ways that subqueries can be used:

1. Subqueries can return a single constant, and this constant can be compared with another value in a WHERE clause.

2. Subqueries can return relations that can be used in various ways in WHERE clauses.

3. Subqueries can appear in FROM clauses, followed by a tuple variable that represents the tuples in the result of the subquery.

**Subqueries that Produce Scalar Values**

An atomic value that can appear as one component of a tuple is referred to as a *scalar.* A select-from-where expression can produce a relation with any number of attributes in its schema, and there can be any number of tuples in the relation. However, often we are only interested in values of a single attribute.

Furthermore, sometimes we can deduce from information about keys, or from other information, that there will be only a single value produced for that attribute. If so, we can use this select-from-where expression, surrounded by parentheses, as if it were a constant. In particular, it may appear in a WHERE clause any place we would expect to find a constant or an attribute representing a component of a tuple. For instance, we may compare the result of such a subquery to a constant or attribute.

E xam ple: Let us recall Example 6.12, where we asked for the producer of *Star Wars.* We had to query the two relations

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

MovieExec(name, address, cert#, netWorth)

Because only the former has movie title information and only the latter has producer names. The information is linked by “certificate numbers.” These numbers uniquely identify producers. The query we developed is:

SELECT name

FROM Movies, MovieExec

WHERE title = ’Star Wars’ AND producerC# = cert#;

There is another way to look at this query. We need the Movies relation only to get the certificate number for the producer of *Star Wars.* Once we have it, we can query the relation MovieExec to find the name of the person with this certificate. The first problem, getting the certificate number, can be written as a subquery, and the result, which we expect will be a single value, can be used in the “main” query to achieve the same effect as the query above. Lines (4) through (6) of Fig. 6.6 are the subquery. Looking only at this simple query by itself, we see that the result will be a unary relation with

1) SELECT name

2) FROM MovieExec

3) WHERE cert# =

4) (SELECT producerC#

5) FROM Movies

6) WHERE title = ’Star Wars’

);

The tuple will look like (12345), that is, a single component with some integer, perhaps 12345 or whatever George Lucas’ certificate number is. If zero tuples or more than one tuple is produced by the subquery of lines (4) through (6), it is a run-time error. Having executed this subquery, we can then execute lines (1) through (3) of, as if the value 12345 replaced the entire subquery. That is, the “main” query is executed as if it were

SELECT name

FROM MovieExec

WHERE cert# = 12345;

**Conditions Involving Relations**

There are a number of SQL operators that we can apply to a relation *R* and produce a boolean result. However, the relation *R* must be expressed as a subquery. As a trick, if we want to apply these operators to a stored table Foo, we can use the subquery (SELECT \* FROM Foo). The same trick works for union, intersection, and difference of relations. Notice that those operators, are applied to two subqueries.

Some of the operators below — IN, ALL, and ANY — will be explained first in their simple form where a scalar value *s* is involved. In this situation, the subquery *R* is required to produce a one-column relation. Here are the definitions of the operators:

1. EXISTS *R* is a condition that is true if and only if *R* is not empty.

2. *s* IN *R* is true if and only if *s* is equal to one of the values in *R.* Likewise, *s* NOT IN *R* is true if and only if *s* is equal to no value in *R.*

Here, we assume *R* is a unary relation. We shall discuss extensions to the IN and NOT IN operators where *R* has more than one attribute in its schema and *s* is a tuple.

3. *s* > ALL *R* is true if and only if *s* is greater than every value in unary relation *R.* Similarly, the > operator could be replaced by any of the other five comparison operators, with the analogous meaning: *s* stands in the stated relationship to every tuple in *R.* For instance, *s* <> ALL *R* is

the same as *s* NOT IN *R.*

4. *s* > ANY *R* is true if and only if *s* is greater than at least one value in unary relation *R.* Similarly, any of the other five comparisons could be used in place of >, with the meaning that *s* stands in the stated relationship to at least one tuple of *R.* For instance, *s* = ANY *R* is the same as *s* IN *R.*

The EXISTS, ALL, and ANY operators can be negated by putting NOT in front of the entire expression, just like any other boolean-valued expression. Thus, NOT EXISTS *R* is true if and only if *R* is empty. NOT *s* >= ALL *R* is true if and only if *s* is not the maximum value in *R,* and NOT *s* > ANY *R* is true if and only if *s* is the minimum value in *R.* We shall see several examples of the use

of these operators shortly.

**Conditions Involving Tuples**

A tuple in SQL is represented by a parenthesized list of scalar values. Examples are (123, ’fo o ’) and (name, address, networth). The first of these has constants as components; the second has attributes as components. Mixing of constants and attributes is permitted.

If a tuple *t* has the same number of components as a relation *R,* then it makes sense to compare *t* and *R* in expressions.Examples are *t* IN *R* or *t* <> ANY *R.* The latter comparison means that there is some tuple in *R* other than *t.* Note that when comparing a tuple with members of a relation *R,* we must compare components using the assumed standard order for the attributes of *R.*

1) SELECT name

2) FROM MovieExec

3) WHERE cert# IN

4) (SELECT producerC#

5) FROM Movies

6) WHERE (title, year) IN

7) (SELECT movieTitle, movieYear

8) FROM Starsln

9) WHERE starName = ’Harrison Ford’));

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

Starsln(movieTitle, movieYear, starName)

MovieExec(name, address, cert#, netWorth)

asking for all the producers of movies in which Harrison Ford stars. It consists of a “main” query, a query nested within that, and a third query nested within the second.

We should analyze any query with subqueries from the inside out. Thus, let us start with the innermost nested subquery: lines (7) through (9). This query examines the tuples of the relation Starsln and finds all those tuples whose starName component is ’Harrison Ford’. The titles and years of those movies are returned by this subquery. Recall that title and year, not title alone, is the key for movies, so we need to produce tuples with both attributes to identify a movie uniquely.

**Natural Joins**

As we recall from Section 2.4.8, a natural join differs from a theta-join in that:

1. The join condition is that all pairs of attributes from the two relations

having a common name are equated, and there are no other conditions.

**2**. One of each pair of equated attributes is projected out.

The SQL natural join behaves exactly this way. Keywords NATURAL JOIN appear between the relations to express the tx operator.

**E xam p le**: Suppose we want to compute the natural join of the relations

MovieStar(name, ad d re ss, gender, b irth d a te )

MovieExec(name, ad d ress, c e rt# , netWorth)

The result will be a relation whose schema includes attributes name and address plus all the attributes that appear in one or the other of the two relations. A tuple of the result will represent an individual who is both a star and an executive and will have all the information pertinent to either: a name, address,gender, birthdate, certificate number, and net worth. The expression

MovieStar NATURAL JOIN MovieExec;

succinctly describes the desired relation.

**Full-Relation Operations:**

First, we deal with the fact that SQL uses relations that are bags rather than sets, and a tuple can appear more than once in a relation.SQL has aggregation operators and a GROUP-BY clause. There is also a “HAVING” clause that allows selection of certain groups in a way that depends on the group as a whole, rather than on individual tuples.

**Eliminating Duplicates:**

A relation, being a set, cannot have more than one copy of any given tuple. When a SQL query creates a new relation, the SQL system does not ordinarily eliminate duplicates.

If we want only to see each

producer once, we may change line (**1**) of the query to

1) SELECT DISTINCT name

Then, the list of producers will have duplicate occurrences of names eliminated

before printing.

Thus, George Lucas is printed only once.

**Grouping and Aggregation in SQL**

**Aggregation Operators:**

SQL uses the five aggregation operators SUM, AVG, MIN, MAX, and COUNT .These operators are used by applying them to a scalarvalued expression, typically a column name, in a SELECT clause. One exception is the expression C0UNT(\*), which counts all the tuples in the relation that is constructed from the FROM clause and WHERE clause of the query.

In addition, we have the option of eliminating duplicates from the column before applying the aggregation operator by using the keyword DISTINCT. That is, an expression such as COUNT (DISTINCT x) counts the number of distinct values in column *x.* We could use any of the other operators in place of COUNT here, but expressions such as SUM (DISTINCT x) rarely make sense, since it asks us to sum the different values in column *x.*

**Defining a Relation Schema in SQL:**

SQL (pronounced “sequel”) is the principal language used to describe and manipulate relational databases. There is a current standard for SQL, called SQL- 99. Most commercial database management systems implement something similar, but not identical to, the standard. There are two aspects to SQL:

1. The *Data-Definition* sublanguage for declaring database schemas and

2. The *Data-Manipulation* sublanguage for *querying* (asking questions about) databases and for modifying the database.

The distinction between these two sublanguages is found in most languages; e.g., C or Java have portions that declare data and other portions that are executable code. These correspond to data-definition and data-manipulation, respectively.

**Relations in SQL**

SQL makes a distinction between three kinds of relations:

1. Stored relations, which are called *tables.* These are the kind of relation we deal with ordinarily — a relation that exists in the database and that can be modified by changing its tuples, as well as queried.

2. *Views,* which are relations defined by a computation. These relations are not stored, but are constructed, in whole or in part, when needed.

The SQL CREATE TABLE statement declares the schema for a stored relation. It gives a name for the table, its attributes, and their data types. It also allows us to declare a key, or even several keys, for a relation. There are many other features to the CREATE TABLE statement, including many forms of constraints that can be declared, and the declaration of *indexes* (data structures that speed up many operations on the table) but we shall leave those for the appropriate time.

**Data Types:**

To begin, let us introduce the primitive data types that are supported by SQL systems. All attributes must have a data type.

1. Character strings of fixed or varying length. The type CHAR(n) denotes a fixed-length string of up to *n* characters.

VARCHAR(n) also denotes a string of up to *n* characters. The difference is implementation-dependent; typically CHAR implies that short strings are padded to make *n* characters, while VARCHAR implies that an endmarker or string-length is used. SQL permits reasonable coercions between values of character-string types.

2. Bit strings of fixed or varying length. These strings are analogous to fixed and varying-length character strings, but their values are strings of bits rather than characters. The type BIT (n) denotes bit strings of length *n,* while BIT VARYING (n) denotes bit strings of length up to *n.* 3. The type BOOLEAN denotes an attribute whose value is logical. The possible values of such an attribute are TRUE, FALSE, and — although it would

4. The type INT or INTEGER (these names are synonyms) denotes typical integer values. The type SHORTINT also denotes integers Floating-point numbers can be represented in a variety of ways A higher precision can be obtained with the type DOUBLE PRECISION; again the distinction between these types is as in C. SQL also has types that are real numbers with a fixed decimal point. For example, DECIMAL(n,d) allows values that consist of n decimal digits, with the decimal point assumed to be *d* positions from the right. Thus, 0123.45 is a possible value of type DECIMAL(6,2). NUMERIC is almost a synonym for DECIMAL, although there are possible implementation-dependent differences.

5. Dates and times can be represented by the data types DATE and TIME, respectively (see the box on “Dates and Times in SQL”).

**Simple Table Declarations**

The simplest form of declaration of a relation schema consists of the keywords CREATE TABLE followed by the name of the relation and a parenthesized, comma-separated list of the attribute names and their types.

E x am p le:

CREATE TABLE Movies (

t i t l e CHAR(IOO),

year INT,

len g th INT,

genre CHAR(10),

studioName CHAR(30),

producerC# INT

);

**Modifying Relation Schemas**

We can delete a relation *R* by the SQL statement:

DROP TABLE R;

Relation *R* is no longer part of the database schema, and we can no longer access any of its tuples.

**Database Modifications:**

To this point, we have focused on the normal SQL query form: the select-from where statement. There are a number of other statement forms that do not return a result, but rather change the state of the database.

1. Insert tuples into a relation.

2. Delete certain tuples from a relation.

3. Update values of certain components of certain existing tuples.

We refer to these three types of operations collectively as *modifications.*

**Insertion**

The basic form of insertion statement is:

INSERT INTO *R (A 1*, . . . , *A n)* VALUES (ui, . . . , *v*n);

A tuple is created using the value for attribute *Ai,* for \* = **1**,2 ,,.. , n. If the list of attributes does not include all attributes of the relation *R,* then the tuple created has default values for all missing attributes.

E x am p le

1) INSERT INTO Starsln(movieTitle, movieYear, starName)

2) VALUES(’The Maltese Falcon’, 1942, ’Sydney Greenstreet’);

The effect of executing this statement is that a tuple with the three components on line (2) is inserted into the relation S ta rs ln . Since all attributes of S ta rs ln are mentioned on line (1), there is no need to add default components.

**Deletion**

The form of a deletion is

DELETE FROM *R* WHERE <condition> ;

We can delete from relation S ta rsln (m o v ie T itle , movieYear, starName) the fact that Sydney Greenstreet was a star in *The Maltese Falcon* by the SQL statement:

DELETE FROM Starsln

WHERE m ovieT itle = ’The Maltese Falcon ’ AND

movieYear = 1942 AND

starName = ‘Sydney Greenstreet’ ;

**Updates**

An *update* in SQL is a very specific kind of change to the database: one or more tuples that already exist in the database have some of their components changed. The general form of an update statement is:

UPDATE *R* SET <new-value assignments> WHERE <condition>;

E x am p le

MovieExec(name, ad d ress, c e rt# , netWorth)

by attaching the title P res, in front of the name of every movie executive who is the president of a studio. The condition the desired tuples satisfy is that their certificate numbers appear in the presC# component of some tuple in the Studio relation. We express this update as:

1) UPDATE MovieExec

2) SET name = ’P re s. ’ I I name

3) WHERE c e rt# IN (SELECT presC# FROM S tu d io );

**Defining Views:**

Views are relations, except that they are not physically stored. For presenting different information to different users Employee(ssn, name, department, project, salary);

CREATE VIEW Developers AS

SELECT name, project

FROM Employee

WHERE department = “Development”

Payroll has access to Employee, others only to Developers

**Types of Views**

* Virtual views:
  + Used in databases
  + Computed only on-demand – slow*er* at runtime
  + Always up to date
* Materialized views
  + Used in data warehouses
  + Precomputed offline – fast*er* at runtime
  + May have stale data

**Transactions in SQL**

To this point, our model of operations on the database has been that of one user querying or modifying the database. Thus, operations on the database are executed one at a time, and the database state left by one operation is the state upon which the next operation acts. Moreover, we imagine that operations are carried out in their entirety (“atomically”). That is, we assumed it is impossible for the hardware or software to fail in the middle of a modification, leaving the database in a state that cannot be explained as the result of the operations performed on it.

Real life is often considerably more complicated. We shall first consider what can happen to leave the database in a state that doesn’t reflect the operations performed on it, and then we shall consider the tools SQL gives the user to assure that these problems do not occur.

**Serializability**

In applications like Web services, banking, or airline reservations, hundreds of operations per second may be performed on the database. The operations initiate at any of thousands or millions of sites, such as desktop computers or automatic teller machines. It is entirely possible that we could have two operations affecting the same bank account or flight, and for those operations to overlap in time. If so, they might interact in strange ways.

Here is an example of what could go wrong if the DBMS were completely unconstrained as to the order in which it operated upon the database. This example involves a database interacting with people, and it is intended to illustrate why it is important to control the sequences in which interacting events can occur. However, a DBMS would not control events that were so “large” that they involved waiting for a user to make a choice. The event sequences controlled by the DBMS involve only the execution of SQL statements.

Example: The typical airline gives customers a Web interface where they can choose a seat for their flight. This interface shows a map of available seats, and the data for this map is obtained from the airline’s database. There might be a relation such as:

Flights(fltNo, fltDate, seatNo, seatStatus) upon which we can issue the query:

SELECT seatNo

FROM Flights

WHERE fltNo = 123 AND fltDate = DATE ’2008-12-25’

AND seatStatus = ’available’;

The flight number and date are example data, which would in fact be obtained from previous interactions with the customer. When the customer clicks on an empty seat, say 22A, that seat is reserved for them. The database is modified by an update-statement, such as:

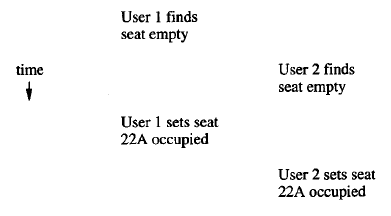
UPDATE Flights

SET seatStatus = ’occupied’

WHERE fltNo = 123 AND fltDate = DATE ’2008-12-25’

AND seatNo = ’22A’;

However, this customer may not be the only one reserving a seat on flight 123 on Dec. 25, 2008 and this exact moment. Another customer may have asked for the seat map at the same time, in which case they also see seat 22A empty. Should they also choose seat 22A, they too believe they have reserved 22A.



**Atomicity:**

In addition to non serialized behavior that can occur if two or more database operations are performed about the same time, it is possible for a single operation to put the database in an unacceptable state if there is a hardware or software “crash” while the operation is executing. Here is another example suggesting what might occur. As in Example 6.40, we should remember that real database systems do not allow this sort of error to occur in properly designed application programs.

**Transactions:**

The solution to the problems of serialization and atomicity posed in Sections 6.6.1 and 6.6.2 is to group database operations into *transactions.* A transaction is a collection of one or more operations on the database that must be executed atomically; that is, either all operations are performed or none are. In addition, SQL requires that, as a default, transactions are executed in a serializable manner.

A DBMS may allow the user to specify a less stringent constraint on the interleaving of operations from two or more transactions. SQL allows the programmer to group several statements into a single transaction.

The SQL command START TRANSACTION is used to mark the beginning of a transaction. There are two ways to end a transaction:

1. The SQL statement COMMIT causes the transaction to end successfully. Whatever changes to the database were caused by the SQL statement or statements since the current transaction began are installed permanently in the database (i.e., they are *committed).* Before the COMMIT statement is executed, changes are tentative and may or may not be visible to other

transactions.

2. The SQL statement ROLLBACK causes the transaction to *abort,* or terminate unsuccessfully. Any changes made in response to the SQL statements of the transaction are undone (i.e., they are *rolled back),* so they never permanently appear in the database.

**Read-Only Transactions:**

If two executions of the function tried to book the same seat at the same time, what could happen if there was a crash in the middle of a funds transfer. However, when a transaction only reads data and does not write data, we have more freedom to let the transaction execute in parallel with other transactions.

**Dirty Reads**

*Dirty data* is a common term for data written by a transaction that has not yet committed. A *dirty read* is a read of dirty data written by another transaction. The risk in reading dirty data is that the transaction that wrote it may eventually abort. If so, then the dirty data will be removed from the database, and the world is supposed to behave as if that data never existed. If some other transaction has read the dirty data, then that transaction might commit or take some other action that reflects its knowledge of the dirty data. Sometimes the dirty read matters, and sometimes it doesn’t. Other times it matters little enough that it makes sense to risk an occasional dirty read and thus avoid:

1. The time-consuming work by the DBMS that is needed to prevent dirty reads, and

2. The loss of parallelism that results from waiting until there is no possibility of a dirty read.

Here are some examples of what might happen when dirty reads are allowed.

E x am p le : Let us reconsider the account transfer of Example 6.41. However, suppose that transfers are implemented by a program *P* that executes the following sequence of steps:

1. Add money to account 2.

2. Test if account 1 has enough money.

(a) If there is not enough money, remove the money from account 2 and end**.**

(b) If there is enough money, subtract the money from account 1 and end.

If program *P* is executed serializably, then it doesn’t m atter that we have put money temporarily into account 2. No one will see that money, and it gets removed if the transfer can’t be made.

However, suppose dirty reads are possible. Imagine there are three accounts:

*A I, A2,* and ^43, with $100, $200, and $300, respectively.

**Other Isolation Levels** :

SQL provides a total of four isolation levels.

Two of them we have already seen:

1.Serializable

2.Read-uncommitted (dirty reads allowed).

3.Read-committed and

4.Repeatable-read.

They can be specified for a given transaction by SET TRANSACTION ISOLATION LEVEL READ COMMITTED; or SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; respectively. For each, the default is that transactions are read-write, so we can add READ ONLY to either statement, if appropriate. Incidentally, we also have the option of specifying SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

The read-committed isolation level, as its name implies, forbids the reading of dirty (uncommitted) data. However, it does allow a transaction running at this isolation level to issue the same query several times and get different answers, as long as the answers reflect data that has been written by transactions that already committed.

E xam ple: Let us reconsider the seat-choosing program but suppose we declare it to run with isolation level read-committed. Then when it searches for a seat at Step (1), it will not see seats as booked if some other transaction is reserving them but not committed.8 However, if the traveler rejects seats, and one execution of the function queries for available seats many times, it may see a different set of available seats each time it queries, as other transactions successfully book seats or cancel seats in parallel with our transactions.

**Constraints and Triggers**

**Keys and Foreign keys**

**Declaring Foreign-Key Constraints**

A foreign key constraint is an assertion that values for certain attributes must make sense. Recall, for instance, that in Example 2.21 we considered 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.

1. 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
2. REFERENCES <table> (< attribute> )

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> )**

whose primary key is name and which has a foreign key presC# that references c e rt# of relation

MovieExec(name, ad d ress, c e rt# , netWorth)

We may declare presC# directly to reference c e rt# as follows:

CREATE TABLE Studio (

name CHAR(30) PRIMARY KEY,

address VARCHAR(255),

presC# INT REFERENCES M ovieExec(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 M ovieExec(cert#)

);

Notice that the referenced attribute, c e rt# 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 S tudio tuple, that value must also appear in the c e rt# component of some MovieExec tuple. The one exception is that, should a particular S tudio tuple have NULL as the value of its presC# component, there is no requirement that NULL appear as the value of a c e rt# component (but note that c e rt# is a primary key and therefore cannot have NULL’s anyway).

**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.

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 7.2.2 we cover the principal form of constraints of type (1): *attribute-based* CHECK *constraints.* The second type, the tuple-based constraints, are covered.

There are other, more general kinds of constraints that we shall meet in Sections 7.4 and 7.5. 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.

**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.

**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*

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

MovieExec(name, ad d ress, c e rt# , netWorth)

It is triggered by updates to the netW orth attribute. The effect of this trigger is to foil any attem pt to lower the net worth of a movie executive.

1) CREATE TRIGGER NetW orthTrigger

2) AFTER UPDATE OF netW orth ON MovieExec

3) REFERENCING

4) OLD ROW AS OldTuple,

5) NEW ROW AS NewTuple

6) FOR EACH ROW

7) WHEN (OldTuple.netW orth > NewTuple.netWorth)

8) UPDATE MovieExec

9) SET netWorth = OldTuple.netW orth

10) WHERE c e rt# = NewTuple. c e r t# ;

**Schema-Level Constraints and Triggers:**

The most powerful forms of active elements in SQL are not associated with specific tuples or components of tuples. These elements, called "triggers" and "assertions," are part of the database schema, on a par with the relation -ns 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 just require the programmer to state what must be true, triggers are the feature DBMS's usually 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.

**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>)**

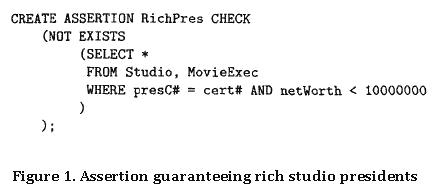
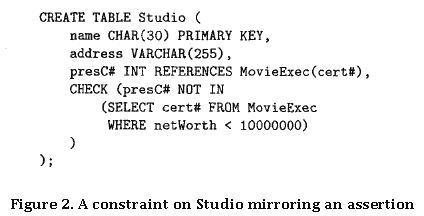
The condition in an assertion must be true when the assertion is created and must always remain true; any database modification whatsoever that causes it to become false will be rejected. Remember that the other types of CHECK constraints we have covered can be violated under certain conditions, if they involve sub queries. 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 example, in line (6) of "Tuple-Based CHECK Constraints" Figure 1 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, usually 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 instance, 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 example, this way we could require that a sum always be less than some limiting value.

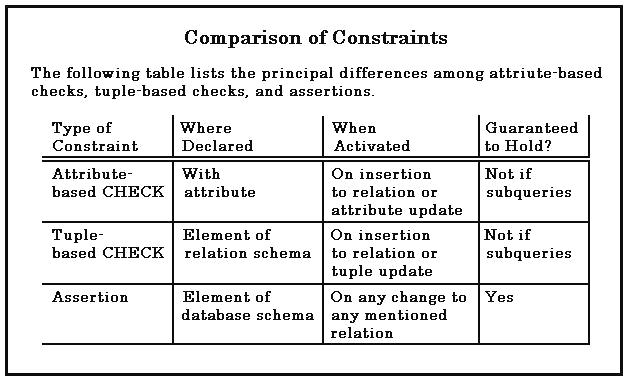
**Example 1 :** Assume we wish to require that no one can become the president of a studio unless their  net worth is at least $10,000,000. We declare an assertion to the effect that the set of movie studios with presidents having a net worth less than $10,000,000 is empty. This assertion involves the two relations

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

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

  
The assertion is shown in Figure 1.  
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 example, we can add to the CREATE TABLE statement of "Declaring Foreign-Key Constraints" Example 1 a constraint on Studio as shown in Figure 2.  
  
  
 Note, nevertheless, that the constraint of Figure 2 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 2 :** Here is another instance of an assertion. It involves the relation  
  
**Movie(title, year, length, inColor, 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(1ength) 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(1ength) FROM Movie GROUP BY studioName));**  
 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

**DROP ASSERTION <assertion name>**

**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 "Deferring the Checking of Constraints" and cover them with more detail in "Transactions in SQL").

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.