Module 3: Introduction to SQL

Overview of the SQL Query Language

IBM developed the original version of SQL, originally called Sequel, as part of the System R project in the early 1970s. The Sequel language has evolved since then, and its name has changed to SQL (Structured Query Language).

The SQL language has several parts:

- **Data-definition language** (DDL). The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.
- **Data-manipulation language** (DML). The SQL DML provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
- **Integrity**. The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.
- **View definition**. The SQL DDL includes commands for defining views.
- **Transaction control**. SQL includes commands for specifying the beginning and ending of transactions.
- **Embedded SQL** and **dynamic SQL**. Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, and Java.
- **Authorization**. The SQL DDL includes commands for specifying access rights to relations and views.

SQL Data Definition

The set of relations in a database must be specified to the system by means of a data-definition language (DDL).

• The schema for each relation.

• The types of values associated with each attribute.

• The integrity constraints.

• The set of indices to be maintained for each relation.

• The security and authorization information for each relation.
• The physical storage structure of each relation on disk.

Basic Types

The SQL standard supports a variety of built-in types, including:

• **char**(*n*): A fixed-length character string with user-specified length *n*. The full form, **character**, can be used instead.

• **varchar**(*n*): A variable-length character string with user-specified maximum length *n*. The full form, **character varying**, is equivalent.

• **int**: An integer (a finite subset of the integers that is machine dependent). The full form, **integer**, is equivalent.

• **small int**: A small integer (a machine-dependent subset of the integer type).

• **numeric**(*p, d*):A fixed-point number with user-specified precision. The number consists of *p* digits (plus a sign), and *d* of the *p* digits are to the right of the decimal point. Thus, **numeric**(3,1) allows 44.5 to be stored exactly, but neither 444.5 or 0.32 can be stored exactly in a field of this type.

• **real, double precision**: Floating-point and double-precision floating-point numbers with machine-dependent precision.

• **float**(*n*): A floating-point number, with precision of at least *n* digits.

The **char** data type stores fixed length strings. Consider, for example, an attribute *A* of type **char**(10). If we store a string "Avi" in this attribute, 7 spaces are appended to the string to make it 10 characters long. In contrast, if attribute *B* were of type **varchar**(10), and we store "Avi" in attribute *B*, no spaces would be added.

Basic Schema Definition

Create table: We define an SQL relation by using the **create table** command. The following command creates a relation *department* in the database.

create table department
(dept name varchar (20),
building varchar (15),
budget numeric (12,2),
primary key (dept name));

The relation created above has three attributes, *dept name*, which is a character string of maximum length 20, *building*, which is a character string of maximum length 15, and *budget*, which is a number with 12 digits in total, 2 of which are after the decimal point. The **create table** command also specifies that the *dept name* attribute is the primary key of the *department* relation.

The general form of the **create table** command is:

create table *r*(*A*1 *D*1,*A*2 *D*2,. . . ,*An Dn*,
_integrity-constraint1
_,
. . . ,
_integrity-constraint *k*
_);

where *r* is the name of the relation, each *Ai* is the name of an attribute in the schema of relation *r*, and *Di* is the domain of attribute *Ai*; that is, *Di* specifies the type of attribute *Ai* along with optional constraints that restrict the set of allowed values for *Ai* .
The semicolon shown at the end of the **create table** statements is terminating character.

Constraints:

• **Primary key** (*Aj*1 , *Aj*2, . . . , *Ajm*): The **primary-key** specification says that attributes *Aj*1 , *Aj*2, . . . , *Ajm* form the primary key for the relation. The primary key attributes are required to be *nonnull* and *unique*; that is, no tuple can have a null value for a primary-key attribute, and no two tuples in the relation can be equal on all the primary-key attributes. Although the primary-key specification is optional, it is generally a good idea to specify a primary key for each relation.

• **Foreign key** (*Ak*1 , *Ak*2, . . . , *Akn*) **references** *s*: The **foreign key** specification says that the values of attributes (*Ak*1 , *Ak*2, . . . , *Akn*) for any tuple in the relation must correspond to values of the primary key attributes of some tuple in relation *s*.

Figure below presents a partial SQL DDL definition of the university database we use in the text. The definition of the *course* table has a declaration "**foreign key** (*dept name*) **references** *department*". This foreign-key declaration specifies that for each course tuple, the department name specified in the tuple must exist in the primary key attribute (*dept name*) of the *department* relation. Without this constraint, it is possible for a course to specify a nonexistent

department name. Figure below also shows foreign key constraints on tables *section*, *instructor* and *teaches*.

• **Not null**: The **not null** constraint on an attribute specifies that the null value is not allowed for that attribute; in other words, the constraint excludes the null value from the domain of that attribute. For example, in Figure 3.1, the **not null** constraint on the *name* attribute of the *instructor* relation ensures that the name of an instructor cannot be null.

Basic Structure of SQL Queries

SELECT---FROM--- WHERE structure:

The basic structure of an SQL query consists of three clauses: **select**, **from**, and **where**. The query takes as its input the relations listed in the **from** clause, operates on them as specified in the **where** and **select** clauses, and then produces a relation as the result.

Queries on a Single Relation

Ex: "Find the names of all instructors."

Instructor names are found in the *instructor* relation, so we put that relation in the **from** clause. The instructor's name appears in the *name* attribute, so we put that in the **select** clause.

select *name* **from** *instructor*;

- **Distinct:** In those cases where we want to force the elimination of duplicates, we insert the keyword **distinct** after **select**. We can rewrite the preceding query as:

select distinct *dept name* **from** *instructor*;

SQL allows us to use the keyword **all** to specify explicitly that duplicates are not removed:

select all *dept name* **from** *instructor*;

- **Arithmetic expressions**: The **select** clause may also contain arithmetic expressions involving the operators +, −, ∗, and / operating on constants or attributes of tuples. For example, the query:

select *ID*, *name*, *dept name*, *salary* * 1.1 **from** *instructor*;

returns a relation that is the same as the *instructor* relation, except that the attribute *salary* is multiplied by 1.1. This shows what would result if we gave a 10% raise to each instructor; note, however, that it does not result in any change to the *instructor* relation.

Ex: "Find the names of all instructors in the Computer Science department who have salary greater than $70,000." This query can be written in SQL as:

select *name* **from** *instructor***where** *dept name* = 'Comp. Sci.' **and** *salary* > 70000;

Logical and relational operators: SQL allows the use of the logical connectives **and**, **or**, and **not** in the **where** clause. The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>. SQL allows us to use the comparison operators to compare strings and arithmetic expressions, as well as special types, such as date types.

Queries on Multiple Relations

An an example, suppose we want to answer the query "Retrieve the names of all instructors, along with their department names and department building name."

In SQL, to answer the above query,we list the relations that need to be accessed in the **from** clause, and specify the matching condition in the **where** clause. The above query can be written in SQL as

select *name, instructor.dept name, building*
from *instructor, department*
where *instructor.dept name= department.dept name*;

We now consider the general case of SQL queries involving multiple relations. As we have seen earlier, an SQL query can contain three types of clauses, the **select** clause, the **from** clause, and the **where** clause. The role of each clause is as follows:

• The **select** clause is used to list the attributes desired in the result of a query.
• The **from** clause is a list of the relations to be accessed in the evaluation of the query.
• The **where** clause is a predicate involving attributes of the relation in the **from** clause.

A typical SQL query has the form

select $A1, A2, \ldots, An$
from $r1, r2, \ldots, rm$
where P;

If we only wished to find instructor names and course identifiers for instructors in the Computer Science department, we could add an extra predicate to the **where** clause, as shown below.

select *name, course id*

from *instructor*, *teaches*
where *instructor*.*ID*= *teaches*.*ID* **and** *instructor*.*dept name* = 'Comp. Sci.';

Note that since the *dept name* attribute occurs only in the *instructor* relation, we could have used just *dept name*, instead of *instructor*.*dept name* in the above query.

1. Generate a Cartesian product of the relations listed in the **from** clause

2. Apply the predicates specified in the **where** clause on the result of Step 1.

3. For each tuple in the result of Step 2, output the attributes (or results of expressions) specified in the **select** clause.

The Natural Join

To make the life of an SQL programmer easier for this common case, SQL supports an operation called the *natural join*, which we describe below. In fact SQL supports several other ways in which information from two or more relations can be **joined** together.

Unlike the Cartesian product of two relations, which concatenates each tuple of the first relation with every tuple of the second, natural join considers only those pairs of tuples with the same value on those attributes that appear in the schemas of both relations.

- Consider the query "For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught", which we wrote earlier as:

select *name*, *course id*
from *instructor*, *teaches*
where *instructor*.*ID*= *teaches*.*ID*;

This query can be written more concisely using the natural-join operation in SQL as:

select *name*, *course id* **from** *instructor* **natural join** *teaches*;

A **from** clause in an SQL query can have multiple relations combined using natural join, as shown here:

select *A*1, *A*2, . . . , *An*
from *r*1 **natural join** *r*2 **natural join** . . . **natural join** *rm*
where *P*;

More generally, a **from** clause can be of the form

from *E*1, *E*2, . . . , *En*

where each *Ei* can be a single relation or an expression involving natural joins. For example, suppose we wish to answer the query "List the names of instructors along with the the titles of courses that they teach." The query can be written in SQL as follows:

select *name, title* **from** *instructor* **natural join** *teaches, course*
where *teaches.course id= course.course id*;

In contrast the following SQL query does *not* compute the same result:

select *name, title*
from *instructor* **natural join** *teaches* **natural join** *course*;

To see why, note that the natural join of *instructor* and *teaches* contains the attributes (*ID, name, dept name, salary, course id, sec id*), while the *course* relation contains the attributes (*course id, title, dept name, credits*). As a result, the natural join of these two would require that the *dept name* attribute values from the two inputs be the same, in addition to requiring that the *course id* values be the same.

Additional Basic Operations

There are number of additional basic operations that are supported in SQL.

3.4.1 The Rename Operation

Consider again the query that we used earlier:

select *name, course id*
from *instructor, teaches*
where *instructor.ID= teaches.ID*;

The result of this query is a relation with the following attributes: *name, course id*

The names of the attributes in the result are derived from the names of the attributes in the relations in the **from** clause.

We cannot, however, always derive names in this way, for several reasons:

- First, two relations in the **from** clause may have attributes with the same name, in which case an attribute name is duplicated in the result.
- Second, if we used an arithmetic expression in the **select** clause, the resultant attribute does not have a name.
- Third, even if an attribute name can be derived from the base relations as in the preceding example, we may want to change the attribute name in the result. Hence, SQL provides a way of renaming the attributes of a result relation.

Aliasing (as): It uses the **as** clause, taking the form: *old-name* **as** *new-name*
For example, if we want the attribute name *name* to be replaced with the name *instructor name*, we can rewrite the preceding query as:

select *name* **as** *instructor name*, *course id* **from** *instructor*, *teaches* **where** *instructor.ID= teaches.ID*;

- "For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught."

select *T.name*, *S.course id*
from *instructor* **as** *T, teaches* **as** *S*
where *T.ID= S.ID*;

- "Find the names of all instructors whose salary is greater than at least one instructor in the Biology department."

select distinct *T.name*
from *instructor* **as** *T, instructor* **as** *S*
where *T.salary > S.salary* **and** *S.dept name* = 'Biology';

String Operations(LIKE operator)
SQL specifies strings by enclosing them in single quotes, for example, 'Computer'. A single quote character that is part of a string can be specified by using two single quote characters;

The SQL standard specifies that the equality operation on strings is case sensitive; as a result the expression "'comp. sci.' = 'Comp. Sci.'" evaluates to false.

SQL also permits a variety of functions on character strings, such as concatenating (using "_"), extracting substrings, finding the length of strings, converting strings to uppercase (using the function **upper**(*s*) where *s* is a string) and lowercase (using the function **lower**(*s*)), removing spaces at the end of the string (using **trim**(*s*)) and so on.
Pattern matching can be performed on strings, using the operator **like**. We describe patterns by using two special characters:
• **Percent (%):** The % character matches any substring.
• **Underscore (_):** The character matches any character.

Patterns are case sensitive; that is, uppercase characters do not match lowercase characters, or vice versa. To illustrate pattern matching, we consider the following examples:

• 'Intro%' matches any string beginning with "Intro".

• '%Comp%' matches any string containing "Comp" as a substring,

for example, 'Intro. to Computer Science', and 'Computational Biology'.

•___' matches any string of exactly three characters.

• ___%' matches any string of at least three characters.

SQL expresses patterns by using the **like** comparison operator. Consider the query "Find the names of all departments whose building name includes the substring 'Watson'." This query can be written as:

select *dept name*
from *department*
where *building* **like** '%Watson%';

Use of Asterisk(*):

The asterisk symbol " * " can be used in the **select** clause to denote "all attributes." Thus, the use of *instructor*.* in the **select** clause of the query:

select *instructor*.*
from *instructor*, *teaches*
where *instructor.ID= teaches.ID*;

Ordering the Display of Tuples

SQL offers the user some control over the order in which tuples in a relation are displayed. The **order by** clause causes the tuples in the result of a query to appear in sorted order. To list in alphabetic order all instructors in the Physics department, we write:

select *name* **from** *instructor* **where** *dept name* = 'Physics'
order by *name*;

By default, the **order by** clause lists items in ascending order. To specify the sort order, we may specify **desc** for descending order or **asc** for ascending order. Furthermore, ordering can be performed on multiple attributes. Suppose that we wish to list the entire *instructor* relation

in descending order of *salary*. If several instructors have the same salary, we order them in ascending order by name. We express this query in SQL as follows:

select *
from *instructor*
order by *salary* **desc**, *name* **asc**;

Where Clause Predicates

Between: SQL includes a **between** comparison operator to simplify **where** clauses that specify that a value be less than or equal to some value and greater than or equal to some other value. Similarly, we can use the **not between** comparison operator.

- If we wish to find the names of instructors with salary amounts between $90,000 and $100,000, we can use the **between** comparison to write:

  select *name* **from** *instructor* **where** *salary* **between** 90000 **and** 100000;

- "Find the instructor names and the courses they taught for all instructors in the Biology department who have taught some course."

To write this query, we can modify either of the SQL queries we saw earlier, by adding an extra condition in the **where** clause.

select *name, course id*
from *instructor, teaches*
where *instructor.ID= teaches.ID* **and** *dept name* = 'Biology';

SQL permits us to use the notation (*v*1, *v*2, . . . , *vn*) to denote a tuple of arity *n* containing values *v*1, *v*2, . . . , *vn*. The comparison operators can be used on tuples, and the ordering is defined lexicographically. For example, (*a*1, *a*2) <= (*b*1, *b*2) is true if *a*1 <= *b*1 **and** *a*2 <= *b*2; similarly, the two tuples are equal if all their attributes are equal. Thus, the preceding SQL query can be rewritten as follows:5

select *name, course id*
from *instructor, teaches*
where (*instructor.ID, dept name*) = (*teaches.ID*, 'Biology');

Null Values

Null values present special problems in relational operations, including arithmetic operations, comparison operations, and set operations.

The result of an arithmetic expression (involving, for example +, −, ∗, or /) is null if any of the input values is null. For example, if a query has an expression *r.A*+ 5, and *r.A*is null for a particular tuple, then the expression result must also be null for that tuple.

Comparisons involving nulls are more of a problem. For example, consider the comparison "1 < **null**". It would be wrong to say this is true since we do not know what the null value represents. But it would likewise be wrong to claim this expression is false; if we did, "**not** (1 < **null**)" would evaluate to true, which does not make sense. SQL therefore treats as **unknown** the result of any comparison involving a *null* value (other than predicates **is null** and **is not null**, which are described later in this section). This creates a third logical value in addition to *true* and *false*.

Since the predicate in a **where** clause can involve Boolean operations such as **and**, **or**, and **not** on the results of comparisons, the definitions of the Boolean operations are extended to deal with the value **unknown**.

- **and**: The result of *true* **and** *unknown* is *unknown*, *false* **and** *unknown* is *false*, while *unknown* **and** *unknown* is *unknown*.
- **or**: The result of *true* **or** *unknown* is *true*, *false* **or** *unknown* is *unknown*, while *unknown* **or** *unknown* is *unknown*.
- **not**: The result of **not** *unknown* is *unknown*.

You can verify that if *r.A* is null, then "1 < *r.A*" as well as "**not** (1 < *r.A*)" evaluate to unknown.

If the **where** clause predicate evaluates to either **false** or **unknown** for a tuple, that tuple is not added to the result.

SQL uses the special keyword **null** in a predicate to test for a null value. Thus, to find all instructors who appear in the *instructor* relation with null values for *salary*, we write:

select *name*
from *instructor*
where *salary* **is null**;

The predicate **is not null** succeeds if the value on which it is applied is not null.

Aggregate Functions

Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five built-in aggregate functions:

• Average: **avg**

• Minimum: **min**

• Maximum: **max**

• Total: **sum**

• Count: **count**

The input to **sum** and **avg** must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well.

Basic Aggregation

Consider the query "Find the average salary of instructors in the Computer Science department."We write this query as follows:

select avg (*salary*)
from *instructor*
where *dept name*= 'Comp. Sci.';

The result of this query is a relation with a single attribute, containing a single tuple with a numerical value corresponding to the average salary of instructors in the Computer Science department. The database system may give an arbitrary name to the result relation attribute that is generated by aggregation; however, we can give a meaningful name to the attribute by using the **as** clause as follows:

select avg (*salary*) **as** *avg salary*
from *instructor*
where *dept name*= 'Comp. Sci.';

In the *instructor* relation of Figure 2.1, the salaries in the Computer Science department are $75,000, $65,000, and $92,000. The average balance is $232,000/3 = $77,333.33.

The required information is contained in the relation *teaches*, and we write this query as follows:

select count (**distinct** *ID*)
from *teaches*
where *semester* = 'Spring' **and** *year* = 2010;

Because of the keyword **distinct** preceding *ID*, even if an instructor teaches more than one course, she is counted only once in the result. We use the aggregate function **count** frequently to count the number of tuples in a relation. The notation for this function in SQL is **count** (*). Thus, to find the number of tuples in the *course* relation, we write

select count (*) **from** *course*;

Group By : Aggregation with Grouping

As an illustration, consider the query "Find the average salary in each department."

We write this query as follows:

select *dept name*, **avg** (*salary*) **as** *avg salary*
from *instructor*
group by *dept name*;

- "Find the number of instructors in each department who teach a course in the Spring 2010 semester."

However, this information has to be joined with information from the *instructor* relation to get the department name of each instructor. Thus, we write this query as follows:

select *dept name*, **count** (**distinct** *ID*) **as** *instr count*
from *instructor* **natural join** *teaches*
where *semester* = 'Spring' **and** *year* = 2010
group by *dept name*;

dept_name	*instr_count*
Comp. Sci.	3
Finance	1
History	1
Music	1

When an SQL query uses grouping, it is important to ensure that the only attributes that appear in the **select** statement without being aggregated are those that are present in the **group by** clause.

For example, the following query is erroneous since *ID* does not appear in the **group by** clause, and yet it appears in the **select** clause without being aggregated:

select *dept name*, *ID*, **avg** (*salary*) **from** *instructor* **group by** *dept name*;

Each instructor in a particular group (defined by *dept name*) can have a different *ID*, and since only one tuple is output for each group, there is no unique way of choosing which *ID* value to output. As a result, such cases are disallowed by SQL.

The Having Clause

At times, it is useful to state a condition that applies to groups rather than to tuples. For example, we might be interested in only those departments where the average salary of the instructors is more than $42,000. This condition does not apply to a single tuple; rather, it applies to each group constructed by the **group by** clause. To express such a query, we use the **having** clause of SQL. SQL applies predicates in the **having** clause after groups have been formed, so aggregate functions may be used.We express this query in SQL as follows:

select *dept name*, **avg** (*salary*) **as** *avg salary*
from *instructor*
group by *dept name*
having avg (*salary*) > 42000;

dept_name	*avg(avg_salary)*
Physics	91000
Elec. Eng.	80000
Finance	85000
Comp. Sci.	77333
Biology	72000
History	61000

The meaning of a query containing aggregation, **group by**, or **having** clauses is defined by the following sequence of operations:

1. As was the case for queries without aggregation, the **from** clause is first evaluated to get a relation.

2. If a **where** clause is present, the predicate in the **where** clause is applied on the result relation of the **from** clause.

3. Tuples satisfying the **where** predicate are then placed into groups by the **group by** clause if it is present. If the **group by** clause is absent, the entire set of tuples satisfying the **where** predicate is treated as being in one group.

4. The **having** clause, if it is present, is applied to each group; the groups that do not satisfy the **having** clause predicate are removed.

5. The **select** clause uses the remaining groups to generate tuples of the result of the query, applying the aggregate functions to get a single result tuple for each group.

To illustrate the use of both a **having** clause and a **where** clause in the same query, we consider the query

- "For each course section offered in 2009, find the average total credits (*tot cred*) of all students enrolled in the section, if the section had at least 2 students."

select *course id*, *semester*, *year*, *sec id*, **avg** (*tot cred*)
from *takes* **natural join** *student*
where *year* = 2009
group by *course id*, *semester*, *year*, *sec id*
having count (*ID*) >= 2;

Nested Sub queries:

SQL provides a mechanism for nesting subqueries. A subquery is a **select-from where** expression that is nested within another query. A common use of subqueries is to perform tests for set membership, make set comparisons, and determine set cardinality, by nesting subqueries in the **where** clause.

Set Membership

SQL allows testing tuples for membership in a relation. The ***in*** connective tests for set membership, where the set is a collection of values produced by a **select** clause. The **not in** connective tests for the absence of set membership.

"Find all the courses taught in the both the Fall 2009 and Spring 2010 semesters."

select distinct *course id* **from** *section* **where** *semester* = 'Fall' **and** *year*= 2009 **and** *course id* **in** (**select** *course id* **from** *section* **where** *semester* = 'Spring' **and** *year*= 2010);

We use the **not in** construct in a way similar to the **in** construct. For example, to find all the courses taught in the Fall 2009 semester but not in the Spring 2010 semester, we can write:

select distinct *course id* **from** *section* **where** *semester* = 'Fall' **and** *year*= 2009 **and** *course id* **not in** (**select** *course id* **from** *section* **where** *semester* = 'Spring' **and** *year*= 2010);

The **in** and **not in** operators can also be used on enumerated sets. The following query selects the names of instructors whose names are neither “Mozart” nor “Einstein”.

select distinct *name*
from *instructor*
where *name* **not in** (’Mozart’, ’Einstein’);

- “find the total number of (distinct) students who have taken course sections taught by the instructor with *ID* 110011” as follows:
  select count (**distinct** *ID*) **from** *takes*
  where (*course id*, *sec id*, *semester*, *year*) **in** (**select** *course id*, *sec id*, *semester*, *year*
  from *teaches* **where** *teaches.ID*= 10101);

Set Comparison

As an example of the ability of a nested subquery to compare sets, consider the query
“Find the names of all instructors whose salary is greater than at least one instructor in the Biology department.”

select distinct *T.name*
from *instructor* **as** *T*, *instructor* **as** *S*
where *T.salary* > *S.salary* **and** *S.dept name* = ’Biology’;

SQL does, however, offer an alternative style for writing the preceding query. The phrase “greater than at least one” is represented in SQL by > **some**. This construct allows us to rewrite the query in a form that resembles closely our formulation of the query in English.

select *name* **from** *instructor* **where** *salary* > **some** (**select** *salary* **from** *instructor* **where** *dept name* = ’Biology’);

The subquery:

(**select** *salary* **from** *instructor* **where** *dept name* = ’Biology’)

generates the set of all salary values of all instructors in the Biology department. The > **some** comparison in the **where** clause of the outer **select** is true if the *salary* value of the tuple is greater than at least one member of the set of all salary values for instructors in Biology.

SQL also allows < **some**, <= **some**, >= **some**, = **some**, and <> **some** comparisons. As an exercise, verify that = **some** is identical to **in**, whereas <> **some** is *not* the same as **not in**.

The construct > **all** corresponds to the phrase "greater than all." Using this construct, we write the query as follows:

select *name* **from** *instructor* **where** *salary* > **all** (**select** *salary* **from** *instructor* **where** *dept name* = 'Biology');

As it does for **some**, SQL also allows < **all**, <= **all**, >= **all**, = **all**, and <> **all** comparisons. As an exercise, verify that <> **all** is identical to **not in**, whereas =**all** is *not* the same as **in**.

- "Find the departments that have the highest average salary."

those departments for which the average salary is greater than or equal to all average salaries:

select *dept name*
from *instructor*
group by *dept name*
having avg (*salary*) >= **all** (**select avg** (*salary*)
from *instructor*
group by *dept name*);

Exists and Not Exists: Test for Empty Relations

Exists: SQL includes a feature for testing whether a subquery has any tuples in its result. The **exists** construct returns the value **true** if the argument subquery is nonempty

- "Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester"
  select *course id* **from** *section* **as** *S* **where** *semester* = 'Fall' **and** *year*= 2009 **and**
  exists (**select** * **from** *section* **as** *T* **where** *semester* = 'Spring' **and** *year*= 2010 **and**
  S.course id= T.course id);

Correlated subquery : The above query also illustrates a feature of SQL where a correlation name from an outer query (*S* in the above query), can be used in a subquery in the **where** clause. A subquery that uses a correlation name from an outer query is called a **correlated subquery**.

Not exists: We can use the **not exists** construct to simulate the set containment (that is, superset) operation: We can write "relation *A* contains relation *B*" as "**not exists** (*B* **except** *A*)."

- "Find all students who have taken all courses offered in the Biology department."

select distinct *S.ID, S.name* **from** *student* **as** *S* **where not exists** ((**select** *course id* **from** *course* **where** *dept name* = 'Biology') **except** (**select** *T.course id* **from** *takes* **as** *T* **where** *S.ID = T.ID*));

UNIQUE : Test for the Absence of Duplicate Tuples

The **unique** construct returns the value **true** if the argument subquery contains no duplicate tuples

"Find all courses that were offered at most once in 2009"

select *T.course id*
from *course* **as** *T*
where unique (**select** *R.course id* **from** *section* **as** *R* **where** *T.course id= R.course id* **and** *R.year* = 2009);

We can test for the existence of duplicate tuples in a subquery by using the **not unique** construct. To illustrate this construct, consider the query "Find all courses that were offered at least twice in 2009" as follows:

select *T.course id*
from *course* **as** *T*
where not unique (**select** *R.course id*
from *section* **as** *R*
where *T.course id= R.course id* **and**
R.year = 2009);

Subqueries in the From Clause

SQL allows a subquery expression to be used in the **from** clause. The key concept applied here is that any **select-from-where** expression returns a relation as a result and, therefore, can be inserted into another **select-from-where** anywhere that a relation can appear

"Find the average instructors' salaries of those departments where the average salary is greater than $42,000."

select *dept name, avg salary*
from (**select** *dept name*, **avg** (*salary*) **as** *avg salary*
from *instructor*

group by *dept name*)
where *avg salary* > 42000;

We can give the subquery result relation a name, and rename the attributes, using the **as** clause

select *dept name, avg salary*
from (**select** *dept name*, **avg** (*salary*)
from *instructor*
group by *dept name*)
as *dept avg* (*dept name, avg salary*)
where *avg salary* > 42000;

suppose we wish to find the maximum total salary across all departments of the total salary at each department.

select max (*tot salary*)
from (**select** *dept name*, **sum**(*salary*)
from *instructor*
group by *dept name*) **as** *dept total* (*dept name, tot salary*);

We note that nested subqueries in the **from** clause cannot use correlation variables allows a subquery in the **from** clause that is prefixed by the **lateral** keyword
For example, if we wish to print the names of each instructor, along with their salary and the average salary in their department, we could write the query as follows:
select *name, salary, avg salary*
from *instructor I1*, **lateral** (**select avg**(*salary*) as *avg salary*
from *instructor I2*
where *I2.dept name= I1.dept name*);

The with Clause

The **with** clause provides away of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs Consider the following query, which finds those departments with the maximum budget.

with *max budget* (*value*) **as**
(**select max**(*budget*)

from *department*)
select *budget*
from *department, max budget*
where *department.budget = max budget.value*;

For example, suppose we want to find all departments where the total salary is greater than the average of the total salary at all departments.We can write the query using the **with** clause as follows.

with *dept total* (*dept name*, *value*) **as**
(**select** *dept name*, **sum**(*salary*)
from *instructor*
group by *dept name*),
dept total avg(*value*) **as**
(**select avg**(*value*)
from *dept total*)
select *dept name*
from *dept total, dept total avg*
where *dept total.value >= dept total avg.value*;

Scalar Subqueries

SQL allows subqueries to occur wherever an expression returning a value is permitted, provided the subquery returns only one tuple containing a single attribute; such subqueries are called **scalar subqueries**.

in the following example that lists all departments along with the number of instructors in each department:

select *dept name*,
(**select count**(*)
from *instructor*
where *department.dept name = instructor.dept name*)
as *num instructors*
from *department*;

The subquery in the above example is guaranteed to return only a single value since it has a **count**(*) aggregate without a **group by**. Scalar subqueries can occur in **select**, **where**, and **having** clauses. Scalar subqueries may also be defined without aggregates.

Modification of the Database

We have restricted our attention until now to the extraction of information from the database.Now, we show how to add, remove, or change information with SQL.

Deletion

Adelete request is expressed inmuch the sameway as a query.We can delete only whole tuples; we cannot delete values on only particular attributes.

delete from *r* **where** *P*;

where *P* represents a predicate and *r* represents a relation.

The predicate in the **where** clause may be as complex as a **select** command's **where** clause.

delete from *instructor*; deletes all tuples from the *instructor* relation. The *instructor* relation itself still exists, but it is empty.

Here are examples of SQL delete requests:

• Delete all tuples in the *instructor* relation pertaining to instructors in the Finance department.

delete from *instructor* **where** *dept name*= 'Finance';

• Delete all instructors with a salary between $13,000 and $15,000.

delete from *instructor*
where *salary* **between** 13000 **and** 15000;

• Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building.

delete from *instructor*
where *dept name* **in** (**select** *dept name*
from *department*
where *building* = 'Watson');

This **delete** request first finds all departments located in Watson, and then deletes all *instructor* tuples pertaining to those departments. Note that, although we may delete tuples from only one relation at a time, we may reference any number of relations in a **select-from-where** nested in the **where** clause of a **delete**. The **delete** request can contain a nested **select** that references the relation fromwhich tuples are to be deleted.

For example, suppose that we want to delete the records of all instructors with salary below the average at the university.We could write:

delete from *instructor*
where *salary*< (**select avg** (*salary*)
from *instructor*);

The **delete** statement first tests each tuple in the relation *instructor* to check whether the salary is less than the average salary of instructors in the university.

Insertion

To insert data into a relation, we either specify a tuple to be inserted or write a querywhose result is a set of tuples to be inserted. The simplest **insert** statement is a request to insert one tuple.

insert into *course* **values** ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

Suppose that we want to make each student in the Music department who has earned more than 144 credit hours, an instructor in the Music department, with a salary of $18,000. We write:

insert into *instructor* **select** *ID*, *name*, *dept name*, 18000 **from** *student* **where** *dept name* = 'Music' **and** *tot cred* > 144;

Instead of specifying a tuple as we did earlier in this section, we use a **select** to specify a set of tuples. SQL evaluates the **select** statement first, giving a set of tuples that is then inserted into the *instructor* relation. Each tuple has an *ID*, a *name*, a *dept name* (Music), and an salary of $18,000.

Updates

As we could for **insert** and **delete**, we can choose the tuples to be updated by using a query. Suppose that annual salary increases are being made, and salaries of all instructors are to be increased by 5 percent.We write:

update *instructor*
set *salary= salary* * 1.05;

If a salary increase is to be paid only to instructors with salary of less than $70,000, we can write:

update *instructor*

set *salary* = *salary* * 1.05
where *salary* < 70000;

In general, the **where** clause of the **update** statement may contain any construct legal in the **where** clause of the **select** statement.

"Give a 5 percent salary raise to instructors whose salary is less than average" as follows:

update *instructor*
set *salary* = *salary* * 1.05
where *salary* < (**select avg** (*salary*)
from *instructor*);

Consider an update where we set the *tot cred* attribute of each *student* tuple to the sum of the credits of courses successfully completed by the student. We assume that a course is successfully completed if the student has a grade that is not 'F' or null.

update *student S* **set** *tot cred* = (**select sum**(*credits*) **from** *takes* **natural join** *course*
where *S.ID= takes.ID* **and** *takes.grade* <> 'F' **and** *takes.grade* **is not null**);

Join Expressions

SQL provides other forms of the join operation, including the ability to specify an explicit **join predicate.**

The **on** condition allows a general predicate over the relations being joined. This predicate is written like a **where** clause predicate except for the use of the keyword **on** rather than **where**. Consider the following query, which has a join expression containing the **on** condition.

select * **from** *student* **join** *takes* **on** *student.ID= takes.ID*;

The **on** condition above specifies that a tuple from *student* matches a tuple from *takes* if their *ID* values are equal. The join expression in this case is almost the same as the join expression *student* **natural join** *takes* The **outer join** operation works in a manner similar to the join operations we have already studied, but preserve those tuples that would be lost in a join, by creating tuples in the result containing null values.

There are in fact three forms of outer join:

• The **left outer join** preserves tuples only in the relation named before (to the left of) the **left outer join** operation.

• The **right outer join** preserves tuples only in the relation named after (to the right of) the **right outer join** operation.

• The **full outer join** preserves tuples in both relations.

In contrast, the join operations we studied earlier that do not preserve non matched tuples are called **inner join** operations, to distinguish them from the outer-join operations.

Then, for every tuple *t* in the left-hand-side relation that does not match any tuple in the right-hand-side relation in the inner join, add a tuple *r* to the result of the join constructed as follows:

• The attributes of tuple *r* that are derived from the left-hand-side relation are filled in with the values from tuple *t*.

• The remaining attributes of *r* are filled with null values.

select * **from** *student* **natural left outer join** *takes*;

As another example of the use of the outer-join operation, we can write the query

- "Find all students who have not taken a course" as:

  select *ID* **from** *student* **natural left outer join** *takes*
  where *course id* **is** *null*;

Tuples from the righthand-side relation that do not match any tuple in the left-hand-side relation are padded with nulls and are added to the result of the right outer join.

select * **from** *takes* **natural right outer join** *student*;

- "Display a list of all students in the Comp. Sci. department, along with the course sections, if any, that they have taken in Spring 2009;
  All course sections from Spring 2009 must be displayed, even if no student from the Comp. Sci. department has taken the course section."

  select * **from** (**select** * **from** *student* **where** *dept name*= 'Comp. Sci')
  natural full outer join
  (**select** * **from** *takes* **where** *semester* = 'Spring' **and** *year* = 2009);

The **on** clause can be used with outer joins

select *

from *student* **left outer join** *takes* **on** *student.ID= takes.ID*;

Views

Instead, SQL allows a "virtual relation" to be defined by a query, and the relation conceptually contains the result of the query. The virtual relation is not precomputed and stored, but instead is computed by executing the query whenever the virtual relation is used.

Any such relation that is not part of the logical model, but is made visible to a user as a virtual relation, is called a **view**

View Definition

We define a view in SQL by using the **create view** command

The form of the **create view** command is:

create view *v* **as** <query expression>;

where <query expression> is any legal query expression. The view name is represented by v.

create view faculty as

select ID, name, dept name

from instructor;

As explained earlier, the view relation conceptually contains the tuples in the query result, but is not precomputed and stored. Instead, the database system stores the query expression associated with the view relation.

To create a view that lists all course sections offered by the Physics department in the Fall 2009 semester with the building and room number of each section, we write:

create view physics fall 2009 as select course.course id, sec id, building, room number from course, section where course.courseid = section.courseid and course.dept name = 'Physics' and section.semester = 'Fall' and section.year = '2009';

Using Views in SQL Queries

Using the view **physics fall 2009,** we can find all Physics courses offered in the Fall 2009 semester in the Watson building by writing:

- select course id from physics fall 2009 where building= 'Watson';

View names may appear in a query any place where a relation name may appear, The attribute names of a view can be specified explicitly as follows:

create view departments total salary(dept name, total salary) as select dept name, sum (salary) from instructor group by dept name;

One view may be used in the expression defining another view. For example, we can define a view physics fall 2009 watson that lists the course ID and room number of all Physics courses offered in the Fall 2009 semester in the Watson building as follows:

create view physics fall 2009 watson as select course id, room number from physics fall 2009 where building= 'Watson';

Materialized Views

Certain database systems allow view relations to be stored, but they make sure that, if the actual relations used in the view definition change, the view is kept up-to-date. Such views are called **materialized views**.

For example, consider the view *departments total salary*. If the above view is materialized, its results would be stored in the database. However, if an *instructor* tuple is added to or deleted from the *instructor* relation, the result of the query defining the view would change, and as a result the materialized view's contents must be updated. The process of keeping the materialized view up-to-date is called **materialized view maintenance**

Update of a View

The difficulty is that a modification to the database expressed in terms of a view must be translated to a modification to the actual relations in the logical model of the database. Another problem with modification of the database through views occurs with a view such as:

create view *instructor info* **as**
select *ID, name, building*
from *instructor, department*
where *instructor.dept name= department.dept name*;

This view lists the *ID*, *name*, and building-name of each instructor in the university.
Consider the following insertion through this view:

insert into *instructor info* **values** ('69987', 'White', 'Taylor');

Suppose there is no instructor with ID 69987, and no department in the Taylor building. Then the only possible method of inserting tuples into the *instructor* and *department* relations is to

- Insert ('69987', 'White', *null*, *null*) into *instructor* and (*null*, 'Taylor', *null*) into *department*.

In general, an SQL view is said to be **updatable** (that is, inserts, updates or deletes can be applied on the view) if the following conditions are all satisfied by the query defining the view:
• The **from** clause has only one database relation.
• The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
• Any attribute not listed in the **select** clause can be set to *null*; that is, it does not have a **not null** constraint and is not part of a primary key.
• The query does not have a **group by** or **having** clause.

Transactions

A **transaction** consists of a sequence of query and/or update statements. The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed.
Commit work commits the current transaction; that is, it makes the updates performed by the transaction become permanent in the database. After the transaction is committed, a new transaction is automatically started.
Rollback work causes the current transaction to be rolled back; that is, it undoes all the updates performed by the SQL statements in the transaction. The keyword **work** is optional in both the statements
Once a transaction has executed **commit work**, its effects can no longer be undone by **rollback work**.

The database system guarantees that in the event of some failure, such as an error in one of the SQL statements, a power outage, or a system crash, a transaction's effects will be rolled back if it has not yet executed **commit work**. the database provides an abstraction of a transaction as being **atomic**, that is, indivisible. Either all the effects of the transaction are reflected in the database, or none are (after rollback).

A better alternative, which is part of the SQL:1999 standard (but supported by only some SQL implementations currently), is to allow multiple SQL statements to be enclosed between the keywords **begin atomic** . . . **end**. All the statements between the keywords then form a single transaction.

Integrity Constraints

Integrity constraints ensure that changes made to the database by authorized users do not result in a loss of data consistency.

Examples of integrity constraints are:

• An instructor name cannot be *null*.

• No two instructors can have the same instructor ID.

• Every department name in the *course* relation must have a matching department name in the *department* relation.

• The budget of a department must be greater than $0.00.

Integrity constraints are usually identified as part of the database schema design process, and declared as part of the **create table** command used to create relations. However, integrity constraints can also be added to an existing relation by using the command **alter table** *table-name* **add** *constraint*, where *constraint* can be any constraint on the relation.

Constraints on a Single Relation

The **create table** command may also include integrity-constraint statements. In addition to the primary-key constraint, there are a number of other ones that can be included in the **create table** command.

The allowed integrity constraints include

• **NOT NULL**

• **UNIQUE**

• **CHECK**(<PREDICATE>)

Not Null Constraint:

we wish to forbid null values, and we can do so by restricting the domain of the attributes *name* and *budget* to exclude null values, by declaring it as follows:

name **varchar**(20) **not null**

budget **numeric**(12,2) **not null**

The **not null** specification prohibits the insertion of a null value for the attribute.

Unique Constraint:

SQL also supports an integrity constraint:

unique (*Aj*1 , *Aj*2, . . . , *Ajm*)

The **unique** specification says that attributes *Aj*1 , *Aj*2, . . . , *Ajm* form a candidate key; that is, no two tuples in the relation can be equal on all the listed attributes.

The check Clause:

A common use of the **check** clause is to ensure that attribute values satisfy specified conditions, in effect creating a powerful type system. For instance, a clause **check** (*budget* > 0) in the **create table** command for relation *department* would ensure that the value of **budget** is nonnegative.

create table *section* (*course id* **varchar** (8),*sec id* **varchar** (8),*semester* **varchar** (6),
year **numeric** (4,0), *building* **varchar** (15), *room number* **varchar** (7),
timeslotid **varchar** 4), **primary key** (*course id*, *sec id*, *semester*, *year*),
check (*semester* **in** ('Fall', 'Winter', 'Spring', 'Summer')));

Referential Integrity:

Often, we wish to ensure that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation. This condition is called **referential integrity**.

The definition of the *course* table has a declaration "**foreign key** (*dept name*) **references** *department*". This foreign-key declaration specifies that for each course tuple, the department name specified in the tuple must exist in the *department* relation.

By default, in SQL a foreign key references the primary-key attributes of the referenced table. SQL also supports a version of the **references** clause where a list of attributes of the referenced relation can be specified explicitly. The specified list of attributes must, however, be declared

as a candidate key of the referenced relation, using either a **primary key** constraint, or a **unique** constraint.

We can use the following short form as part of an attribute definition to declare that the attribute forms a foreign key:

dept name **varchar**(20) **references** *department*

create table *course*
(. . .
foreign key (*dept name*) **references** *department*
on delete cascade
on update cascade,
. . .);

Because of the clause **on delete cascade** associated with the foreign-key declaration, if a delete of a tuple in *department* results in this referential-integrity constraint being violated, the system does not reject the delete. Instead, the delete "cascades" to the *course* relation, deleting the tuple that refers to the department that was deleted.

Integrity Constraint Violation During a Transaction

The SQL standard allows a clause **initially deferred** to be added to a constraint specification; the constraint would then be checked at the end of a transaction, and not at intermediate steps. A constraint can alternatively be specified as **deferrable**, which means it is checked immediately by default, but can be deferred when desired.

For constraints declared as deferrable, executing a statement **set constraints** *constraint-list* **deferred** as part of a transaction causes the checking of the specified constraints to be deferred to the end of that transaction.

Complex Check Conditions and Assertions

The predicate in the **check** clause can be an arbitrary predicate, which can include a subquery. If a database implementation supports subqueries in the **check** clause, we could specify the following referential-integrity constraint on the relation *section*:

check (*time slot id* **in** (**select** *time slot id* **from** *time slot*))

The **check** condition verifies that the *time slot id* in each tuple in the *section* relation is actually the identifier of a time slot in the *time slot* relation. Thus, the condition has to be checked not only when a tuple is inserted or modified in *section*, but also when the relation *time slot* changes Complex **check** conditions can be useful when we want to ensure integrity of data, but may be costly to test. For example, the predicate in the **check** clause

Create assertion: An **assertion** is a predicate expressing a condition that we wish the database always to satisfy. Domain constraints and referential-integrity constraints are special forms of assertions. We have paid substantial attention to these forms of assertions because they are easily tested and apply to a wide range of database applications.

Two examples of such constraints are:

• For each tuple in the *student* relation, the value of the attribute *tot cred* must equal the sum of credits of courses that the student has completed successfully.

• An instructor cannot teach in two different classrooms in a semester in the same time slot.1

An assertion in SQL takes the form:

create assertion <assertion-name> **check** <predicate>;

SQL Data Types and Schemas

Date and Time Types in SQL

date: A calendar date containing a (four-digit) year, month, and day of the month.

time: The time of day, in hours, minutes, and seconds. A variant, **time**(*p*), can be used to specify the number of fractional digits for seconds (the default being 0). It is also possible to store time-zone information along with the time by specifying **time with timezone**.

timestamp: A combination of **date** and **time**. A variant, **timestamp**(*p*), can be used to specify the number of fractional digits for seconds (the default here being 6).

Time, Date and time values can be specified like this:

date ’2001-04-25’

time ’09:30:00’

timestamp ’2001-04-25 10:29:01.45’

Default Values

SQL allows a default value to be specified for an attribute as illustrated by the following **create table** statement:

create table *student*
(*ID* **varchar** (5),
name **varchar** (20) **not null**,
dept name **varchar** (20),
tot cred **numeric** (3,0) **default** 0,
primary key (*ID*));

The default value of the *tot cred* attribute is declared to be 0. As a result, when a tuple is inserted into the *student* relation, if no value is provided for the *tot cred* attribute, its value is set to 0.

Schemas, Catalogs, and Environments:

Contemporary database systems provide a three-level hierarchy for naming relations. The top level of the hierarchy consists of **catalogs**, each of which can contain **schemas**. SQL objects such as relations and views are contained within a **schema**.

To identify a relation uniquely, a three-part name may be used, for example,

catalog5.univ schema.course

Thus if *catalog5* is the default catalog, we can use *univ schema.course* to identify the same relation uniquely.

The default catalog and schema are part of an **SQL environment** that is set up for each connection. The environment additionally contains the user identifier.

We can create and drop schemas by means of **create schema** and **drop schema** statements.

Authorization

We may assign a user several forms of authorizations on parts of the database. Authorizations on data include:

• Authorization to read data.
• Authorization to insert new data.
• Authorization to update data.

• Authorization to delete data.

Each of these types of authorizations is called a **privilege**. We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.

When a user submits a query or an update, the SQL implementation first checks if the query or update is authorized, based on the authorizations that the user has been granted. In addition to authorizations on data, users may also be granted authorizations on the database schema, allowing them, for example, to create, modify, or drop relations. A user who has some form of authorization may be allowed to pass on (grant) this authorization to other users, or to withdraw (revoke) an authorization that was granted earlier.

Granting and Revoking of Privileges

The SQL data-definition language includes commands to grant and revoke privileges. The **grant** statement is used to confer authorization. The basic form of this statement is:

grant <privilege list>
on <relation name or view name>
to <user/role list>;

The **select** authorization on a relation is required to read tuples in the relation. The following **grant** statement grants database users Amit and Satoshi **select** authorization on the *department* relation:

grant select on *department* **to** *Amit*, *Satoshi*;

The **update** authorization on a relation allows a user to update any tuple in the relation. The **update** authorization may be given either on all attributes of the relation or on only some. If **update** authorization is included in a **grant** statement, the list of attributes on which update authorization is to be granted optionally appears in parentheses immediately after the **update** keyword.

This **grant** statement gives users Amit and Satoshi update authorization on the *budget* attribute of the *department* relation:

grant update (*budget*) **on** *department* **to** *Amit*, *Satoshi*;

The **insert** authorization on a relation allows a user to insert tuples into the relation

The **delete** authorization on a relation allows a user to delete tuples from a relation.

The user name **public** refers to all current and future users of the system.

Thus, privileges granted to **public** are implicitly granted to all current and future

users.

To revoke an authorization, we use the **revoke** statement. It takes a form almost identical to that of **grant**:

revoke <privilege list>
on <relation name or view name>
from <user/role list>;

Thus, to revoke the privileges that we granted previously, we write

revoke select on *department* **from** *Amit*, *Satoshi*;
revoke update (budget) **on** *department* **from** *Amit*, *Satoshi*;

Roles

The notion of **roles** captures this concept. A set of roles is created in the database. Authorizations can be granted to roles, in exactly the same fashion as they are granted to individual users. Each database user is granted a set of roles (which may be empty) that she is authorized to perform.

In our university database, examples of roles could include *instructor*, *teaching assistant*, *student*, *dean*, and *department chair*.

Roles can be created in SQL as follows:

create role *instructor*;

Roles can then be granted privileges just as the users can, as illustrated in this statement:

grant select on *takes* **to** *instructor*;

Roles can be granted to users, as well as to other roles, as these statements show:

grant *dean* **to** Amit;
create role *dean*;
grant *instructor* **to** *dean*;
grant *dean* **to** Satoshi;

Thus the privileges of a user or a role consist of:

• All privileges directly granted to the user/role.

• All privileges granted to roles that have been granted to the user/role.

Authorization on Views

This staff member is not authorized to see information regarding faculty in other departments. Thus, the staffmember must be denied direct access to the *instructor* relation

This view can be defined in SQL as follows:

create view *geo instructor* **as**
(**select** * **from** *instructor*
where *dept name* = 'Geology');

Suppose that the staff member issues the following SQL query:

select * **from** *geo instructor*;

A user who creates a view does not necessarily receive all privileges on that view. She receives only those privileges that provide no additional authorization beyond those that she already had. For example, a user who creates a view cannot be given **update** authorization on a view without having **update** authorization on the relations used to define the view.

The **execute** privilege can be granted on a function or procedure, enabling a user to execute the function/procedure.

Authorizations on Schema

Only the owner of the schema can carry out any modification to the schema, such as creating or deleting relations, adding or dropping attributes of relations, and adding or dropping indices. However, SQL includes a **references** privilege that permits a user to declare foreign keys when creating relations. The SQL **references** privilege is granted on specific attributes in a manner like that for the **update** privilege.

The following **grant** statement allows user Mariano to create relations that reference the key *branch name* of the *branch* relation as a foreign key:

grant references (*dept name*) **on** *department* **to** Mariano;

Transfer of Privileges

If we wish to grant a privilege and to allow the recipient to pass the privilege on to other users, we append the **with grant option** clause to the appropriate **grant** command. For example, ifwewish to allowAmit the **select** privilege on *department* and allow Amit to grant this privilege to others, we write:

grant select on *department* **to** Amit **with grant option**;

Consider, as an example, the granting of update authorization on the *teaches* relation of the university database. Assume that, initially, the database administrator grants update authorization on *teaches* to users *U*1, *U*2, and *U*3, who may in turn pass on this authorization to other users.

Revoking of Privileges

As we just saw, revocation of a privilege from a user/role may cause other users/roles also to lose that privilege. This behavior is called *cascading revocation*.

In most database systems, cascading is the default behavior. However, the **revoke** statement may specify **restrict** in order to prevent cascading revocation:

revoke select on *department* **from** Amit, Satoshi **restrict**;

The keyword **cascade** can be used instead of **restrict** to indicate that revocation should cascade; however, it can be omitted, as we have done in the preceding examples, since it is the default behavior. The following **revoke** statement revokes only the grant option, rather than the actual **select** privilege:

revoke grant option for select on *department* **from** Amit;

To grant a privilege with the grantor set to the current role associated with a session, we can add the clause:

granted by current role

Database Programming: Techniques and Issues

Most database systems have an interactive interface where these SQL commands can be typed directly into a monitor for execution by the database system. For example, in a computer system where the Oracle RDBMS is installed, the command SQLPLUS starts the interactive interface. In practice, the majority of database interactions are executed through programs that have been carefully designed and tested.

These programs are generally known as **application programs** or **database applications**, and are used as *canned transactions* by the end users Another common use of database programming is to access a database through an application program that implements a **Web interface**, for example, when making airline reservations or online purchases.

Approaches to Database Programming

The main approaches for database programming are the following

1. Embedding database commands in a general-purpose programming language.

In this approach, database statements are **embedded** into the host programming language, but they are identified by a special prefix. For example, the prefix for embedded SQL is the string EXEC SQL, which precedes all SQL commands in a host language program. A **precompiler**

or **preproccessor** scans the source program code to identify database statements and extract them for processing by the DBMS.

2. Using a library of database functions. A library of functions is made available to the host programming language for database calls. For example, there could be functions to connect to a database, execute a query, execute an update, and so on.

3. Designing a brand-new language. A database programming language is designed from scratch to be compatible with the database model and query language. Additional programming structures such as loops and conditional statements are added to the database language to convert it into a full-fledged programming language. An example of this approach is Oracle's PL/SQL.

Impedance Mismatch

Impedance mismatch is the term used to refer to the problems that occur because of differences between the database model and the programming language model.

For example, the practical relational model has three main constructs: columns (attributes) and their data types, rows (also referred to as tuples or records), and tables (sets or multisets of records). The first problem that may occur is that the *data types of the programming language* differ from the *attribute data types* that are available in the data model. Hence, it is necessary to have a **binding** for each host programming language that specifies for each attribute type the compatible programming language types.

For example, the data types available in C/C++ and Java are different, and both differ from the SQL data types, a binding is needed to map the *query result data structure*, which is a table, to an appropriate data structure in the programming language. A mechanism is needed to loop over the tuples in a **query result** in order to access a single tuple at a time and to extract individual values from the tuple. A **cursor** or **iterator variable** is typically used to loop over the tuples in a query result.

Impedance mismatch is less of a problem when a special database programming language is designed that uses the same data model and data types as the database model. One example of such a language is Oracle's PL/SQL.

Typical Sequence of Interaction in Database Programming

A common architecture for database access is the client/server model, where a **client program** handles the logic of a software application, but includes some calls to one or more **database**

servers to access or update the data. When writing such a program, a common sequence of interaction is the following:

1. When the client program requires access to a particular database, the program must first *establish* or *open* a **connection** to the database server. Typically, this involves specifying the Internet address (URL) of the machine where the database server is located, plus providing a login account name and password for database access.

2. Once the connection is established, the program can interact with the database by submitting queries, updates, and other database commands. In general, most types of SQL statements can be included in an application program.

3. When the program no longer needs access to a particular database, it should *terminate* or *close* the connection to the database.

A program can access multiple databases if needed. In some database programming approaches, only one connection can be active at a time, whereas in other approaches multiple connections can be established simultaneously.

Embedded SQL

In this section, we give an overview of the technique for how SQL statements can be embedded in a general-purpose programming language. We focus on two languages: C

Retrieving Single Tuples with Embedded SQL

When using C as the host language, an embedded SQL statement is distinguished from programming language statements by prefixing it with the keywords EXEC SQL so that a **preprocessor** (or **precompiler**) can separate embedded

SQL statements from the host language code. The SQL statements within a program are terminated by a matching END-EXEC or by a semicolon (;).

These are called **shared variables** because they are used in both the C program and the embedded SQL statements. Shared variables are prefixed by a colon (:) *when they appear in an SQL statement*. This distinguishes program variable names from the names of database schema constructs such as attributes (column names) and relations (table names).

We need to declare program variables to match the types of the database attributes that the program will process A few of the common bindings of C types to SQL types are as follows. The SQL types INTEGER, SMALLINT, REAL, and DOUBLE are mapped to the C types

long, short, float, and double, respectively. Fixed-length and varying-length strings (CHAR[*i*], VARCHAR[*i*]) in SQL can be mapped to arrays of characters (char [i+1], varchar [i+1]) in C. Notice that the only embedded SQL commands in Figure 13.1 are lines 1 and 7, which tell the precompiler to take note of the C variable names between BEGIN DECLARE and END DECLARE because they can be included in embedded SQL statements— as long as they are preceded by a colon (:). Lines 2 through 5 are regular C program declarations. The C program variables declared in lines 2 through 5 correspond to the attributes of the EMPLOYEE and DEPARTMENT tables from the

COMPANY database in Figure 3.5 that was declared by the SQL DDL in Figure 4.1. The variables declared in line 6—SQLCODE and SQLSTATE—are used to communicate errors and exception conditions between the database system and the executing program. Line 0 shows a program variable loop that will not be used in any embedded SQL statement, so it is declared outside the SQL declare section.

```
0)  int loop ;
1)  EXEC SQL BEGIN DECLARE SECTION ;
2)  varchar dname [16], fname [16], lname [16], address [31] ;
3)  char ssn [10], bdate [11], sex [2], minit [2] ;
4)  float salary, raise ;
5)  int dno, dnumber ;
6)  int SQLCODE ; char SQLSTATE [6] ;
7)  EXEC SQL END DECLARE SECTION ;
```

Figure 13.1
C program variables used in the embedded SQL examples E1 and E2.

Example of Embedded SQL Programming.

To illustrate embedded SQL programming is a repeating program segment (loop) that takes as input a Social Security number of an employee and prints some information from the corresponding EMPLOYEE record in the database. The C program code is shown as program segment E1 in Figure 13.2. The program reads (inputs) an Ssn value and then retrieves the EMPLOYEE tuple with that Ssn from the database via the embedded SQL command. The **INTO** clause (line 5) specifies the program variables into which attribute values from the database record are retrieved. C program variables in the INTO clause are prefixed with a colon (:), as we discussed earlier. The INTO clause can be used in this way only when the query result is a single record; if multiple records are retrieved, an error will be generated.

If the value returned by the DBMS in SQLCODE is 0, the previous statement was executed without errors or exception conditions.

```
    //Program Segment E1:
 0) loop = 1 ;
 1) while (loop) {
 2)   prompt("Enter a Social Security Number: ", ssn) ;
 3)   EXEC SQL
 4)     select Fname, Minit, Lname, Address, Salary
 5)     into :fname, :minit, :lname, :address, :salary
 6)     from EMPLOYEE where Ssn = :ssn ;
 7)   if (SQLCODE == 0) printf(fname, minit, lname, address, salary)
 8)     else printf("Social Security Number does not exist: ", ssn) ;
 9)   prompt("More Social Security Numbers (enter 1 for Yes, 0 for No): ", loop) ;
10)   }
```

Figure 13.2
Program segment E1, a C program segment with embedded SQL.

Retrieving Multiple Tuples with Embedded SQL Using Cursors

The cursor is declared when the SQL query command is declared in the program. Later in the program, an **OPEN CURSOR** command fetches the query result from the database and sets the cursor to a position *before the first row* in the result of the query. This becomes the **current row** for the cursor. Subsequently, **FETCH** commands are issued in the program; each FETCH moves the cursor to the *next row* in the result of the query, making it the current row and copying its attribute values into the C (host language) program variables specified in the FETCH command by an INTO clause.

Figure 13.3
Program segment E2, a C program segment that uses cursors with embedded SQL for update purposes.

```
    //Program Segment E2:
 0) prompt("Enter the Department Name: ", dname) ;
 1) EXEC SQL
 2)   select Dnumber into :dnumber
 3)   from DEPARTMENT where Dname = :dname ;
 4) EXEC SQL DECLARE EMP CURSOR FOR
 5)   select Ssn, Fname, Minit, Lname, Salary
 6)   from EMPLOYEE where Dno = :dnumber
 7)   FOR UPDATE OF Salary ;
 8) EXEC SQL OPEN EMP ;
 9) EXEC SQL FETCH from EMP into :ssn, :fname, :minit, :lname, :salary ;
10) while (SQLCODE == 0) {
11)   printf("Employee name is:", Fname, Minit, Lname) ;
12)   prompt("Enter the raise amount: ", raise) ;
13)   EXEC SQL
14)     update EMPLOYEE
15)     set Salary = Salary + :raise
16)     where CURRENT OF EMP ;
17)   EXEC SQL FETCH from EMP into :ssn, :fname, :minit, :lname, :salary ;
18)   }
19) EXEC SQL CLOSE EMP ;
```

Module-4 : Database Design

4.1 Informal Design Guidelines for Relation Schemas

Before discussing the formal theory of relational database design, we discuss four *informal guidelines* that may be used as *measures to determine the quality* of relation schema design:

■ Making sure that the semantics of the attributes is clear in the schema

■ Reducing the redundant information in tuples

■ Reducing the NULL values in tuples

■ Disallowing the possibility of generating spurious tuples.

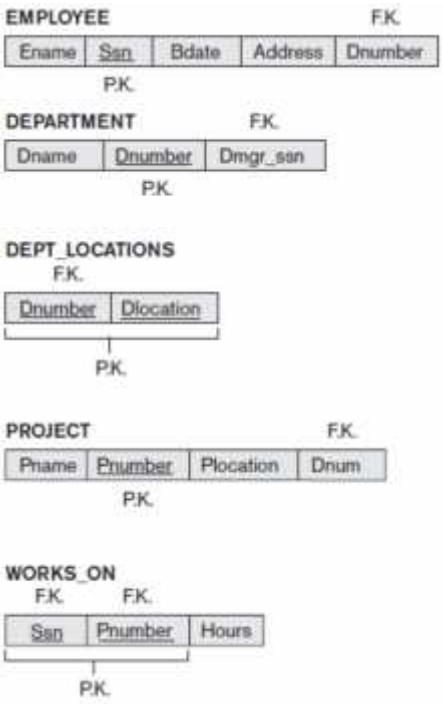
- **Imparting Clear Semantics to Attributes in Relations**

Whenever we group attributes to form a relation schema, we assume that attributes belonging to one relation have certain real-world meaning and a proper interpretation associated with them. The semantics of a relation refers to its meaning resulting from the interpretation of attribute values in a tuple.

Guideline 1

Design a relation schema so that it is easy to explain its meaning. Do not combine attributes from multiple entity types and relationship types into a single relation.

Figure 15.1
A simplified COMPANY relational database schema.



.

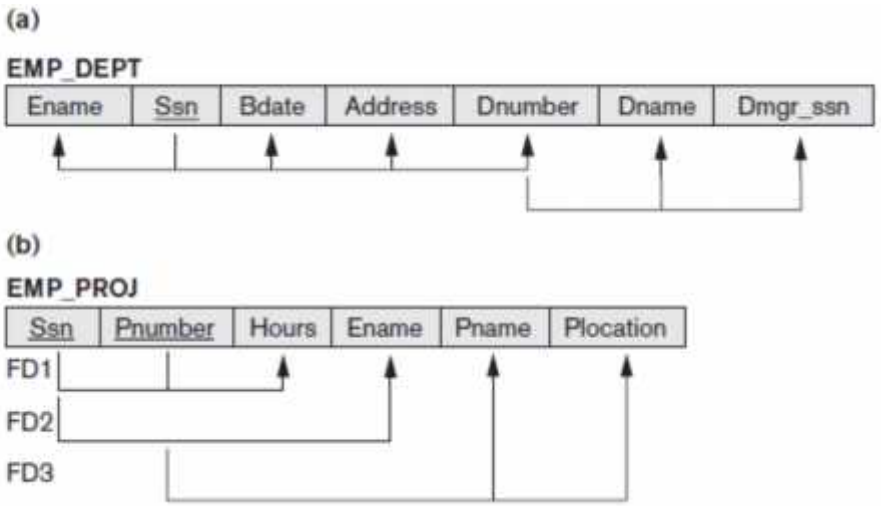


Figure 15.3
Two relation schemas suffering from update anomalies. (a) EMP_DEPT and (b) EMP_PROJ.

- **Redundant Information in Tuples and Update Anomalies**

One goal of schema design is to minimize the storage space used by the base relations (and hence the corresponding files). Grouping attributes into relation schemas has a significant effect on storage space. For example, compare the space used by the two base relations EMPLOYEE and DEPARTMENT in Figure 15.3 with that for an EMP_DEPT base relation in Figure 15.4, which is the result of applying the NATURAL JOIN operation to EMPLOYEE and DEPARTMENT.

In EMP_DEPT, the attribute values pertaining to a particular department (Dnumber, Dname, Dmgr_ssn) are repeated for *every employee who works for that department.*

Storing natural joins of base relations leads to an additional problem referred to as update anomalies. These can be classified into **insertion anomalies, deletion anomalies, and modification anomalies.**

Insertion Anomalies. Insertion anomalies can be differentiated into two types, illustrated by the following examples based on the EMP_DEPT relation:

■ To insert a new employee tuple into EMP_DEPT, we must include either the attribute values for the department that the employee works for, or NULLs (if the employee does not work for a department as yet). For example, to insert a new tuple for an employee who works in department number 5, we must enter all the attribute values of department 5 correctly so that they are *consistent* with the corresponding values for department 5 in other tuples in EMP_DEPT.

Deletion Anomalies. The problem of deletion anomalies is related to the second insertion anomaly situation just discussed. If we delete from EMP_DEPT an employee tuple that happens to represent the last employee working for a particular department, the information

concerning that department is lost from the database. This problem does not occur in the database of Figure 15.2 because DEPARTMENT tuples are stored separately.

Modification Anomalies. In EMP_DEPT, if we change the value of one of the attributes of a particular department—say, the manager of department 5—we must update the tuples of *all* employees who work in that department; otherwise, the database will become inconsistent. If we fail to update some tuples, the same department will be shown to have two different values for manager in different employee tuples, which would be wrong.

Guideline 2

Design the base relation schemas so that no insertion, deletion, or modification anomalies are present in the relations. If any anomalies are present, note them clearly and make sure that the programs that update the database will operate correctly.

Figure 15.2
Sample database state for the relational database schema in Figure 15.1.

EMPLOYEE

Ename	Ssn	Bdate	Address	Dnumber
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4
Wallace, Jennifer S.	987654321	1941-06-20	291Berry, Bellaire, TX	4
Narayan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1

DEPARTMENT

Dname	Dnumber	Dmgr_ssn
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Ssn	Pnumber	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	Null

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

NULL Values in Tuples

In some schema designs we may group many attributes together into a "fat" relation. If many of the attributes do not apply to all tuples in the relation, we end up with many NULLs in those tuples. This can waste space at the storage level and may also lead to problems with understanding the meaning of the attributes and with specifying JOIN operations at the logical level.5 Another problem with NULLs is how to account for them when aggregate operations such as COUNT or SUM are applied. SELECT and JOIN operations involve comparisons; if NULL values are present, the results may become unpredictable.6 Moreover, NULLs can have multiple interpretations, such as the following:

■ The attribute *does not apply* to this tuple. For example, Visa_status may not apply to U.S. students.

■ The attribute value for this tuple is *unknown*. For example, the Date_of_birth may be unknown for an employee.

■ The value is *known but absent*; that is, it has not been recorded yet. For example, the Home_Phone_Number for an employee may exist, but may not be available and recorded yet.

Guideline 3

As far as possible, avoid placing attributes in a base relation whose values may frequently be NULL. If NULLs are unavoidable, make sure that they apply in exceptional cases only and do not apply to a majority of tuples in the relation.

Generation of Spurious Tuples

Consider the two relation schemas EMP_LOCS and EMP_PROJ1 in Figure 15.5(a), which can be used instead of the single EMP_PROJ relation in Figure 15.3(b). A tuple in EMP_LOCS means that the employee whose name is Ename works on *some project* whose location is Plocation. A tuple in EMP_PROJ1 refers to the fact that the employee whose Social Security number is Ssn works Hours per week on the project whose name, number, and location are Pname, Pnumber, and Plocation.

Suppose that we used EMP_PROJ1 and EMP_LOCS as the base relations instead of EMP_PROJ. This produces a particularly bad schema design because we cannot recover the information that was originally in EMP_PROJ from EMP_PROJ1 and EMP_LOCS. If we attempt a NATURAL JOIN operation on EMP_PROJ1 and EMP_LOCS, the result produces many more tuples than the original set of tuples in EMP_PROJ. In Figure 15.6, the result of applying the join to only the tuples *above* the dashed lines in Figure 15.5(b) is shown (to reduce the size of the resulting relation). Additional tuples that were not in EMP_PROJ are called

spurious tuples because they represent spurious information that is not valid. The spurious tuples are marked by asterisks (*) in Figure 15.6.

Guideline 4

Design relation schemas so that they can be joined with equality conditions on attributes that are appropriately related (primary key, foreign key) pairs in a way that guarantees that no spurious tuples are generated.

(a)

EMP_LOCS

Ename	Plocation

P.K.

EMP_PROJ1

Ssn	Pnumber	Hours	Pname	Plocation

P.K.

Figure 15.5
Particularly poor design for the EMP_PROJ relation in Figure 15.3(b). (a) The two relation schemas EMP_LOCS and EMP_PROJ1. (b) The result of projecting the extension of EMP_PROJ from Figure 15.4 onto the relations EMP_LOCS and EMP_PROJ1.

(b)

EMP_LOCS

Ename	Plocation
Smith, John B.	Bellaire
Smith, John B.	Sugarland
Narayan, Ramesh K.	Houston
English, Joyce A.	Bellaire
English, Joyce A.	Sugarland
Wong, Franklin T.	Sugarland
Wong, Franklin T.	Houston
Wong, Franklin T.	Stafford
Zelaya, Alicia J.	Stafford
Jabbar, Ahmad V.	Stafford
Wallace, Jennifer S.	Stafford
Wallace, Jennifer S.	Houston
Borg, James E.	Houston

EMP_PROJ1

Ssn	Pnumber	Hours	Pname	Plocation
123456789	1	32.5	ProductX	Bellaire
123456789	2	7.5	ProductY	Sugarland
666884444	3	40.0	ProductZ	Houston
453453453	1	20.0	ProductX	Bellaire
453453453	2	20.0	ProductY	Sugarland
333445555	2	10.0	ProductY	Sugarland
333445555	3	10.0	ProductZ	Houston
333445555	10	10.0	Computerization	Stafford
333445555	20	10.0	Reorganization	Houston
999887777	30	30.0	Newbenefits	Stafford
999887777	10	10.0	Computerization	Stafford
987987987	10	35.0	Computerization	Stafford
987987987	30	5.0	Newbenefits	Stafford
987654321	30	20.0	Newbenefits	Stafford
987654321	20	15.0	Reorganization	Houston
888665555	20	NULL	Reorganization	Houston

	Ssn	Pnumber	Hours	Pname	Plocation	Ename
	123456789	1	32.5	ProductX	Bellaire	Smith, John B.
*	123456789	1	32.5	ProductX	Bellaire	English, Joyce A.
	123456789	2	7.5	ProductY	Sugarland	Smith, John B.
*	123456789	2	7.5	ProductY	Sugarland	English, Joyce A.
*	123456789	2	7.5	ProductY	Sugarland	Wong, Franklin T.
	666884444	3	40.0	ProductZ	Houston	Narayan, Ramesh K.
*	666884444	3	40.0	ProductZ	Houston	Wong, Franklin T.
*	453453453	1	20.0	ProductX	Bellaire	Smith, John B.
	453453453	1	20.0	ProductX	Bellaire	English, Joyce A.
*	453453453	2	20.0	ProductY	Sugarland	Smith, John B.
	453453453	2	20.0	ProductY	Sugarland	English, Joyce A.
*	453453453	2	20.0	ProductY	Sugarland	Wong, Franklin T.
*	333445555	2	10.0	ProductY	Sugarland	Smith, John B.
*	333445555	2	10.0	ProductY	Sugarland	English, Joyce A.
	333445555	2	10.0	ProductY	Sugarland	Wong, Franklin T.
*	333445555	3	10.0	ProductZ	Houston	Narayan, Ramesh K.
	333445555	3	10.0	ProductZ	Houston	Wong, Franklin T.
	333445555	10	10.0	Computerization	Stafford	Wong, Franklin T.
*	333445555	20	10.0	Reorganization	Houston	Narayan, Ramesh K.
	333445555	20	10.0	Reorganization	Houston	Wong, Franklin T.

Figure 15.6
Result of applying NATURAL JOIN to the tuples above the dashed lines in EMP_PROJ1 and EMP_LOCS of Figure 15.5. Generated spurious tuples are marked by asterisks.

4.2 Functional Dependencies

Definition of Functional Dependency

A functional dependency is a constraint between two sets of attributes from the database. Suppose that our relational database schema has n attributes A1, A2, ..., An;

let us think of the whole database as being described by a single universal relation schema R = {A1, A2, ... , An}.

We do not imply that we will actually store the database as a single universal table; we use this concept only in developing the formal theory of data dependencies.

Definition. A functional dependency, denoted by $X \rightarrow Y$, between two sets of attributes X and Y that are subsets of R specifies a *constraint* on the possible tuples that can form a relation state r of R. The constraint is that, for any two tuples $t1$ and $t2$ in r that have $t1[X] = t2[X]$, they must also have $t1[Y] = t2[Y]$.

Thus, *X* functionally determines *Y* in a relation schema *R* if, and only if, whenever two tuples of *r*(*R*) agree on their *X*-value, they must necessarily agree on their *Y* value.

Note the following:

■ If a constraint on *R* states that there cannot be more than one tuple with a given *X*-value in any relation instance *r*(*R*)—that is, *X* is a candidate key of *R*—this implies that $X \rightarrow Y$ for any subset of attributes *Y* of *R* (because the key constraint implies that no two tuples in any legal state *r*(*R*) will have the same value of *X*). If *X* is a candidate key of *R*, then $X \rightarrow R$.

■ If $X \rightarrow Y$ in *R*, this does not say whether or not $Y \rightarrow X$ in *R*.

For example,

$$\{\text{State, Driver\_license\_number}\} \rightarrow \text{Ssn}$$

Should hold for any adult in the United States and hence should hold whenever these attributes appear in a relation.

Consider the relation schema EMP_PROJ in Figure 15.3(b); from the semantics of the attributes and the relation, we know that the following functional dependencies should hold:

a. $\text{Ssn} \rightarrow \text{Ename}$

b. $\text{Pnumber} \rightarrow \{\text{Pname, Plocation}\}$

c. $\{\text{Ssn, Pnumber}\} \rightarrow \text{Hours}$

These functional dependencies specify that

(a) The value of an employee's Social Security number (Ssn) uniquely determines the employee name (Ename).

(b) The value of a project's number (Pnumber) uniquely determines the project name (Pname) and location (Plocation).

(c) A combination of Ssn and Pnumber values uniquely determines the number of hours the employee currently works on the project per week (Hours).

Alternatively,we say that Ename is functionally determined by (or functionally dependent on) Ssn, or *given a value of Ssn, we know the value of Ename*, and so on.

4.3 Inference Rules for Functional Dependencies

We denote by *F* the set of functional dependencies that are specified on relation schema *R*. Typically, the schema designer specifies the functional dependencies that are *semantically*

obvious; usually, however, numerous other functional dependencies hold in *all* legal relation instances among sets of attributes that can be derived from and satisfy the dependencies in *F*. Those other dependencies can be *inferred* or *deduced* from the FDs in *F*.

D**efinition**. *Formally, the set of all dependencies that include F as well as all dependencies that can be inferred from F is called the closure of F; it is denoted by F+.*

For example, suppose that we specify the following set *F* of obvious functional dependencies on the relation schema in Figure 15.3(a):

F = {Ssn → {Ename, Bdate, Address, Dnumber}, Dnumber → {Dname, Dmgr_ssn} }

Some of the additional functional dependencies that we can *infer* from *F* are the following:

Ssn → {Dname, Dmgr_ssn}

Ssn → Ssn

Dnumber → Dname

The following six rules IR1 through IR6 *Are well-known inference rules for functional dependencies, IR1 through IR3 are called Armstrong rules*

IR1 (reflexive rule)1: If X ⊇ Y, then X → Y.

IR2 (augmentation rule)2: {X → Y} |=XZ → YZ.

IR3 (transitive rule): {X → Y, Y → Z} |=X → Z.

IR4 (decomposition, or projective, rule): {X → YZ} |=X → Y.

IR5 (union, or additive, rule): {X → Y, X → Z} |=X → YZ.

IR6 (pseudotransitive rule): {X → Y,WY → Z} |=WX → Z.

Proof of IR1. Suppose that $X \supseteq Y$ and that two tuples $t1$ and $t2$ exist in some relation instance r of R such that $t1[X] = t2[X]$. Then $t1[Y] = t2[Y]$ because $X \supseteq Y$; hence, $X \rightarrow Y$ must hold in r.

Proof of IR2 (by contradiction). Assume that $X \rightarrow Y$ holds in a relation instance r of R but that $XZ \rightarrow YZ$ does not hold. Then there must exist two tuples $t1$ and $t2$ in r such that (1) $t1[X] = t2[X]$, (2) $t1[Y] = t2[Y]$, (3) $t1[XZ] = t2[XZ]$, and (4) $t1[YZ] \neq t2[YZ]$. This is not possible because from (1) and (3) we deduce (5) $t1[Z] = t2[Z]$, and from (2) and (5) we deduce (6) $t1[YZ] = t2[YZ]$, contradicting

(4).

Proof of IR3. Assume that (1) $X \rightarrow Y$ and (2) $Y \rightarrow Z$ both hold in a relation r. Then for any two tuples $t1$ and $t2$ in r such that $t1[X] = t2[X]$, we must have (3) $t1[Y] = t2[Y]$, from assumption (1); hence we must also have (4) $t1[Z] = t2[Z]$ from (3) and assumption (2); thus $X \rightarrow Z$ must hold in r.

Using similar proof arguments, we can prove the inference rules IR4 to IR6 and any additional valid inference rules. However, a simpler way to prove that an inference rule for functional dependencies is valid is to prove it by using inference rules that have already been shown to be valid. For example, we can prove IR4 through IR6 by using *IR1 through IR3* as follows.

Proof of IR4 (Using IR1 through IR3).

1. $X \rightarrow YZ$ (given).
2. $YZ \rightarrow Y$ (using IR1 and knowing that $YZ \supseteq Y$).
3. $X \rightarrow Y$ (using IR3 on 1 and 2).

Proof of IR5 (using IR1 through IR3).

1. $X \rightarrow Y$ (given).
2. $X \rightarrow Z$ (given).
3. $X \rightarrow XY$ (using IR2 on 1 by augmenting with X; notice that $XX = X$).
4. $XY \rightarrow YZ$ (using IR2 on 2 by augmenting with Y).
5. $X \rightarrow YZ$ (using IR3 on 3 and 4).

Proof of IR6 (using IR1 through IR3).

1. $X \rightarrow Y$ (given).
2. $WY \rightarrow Z$ (given).
3. $WX \rightarrow WY$ (using IR2 on 1 by augmenting with W).
4. $WX \rightarrow Z$ (using IR3 on 3 and 2).

4.4 Normal Forms Based on Primary Keys

Having introduced functional dependencies, we are now ready to use them to specify some aspects of the semantics of relation schemas. We assume that a set of functional dependencies is given for each relation, and that each relation has a designated primary key; this information combined with the tests (conditions) for normal forms drives the *normalization process* for relational schema design. Most practical relational design projects take one of the following two approaches:

■ perform a conceptual schema design using a conceptual model such as ER or EER and map the conceptual design into a set of relations

■ Design the relations based on external knowledge derived from an existing implementation of files or forms or reports

Normalization

The normalization process, as first proposed by Codd (1972a), takes a relation schema through a series of tests to certify whether it satisfies a certain normal form. Normalization of data can be considered a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of

(1) minimizing redundancy and (2) minimizing the insertion, deletion, and update anomalies

The normalization procedure provides database designers with the following:

■ A formal framework for analyzing relation schemas based on their keys and on the functional dependencies among their attributes

■ A series of normal form tests that can be carried out on individual relation schemas so that the relational database can be normalized to any desired

degree

Definition. The normal form of a relation refers to the highest normal form condition that it meets, and hence indicates the degree to which it has been normalized.

Practical Use of Normal Forms

Most practical design projects acquire existing designs of databases from previous designs, designs in legacy models, or from existing files. Normalization is carried out in practice so that the resulting designs are of high quality and meet the desirable properties stated previously. Although several higher normal forms have been defined, the practical utility of these normal forms becomes questionable when the constraints on which they are based are rare, and hard to understand or to detect by the database designers and users who must discover these constraints.

Definitions of Keys and Attributes

- **A superkey** of a relation schema $R = \{A1, A2, \ldots, An\}$ is a set of attributes $S \subseteq R$ with the property that no two tuples $t1$ and $t2$ in any legal relation state r of R will have $t1[S] = t2[S]$.
- **A key *K*** is a superkey with the additional property that removal of any attribute from K will cause K not to be a superkey any more.

The difference between a key and a superkey is that a key has to be *minimal;* that is, if we have a key $K = \{A1, A2, ..., Ak\}$ of R, then $K - \{Ai\}$ is not a key of R for any Ai, $1 \leq i \leq k$. In Figure 15.1, {Ssn} is a key for EMPLOYEE, whereas {Ssn}, {Ssn, Ename}, {Ssn, Ename, Bdate}, and any set of attributes that includes Ssn are all superkeys. If a relation schema has more than one key, each is called a candidate key.

One of the **candidate keys is *arbitrarily*** designated to be the primary key, and the others are called secondary keys.

- **Prime attribute**: An attribute of relation schema R is called a **prime attribute** of R if it is a member of *some candidate key* of R.
- **Nonprime attribute**: An attribute is called **nonprime** if it is not a prime attribute—that is, if it is not a member of any candidate key.

First Normal Form

First normal form **(1NF)** is now considered to be part of the formal definition of a relation in the basic (flat) relational model; historically, it was defined to disallow multivalued attributes, composite attributes, and their combinations.

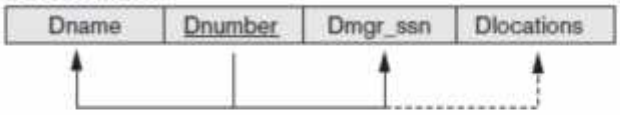
***1NF** states that the domain of an attribute must include only atomic (simple, indivisible) values and that the value of any attribute in a tuple must be a single value from the domain of that attribute.*

Consider the DEPARTMENT relation schema shown in Figure 15.1, whose primary key is Dnumber, and suppose that we extend it by including the Dlocations attribute as shown in Figure 15.9(a). We assume that each department can have *a number of* locations. The DEPARTMENT schema and a sample relation state are shown in Figure 15.9. As we can see, this is not in 1NF because Dlocations is not an atomic attribute, There are two ways we can look at the Dlocations attribute:

■ The domain of Dlocations contains atomic values, but some tuples can have a set of these values. In this case, Dlocations is not functionally dependent on the primary key Dnumber.

(a)

DEPARTMENT



(b)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	(Bellaire, Sugarland, Houston)
Administration	4	987654321	(Stafford)
Headquarters	1	888665555	(Houston)

(c)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Figure 15.9
Normalization into 1NF. (a) A relation schema that is not in 1NF. (b) Sample state of relation DEPARTMENT. (c) 1NF version of the same relation with redundancy.

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

■ The domain of Dlocations contains sets of values and hence is nonatomic. In this case, Dnumber → Dlocations because each set is considered a single member of the attribute domain

There are three main techniques to achieve first normal form for such a relation:

1. Remove the attribute Dlocations that violates 1NF and place it in a separate relation DEPT_LOCATIONS along with the primary key Dnumber of DEPARTMENT. The primary key of this relation is the combination {Dnumber, Dlocation},

2. Expand the key so that there will be a separate tuple in the original DEPARTMENT relation for each location of a DEPARTMENT, as shown in Figure 15.9(c).

3. If a *maximum number of values* is known for the attribute—for example, if it is known that *at most three locations* can exist for a department—replace the Dlocations attribute by three atomic attributes: Dlocation1, Dlocation2, and Dlocation3. This solution has the disadvantage of introducing *NULL values* if most departments have fewer than three locations.

First normal form also disallows multivalued attributes that are themselves composite. These are called nested relations because each tuple can have a relation *within it.*

Each tuple represents an employee entity, and a relation PROJS(Pnumber, Hours) *within each tuple* represents the employee's projects and the hours per week that employee works on each project. The schema of this EMP_PROJ relation can be represented as follows:

EMP_PROJ(Ssn, Ename, {PROJS(Pnumber, Hours)})

(a)

EMP_PROJ

		Projs	
Ssn	Ename	Pnumber	Hours

(b)

EMP_PROJ

Ssn	Ename	Pnumber	Hours
123456789	Smith, John B.	1	32.5
		2	7.5
666884444	Narayan, Ramesh K.	3	40.0
453453453	English, Joyce A.	1	20.0
		2	20.0
333445555	Wong, Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya, Alicia J.	30	30.0
		10	10.0
987987987	Jabbar, Ahmad V.	10	35.0
		30	5.0
987654321	Wallace, Jennifer S.	30	20.0
		20	15.0
888665555	Borg, James E.	20	NULL

(c)

EMP_PROJ1

Ssn	Ename

EMP_PROJ2

Ssn	Pnumber	Hours

Figure 15.10
Normalizing nested relations into 1NF. (a) Schema of the EMP_PROJ relation with a *nested relation* attribute PROJS. (b) Sample extension of the EMP_PROJ relation showing nested relations within each tuple. (c) Decomposition of EMP_PROJ into relations EMP_PROJ1 and EMP_PROJ2 by propagating the primary key.

Second Normal Form

Second normal form **(2NF)** is based on the concept of *full functional dependency.*

Full functional dependency :A functional dependency $X \rightarrow Y$ is a full functional dependency if removal of any attribute A from X means that the dependency does not hold any more; that is, for any attribute $A \varepsilon X$, $(X - \{A\})$ does *not* functionally determine Y.

In Figure 15.3(b), {Ssn, Pnumber} → Hours is a full dependency (neither Ssn → Hours nor Pnumber → Hours holds). However, the dependency {Ssn, Pnumber} → Ename is partial because Ssn → Ename holds.

Definition.: ***2NF*** *A relation schema R is in 2NF if every nonprime attribute A in R is fully functionally dependent on the primary key of R.*

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all. The EMP_PROJ relation in Figure 15.3(b) is in 1NF but is not in 2NF. The nonprime attribute Ename violates 2NF because of FD2, as do the nonprime attributes Pname and Plocation because of FD3. The functional dependencies FD2 and FD3 make Ename, Pname, and Plocation partially dependent

on the primary key {Ssn, Pnumber} of EMP_PROJ, thus violating the 2NF test. the functional dependencies FD1, FD2, and FD3 in Figure 15.3(b) lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure 15.11(a), each of which is in 2NF.

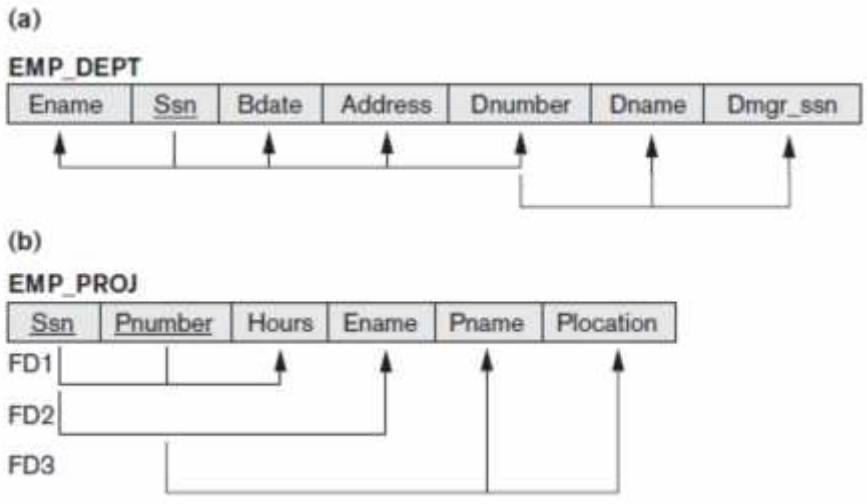


Figure 15.3
Two relation schemas suffering from update anomalies. (a) EMP_DEPT and (b) EMP_PROJ.

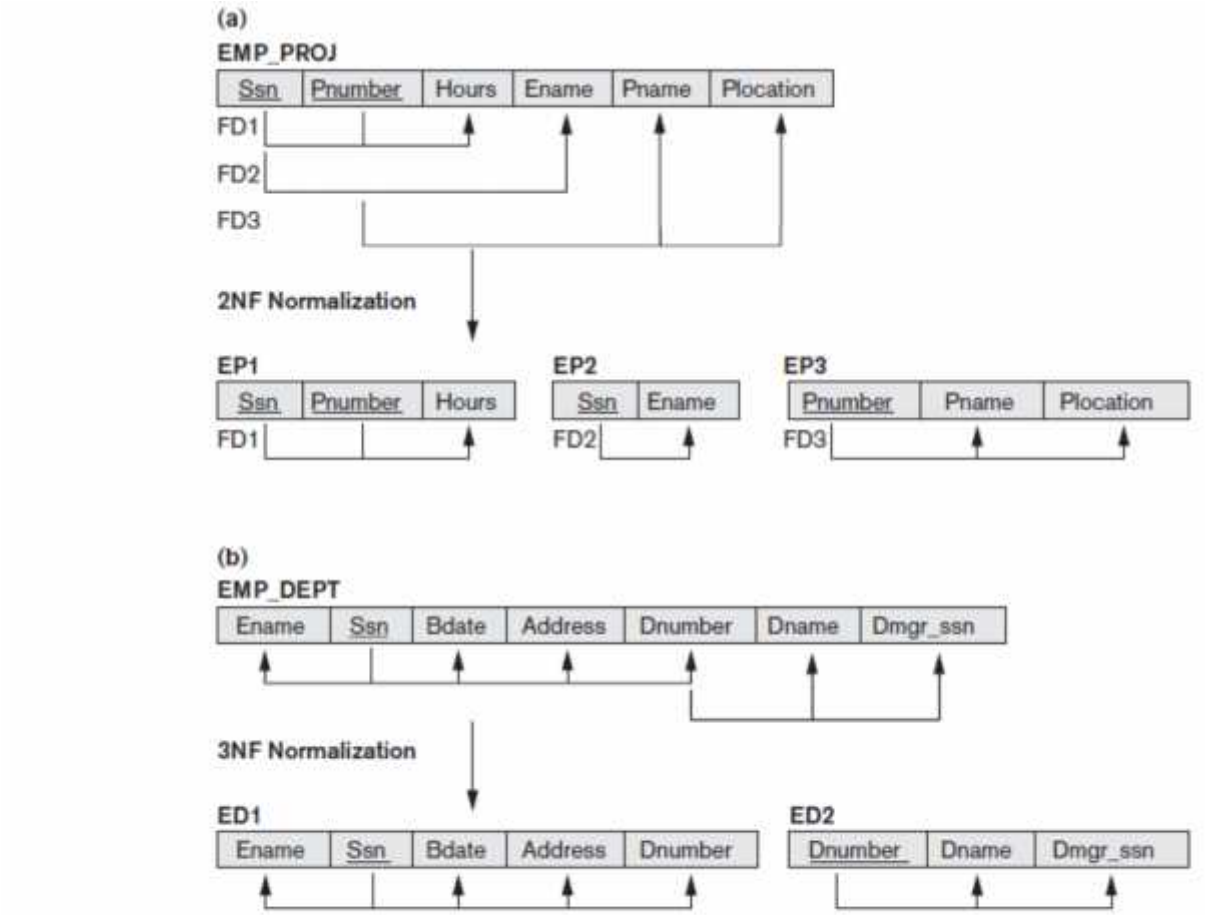


Figure 15.11
Normalizing into 2NF and 3NF. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.

Third Normal Form

Third normal form (3NF) is based on the concept of ***transitive dependency*.**

***Transitive dependency*:** A functional dependency $X \rightarrow Y$ in a relation schema R is a transitive dependency if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold.

The dependency Ssn $\rightarrow$ Dmgr_ssn is transitive through Dnumber in EMP_DEPT in Figure 15.3(a), because both the EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2 shown in Figure 15.11(b). Intuitively, we see that ED1 and ED2 represent independent entity facts about employees and departments. A NATURAL JOIN operation on ED1 and ED2 will recover the original relation EMP_DEPT without generating spurious tuples.

General Definitions of Second and Third Normal Forms

General Definition of Second Normal Form

Definition. A relation schema R is in second normal form (2NF) if every nonprime attribute A in R is not partially dependent on any key of R.

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are *part of* the primary key. If the primary key contains a single attribute, the test need not be applied at all.

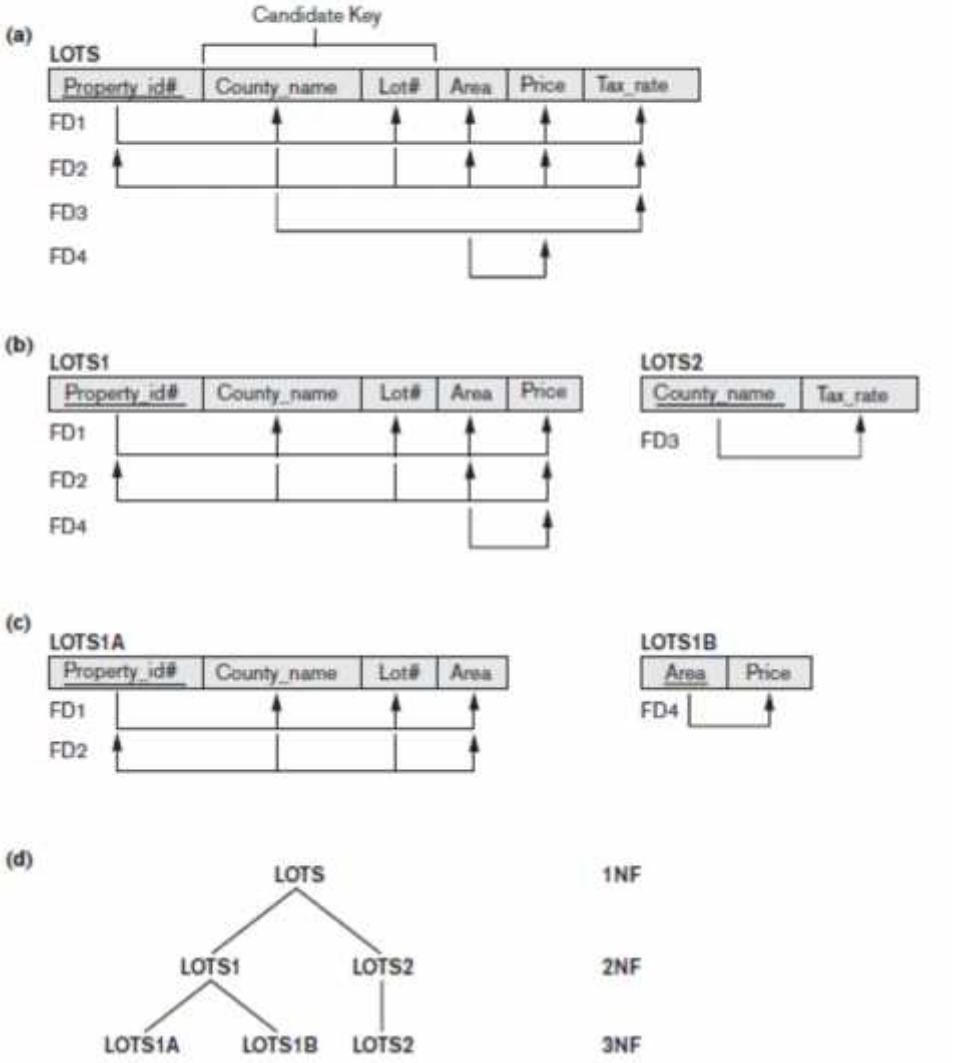
Consider the relation schema LOTS shown in Figure 15.12(a), which describes parcels of land for sale in various counties of a state. Suppose that there are two candidate keys: Property_id# and {County_name, Lot#}; that is, lot numbers are unique only within each county, but Property_id# numbers are unique across counties for the entire state.

Based on the two candidate keys Property_id# and {County_name, Lot#}, the functional dependencies FD1 and FD2 in Figure 15.12(a) hold.We choose Property_id# as the primary key, so it is underlined in Figure 15.12(a), but no special consideration will be given to this key over the other candidate key. Suppose that the following two additional functional dependencies hold in LOTS:

FD3: County_name → Tax_rate

FD4: Area → Price

Figure 15.12
Normalization into 2NF and 3NF. (a) The LOTS relation with its functional dependencies FD1 through FD4. (b) Decomposing into the 2NF relations LOTS1 and LOTS2. (c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B. (d) Summary of the progressive normalization of LOTS.



General Definition of Third Normal Form

A relation schema R is in third normal form (3NF) if, whenever a nontrivial functional dependency X→A holds in R, either (a) X is a superkey of R, or (b) A is a prime attribute of R.

According to this definition, LOTS2 (Figure 15.12(b)) is in 3NF. However, FD4 in LOTS1 violates 3NF because Area is not a superkey and Price is not a prime attribute in LOTS1. To normalize LOTS1 into 3NF, we decompose it into the relation schemas LOTS1A and LOTS1B shown in Figure 15.12(c).We construct LOTS1A by removing the attribute Price that violates 3NF from LOTS1 and placing it with Area (the lefthand side of FD4 that causes the transitive dependency) into another relation LOTS1B. Both LOTS1A and LOTS1B are in 3NF.

Two points are worth noting about this example and the general definition of 3NF:

■ LOTS1 violates 3NF because Price is transitively dependent on each of the candidate keys of LOTS1 via the nonprime attribute Area.

■ This general definition can be applied *directly* to test whether a relation schema is in 3NF; it does *not* have to go through 2NF first. If we apply the above 3NF definition to LOTS with the dependencies FD1 through FD4, we find that *both* FD3 and FD4 violate 3NF. Therefore, we could decompose LOTS into LOTS1A, LOTS1B, and LOTS2 directly. Hence, the transitive and partial dependencies that violate 3NF can be removed *in any order*.

Boyce-Codd Normal Form

Boyce-Codd normal form (BCNF) was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF. That is, every relation in BCNF is also in 3NF; however, a relation in 3NF is *not necessarily* in BCNF. Intuitively, we can see the need for a stronger normal form than 3NF by going back to the LOTS relation schema in

FD5: Area → County_name. If we add this to the other dependencies, the relation schema LOTS1A still is in 3NF because County_name is a prime attribute.

BCNF is a *stronger normal form* that would disallow LOTS1A and suggest the need for decomposing it.

Definition. *A relation schema R is in BCNF if whenever a nontrivial functional dependency X → A holds in R, then X is a superkey of R.*

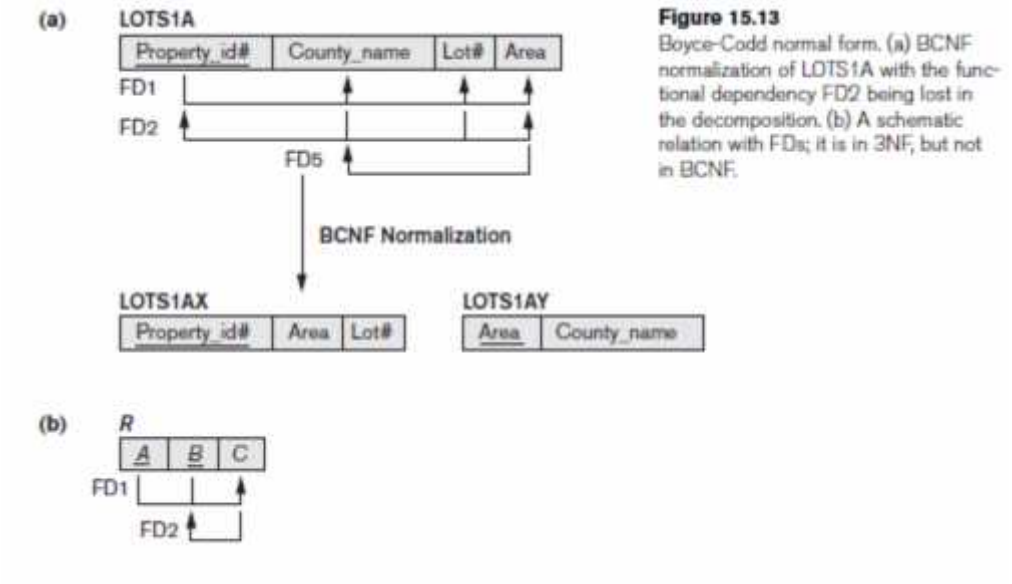


Figure 15.13
Boyce-Codd normal form. (a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition. (b) A schematic relation with FDs; it is in 3NF, but not in BCNF.

The formal definition of BCNF differs from the definition of 3NF in that condition (b) of 3NF, which allows A to be prime, is absent from BCNF. That makes BCNF a stronger normal form compared to 3NF.

In our example, FD5 violates BCNF in LOTS1A because AREA is not a superkey of LOTS1A. Note that FD5 satisfies 3NF in LOTS1A because County_name is a prime attribute (condition b), but this condition does not exist in the definition of BCNF.We can decompose LOTS1A into two BCNF relations LOTS1AX and LOTS1AY, shown in Figure 15.13(a). This decomposition loses the functional dependency FD2 because its attributes no longer coexist in the same relation after decomposition.

Functions and Procedures

Procedures and functions allow "business logic" to be stored in the database, and executed from SQL statements. For example, universities usually have many rules about how many courses a student can take in a given semester, the minimum number of courses a full-time instructor must teach in a year, the maximum number of majors a student can be enrolled in, and so on. SQL allows the definition of functions, procedures, and methods. These can be defined either by the procedural component of SQL, or by an external programming language such as Java, C, or C++.

Declaring and Invoking SQL Functions

Suppose that we want a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept count(dept name varchar(20))
returns integer
begin
declare d count integer;
select count(*) into d count
from instructor
where instructor.dept name= dept name
return d count;
end
```

select *dept name*, *budget*

from *instructor*

where *dept count*(*dept name*) > 12;

Stored procedures

A stored procedure in SQL is a type of code in SQL that can be stored for later use and can be used many times. So, whenever you need to execute the query, instead of calling it you can just call the stored procedure. Values can be passed through stored procedures.

```
CREATE PROCEDURE uspProductList
AS
BEGIN
   SELECT
      product_name,
      list_price
   FROM
      production.products
   ORDER BY
      product_name;
END;
```

Triggers in SQL

A trigger is a statement that the system executes automatically as a side effect of a modification to the database. To design a trigger mechanism, we must meet two requirements:

1. Specify when a trigger is to be executed. This is broken up into an *event* that causes the trigger to be checked and a *condition* that must be satisfied for trigger execution to proceed.

2. Specify the *actions* to be taken when the trigger executes.

A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.

```
create trigger [trigger_name]
[before | after]
{insert | update | delete}
on [table_name]
[for each row]
[trigger_body]
```

- *create trigger [trigger_name]: Creates or replaces an existing trigger with the trigger_name.*
- *[before | after]: This specifies when the trigger will be executed.*
- *{insert | update | delete}: This specifies the DML operation.*
- *on [table_name]: This specifies the name of the table associated with the trigger.*
- *[for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.*
- *[trigger_body]: This provides the operation to be performed as trigger is fired*

BEFORE and AFTER of Trigger:

BEFORE triggers run the trigger action before the triggering statement is run.
AFTER triggers run the trigger action after the triggering statement is run.

Example:

Given Student Report Database, in which student marks assessment is recorded. In such schema, create a trigger so that the total and average of specified marks is automatically inserted whenever a record is insert. Here, as trigger will invoke before record is inserted so, BEFORE Tag can be used.

```
create trigger stud_marks
before INSERT
on
Student
for each row
set Student.total = Student.subj1 + Student.subj2 + Student.subj3, Student.per = Student.total * 60 / 100;
```

Above SQL statement will create a trigger in the student database in which whenever subjects marks are entered, before inserting this data into the database, trigger will compute those two values and insert with the entered values. i.e.,

```
mysql> insert into Student values(0, "ABCDE", 20, 20, 20, 0, 0);
Query OK, 1 row affected (0.09 sec)
```

```
mysql> select * from Student;
+-----+-------+-------+-------+-------+-------+------+
| tid | name  | subj1 | subj2 | subj3 | total | per  |
+-----+-------+-------+-------+-------+-------+------+
| 100 | ABCDE |    20 |    20 |    20 |    60 |   36 |
+-----+-------+-------+-------+-------+-------+------+
1 row in set (0.00 sec)
```