
Relational Database Design by ER- and EER-to-Relational Mapping: Relational Database Design Using ER- to-Relational Mapping.

Basic SQL: SQL Data Definition and Data Types, Specifying Constraints in SQL, Basic retrieval queries in SQL, Insert, Delete and Update Statements in SQL, Additional features of SQL.

More SQL: Complex Queries, Triggers, Views, and Schema Modification: More complex SQL retrieval queries, Specifying constraints as assertions and actions as triggers, Views in SQL, Schema Change Statements in SQL.

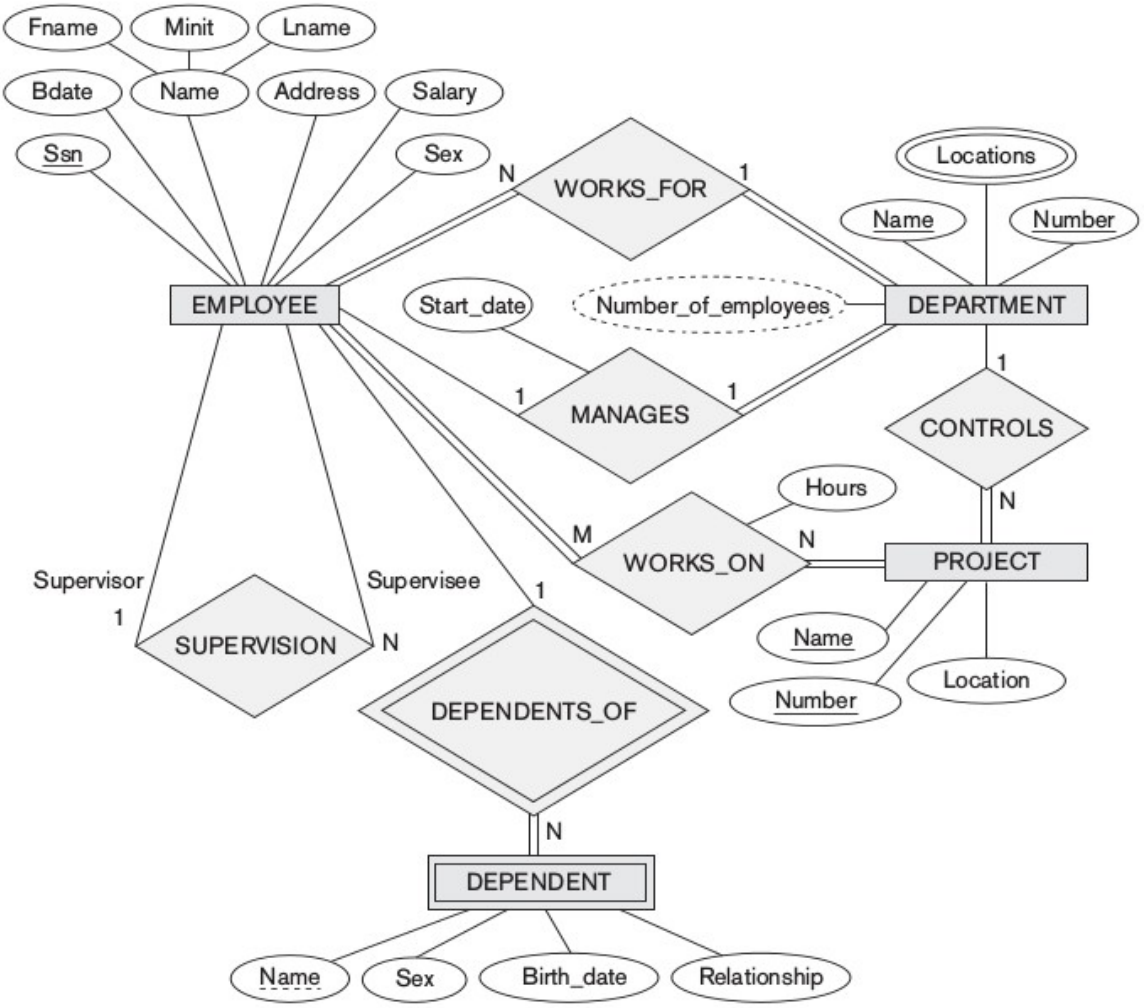
Introduction to SQL Programming Techniques: Database Stored Procedures and SQL/PSM.

Basics of Functional Dependencies and Normalization for Relational Databases: Informal design guidelines for relation schemas, Functional Dependencies, Normal Forms Based on Primary Keys, general definitions of Second and Third Normal Forms.

Relational Database Design Using ER-to-Relational Mapping

ER-to-Relational Mapping Algorithm

In this section we describe the steps of an algorithm for ER-to-relational mapping. We use the COMPANY database example to illustrate the mapping procedure.



Step 1: Mapping of Regular Entity Types. For each regular (strong) entity type E in the ER schema, create a relation R that includes all the simple attributes of E. Include only the simple component attributes of a composite attribute. Choose one of the key attributes of E as the primary key for R. If the chosen key of E is a composite, then the set of simple

attributes that form it will together form the primary key of R.

In our example, we create the relations EMPLOYEE, DEPARTMENT, and PROJECT to correspond to the regular entity types EMPLOYEE, DEPARTMENT, and PROJECT. The foreign key and relationship attributes, if any, are not included yet; they will be added during subsequent steps.

Step 2: Mapping of Weak Entity Types. For each weak entity type W in the ER schema with owner entity type E, create a relation R and include all simple attributes of W as attributes of R. In addition, include as foreign key attributes of R, the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s); this takes care of mapping the identifying relationship type of W. The primary key of R is the combination of the primary key(s) of the owner(s) and the partial key of the weak entity type W, if any.

In our example, we create the relation DEPENDENT in this step to correspond to the weak entity type DEPENDENT. We include the primary key Ssn of the EMPLOYEE relation —which corresponds to the owner entity type—as a foreign key attribute of DEPENDENT; we rename it Essn, although this is not necessary. The primary key of the DEPENDENT relation is combination {Essn,Dependent_name} because Dependent_name is the partial key of DEPENDENT.

Step 3: Mapping of Binary 1:1 Relationship Types. For each binary 1:1 relationship type R in the ER schema, identify the relations S and T that correspond to the entity types participating in R. There are three possible approaches: (1) the foreign key approach, (2) the merged relationship approach, and (3) the cross reference or relationship relation approach. The first approach is the most useful and should be followed unless special conditions exist

1. Foreign key approach: Choose one of the relations—S, say—and include as a foreign key in S the primary key of T. It is better to choose an entity type with total participation in R in the role of S. Include all the simple attributes (or simple components of composite attributes) of the 1:1 relationship type R as attributes of S.
   - In our example, we map the 1:1 relationship type MANAGES by choosing the participating entity type DEPARTMENT to serve in the role of S because its participation in the MANAGES relationship type is total . We include the primary key of the EMPLOYEE relation as foreign key in the DEPARTMENT relation and rename it Mgr_ssn. We also include the simple attribute Start_date of the MANAGES relationship type in the DEPARTMENT relation and rename it Mgr_start_date .
2. Merged relation approach: An alternative mapping of a 1:1 relationship type is to merge the two entity types and the relationship into a single relation. This is possible when both participations are total, as this would indicate that the two tables will have the exact same number of tuples at all times.
3. Cross-reference or relationship relation approach: The third option is to set up a third relation R for the purpose of cross-referencing the primary keys of the two relations S and T representing the entity types. As we will see, this approach is required for binary M:N relationships. The relation R will include the primary key attributes of S and T as foreign keys to S and T. The primary key of R will be one of the two foreign keys, and the other foreign key will be a unique key of R.

Step 4: Mapping of Binary 1:N Relationship Types. For each regular binary 1:N relationship type R, identify the relation S that represents the participating entity type at the N-side of the relationship type. Include as foreign key in S the primary key of the relation T that represents the other entity type participating in R; we do this because each entity instance on the N-side is related to at most one entity instance on the 1-side of the relationship type. Include any simple attributes (or simple components of composite

attributes) of the 1:N relationship type as attributes of S.

In our example, we now map the 1:N relationship types WORKS_FOR, CONTROLS, and SUPERVISION. For WORKS_FOR we include the primary key Dnumber of the DEPARTMENT relation as foreign key in the EMPLOYEE relation and call it Dno. For SUPERVISION we include the primary key of the EMPLOYEE relation as foreign key in the EMPLOYEE relation itself—because the relationship is recursive—and call it Super_ssn. The CONTROLS relationship is mapped to the foreign key attribute Dnum of PROJECT, which references the primary key Dnumber of the DEPARTMENT relation.

An alternative approach is to use the relationship relation (cross-reference) option as in the third option for binary 1:1 relationships. We create a separate relation R whose attributes are the primary keys of S and T, which will also be foreign keys to S and T. The primary key of R is the same as the primary key of S.

Step 5: Mapping of Binary M:N Relationship Types. For each binary M:N relationship type R, create a new relation S to represent R. Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types; their combination will form the primary key of S. Also include any simple attributes of the M:N relationship type (or simple components of composite attributes) as attributes of S. Notice that we cannot represent an M:N relationship type by a single foreign key attribute in one of the participating relations (as we did for 1:1 or 1:N relationship types) because of the M:N cardinality ratio; we must create a separate relationship relation S.

In our example, we map the M:N relationship type WORKS_ON by creating the relation WORKS_ON. We include the primary keys of the PROJECT and EMPLOYEE relations as foreign keys in WORKS_ON and rename them Pno and Essn, respectively. We also include an attribute Hours in WORKS_ON to represent the Hours attribute of the relationship type. The primary key of the WORKS_ON relation is the combination of the foreign key attributes {Essn, Pno}.

Step 7: Mapping of N-ary Relationship Types. For each n-ary relationship type R, where n > 2, create a new relation S to represent R. Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types. Also include any simple attributes of the n-ary relationship type (or simple components of composite attributes) as attributes of S. The primary key of S is usually a combination of all the foreign keys that reference the relations representing the participating entity types.

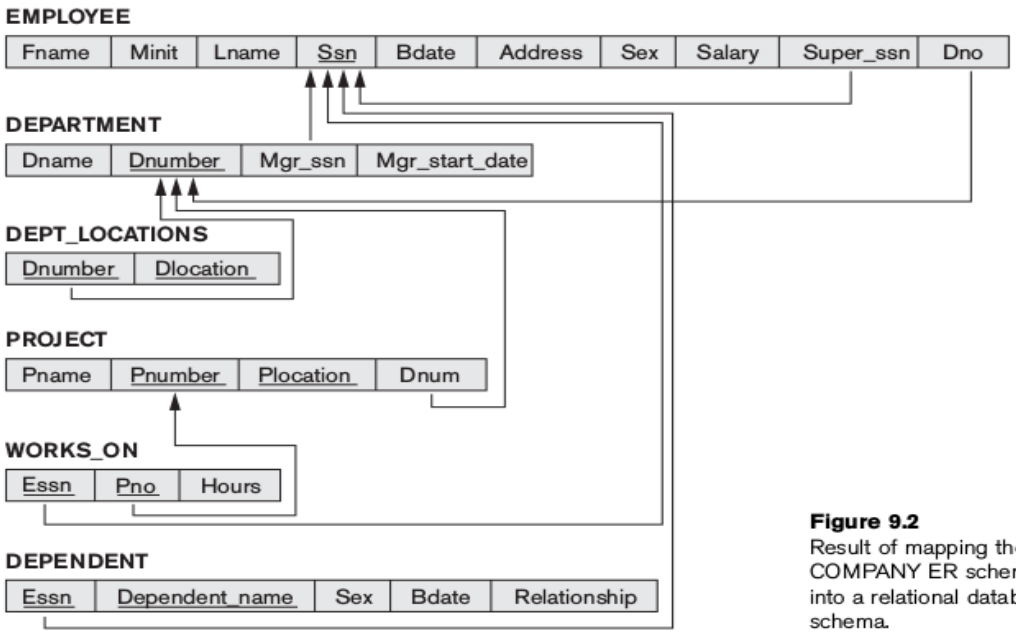


Figure 9.2
Result of mapping the COMPANY ER schema into a relational database schema.

SQL Data Definition and Data Types

SQL uses the terms table, row, and column for the formal relational model terms relation, tuple, and attribute, respectively. We will use the corresponding terms interchangeably. The main SQL command for data definition is the CREATE statement, which can be used to create schemas, tables (relations), and domains (as well as other constructs such as views, assertions, and triggers).

Schema and Catalog Concepts in SQL

Early versions of SQL did not include the concept of a relational database schema; all tables (relations) were considered part of the same schema. The concept of an SQL schema was incorporated in order to group together tables and other constructs that belong to the same database application. An SQL schema is identified by a schema name, and includes an authorization identifier to indicate the user or account who owns the schema, as well as descriptors for each element in the schema. Schema elements include tables, constraints, views, domains, and other constructs that describe the schema. A schema is created via the CREATE SCHEMA statement, which can include all the schema elements definitions.

For example, the following statement creates a schema called COMPANY, owned by the user with authorization identifier 'Jsmith'. Note that each statement in SQL ends with a semicolon.

CREATE SCHEMA COMPANY AUTHORIZATION 'Jsmith';

In addition to the concept of a schema, SQL uses the concept of a catalog—a named collection of schemas in an SQL environment. Integrity constraints such as referential integrity can be defined between relations only if they exist in schemas within the same catalog. Schemas within the same catalog can also share certain elements, such as domain definitions.

The CREATE TABLE Command in SQL

The CREATE TABLE command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints. The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and any attribute constraints, such as NOT NULL. The key, entity integrity, and referential integrity constraints can be specified within the CREATE TABLE statement after the attributes are declared, or they can be added later using the ALTER TABLE command.

Typically, the SQL schema in which the relations are declared is implicitly specified in the environment in which the CREATE TABLE statements are executed. Alternatively, we can explicitly attach the schema name to the relation name, separated by a period. For example, by writing

CREATE TABLE COMPANY.EMPLOYEE ...
rather than
CREATE TABLE EMPLOYEE ...

The relations declared through CREATE TABLE statements are called base tables (or base relations); this means that the relation and its tuples are actually created and stored as a file by the DBMS. In SQL, the attributes in a base table are considered to be ordered in the sequence in which they are specified in the CREATE TABLE statement. However, rows (tuples) are not considered to be ordered within a relation.

Attribute Data Types and Domains in SQL

The basic data types available for attributes include numeric, character string, bit string, Boolean, date, and time.

- **Numeric** data types include integer numbers of various sizes (INTEGER or INT, and SMALLINT) and floating-point(real) numbers of various precision (FLOAT or REAL, and DOUBLE PRECISION). Formatted numbers can be declared by using DECIMAL(i, j)—or DEC(i, j) or NUMERIC(i, j)—where i, the precision, is the total number of decimal digits and j, the scale, is the number of digits after the decimal point.
- **Character-string** data types are either fixed length— CHAR(n) or CHARACTER(n), where n is the number of characters—or varying length— VARCHAR(n) or CHAR VARYING(n) or CHARACTER VARYING(n), where n is the maximum number of characters. For fixed- length strings, a shorter string is padded with blank characters to the right. For example, if the value 'Smith' is for an attribute of type CHAR(10), it is padded with five blank characters to become 'Smith ' if needed.
- **Bit-string** data types are either of fixed length n— BIT(n)—or varying length— BIT VARYING (n), where n is the maximum number of bits. The default for n, the length of a character string or bit string, is 1. Literal bit strings are placed between single quotes but preceded by a B to distinguish them from character strings; for example, B '10101'.
- A **Boolean** data type has the traditional values of TRUE or FALSE. In SQL, because of the presence of NULL values, a three-valued logic is used, so a third possible value for a Boolean data type is UNKNOWN.
- The **DATE** data type has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD. The TIME data type has at least eight positions, with the components HOUR, MINUTE, and SECOND in the form HH:MM:SS. Literal values are represented by single-quoted strings preceded by the keyword DATE or TIME ; for example, DATE '2008-09- 27' or TIME '09:12:47'.
- A **timestamp** data type (TIMESTAMP) includes the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds and an optional WITH TIME ZONE qualifier. Literal values are represented by single quoted strings preceded by the keyword TIMESTAMP, with a blank space between date and time; for example, TIMESTAMP '2008-09-27 09:12:47.648302'.
- Another data type related to DATE, TIME, and TIMESTAMP is the **INTERVAL** data type. This specifies an interval—a relative value that can be used to increment or decrement an absolute value of a date, time, or timestamp.

We can create a domain SSN_TYPE by the following statement:

CREATE DOMAIN SSN_TYPE AS CHAR(9) ;

We can use SSN_TYPE in place of CHAR (9). A domain can also have an optional default specification via a DEFAULT clause, as we discuss later for attributes.

Specifying Constraints in SQL

This section describes the basic constraints that can be specified in SQL as part of table creation. These include key and referential integrity constraints, restrictions on attribute domains and NULLs, and constraints on individual tuples within a relation.

Specifying Attribute Constraints and Attribute Defaults

Because SQL allows NULLs as attribute values, a constraint NOT NULL may be specified if NULL is not permitted for a particular attribute. This is always implicitly specified for the attributes that are part of the primary key of each relation, but it can be specified for any other attributes whose values are required not to be NULL.

It is also possible to define a default value for an attribute by appending the clause DEFAULT <value> to an attribute definition. The default value is included in any new tuple

if an explicit value is not provided for that attribute. If no default clause is specified, the default default value is NULL for attributes that do not have the NOT NULL constraint.

Another type of constraint can restrict attribute or domain values using the CHECK clause following an attribute or domain definition. For example, suppose that department numbers are restricted to integer numbers between 1 and 20; then, we can change the attribute declaration of Dnumber in the DEPARTMENT table to the following:

Dnumber INT NOT NULL CHECK (Dnumber > 0 AND Dnumber < 21);

```
CREATE TABLE EMPLOYEE
(
        Fname VARCHAR(15) NOT NULL,
        Minit CHAR ,
        Lname VARCHAR(15) NOT NULL,
        Ssn CHAR(9) NOT NULL,
        Bdate DATE,
        Address VARCHAR(30),
        Sex CHAR,
        Salary DECIMAL(10 , 2),
        Super_ssn CHAR(9),
        Dno INT NOT NULL DEFAULT 1,
        CONSTRAINT EMPPK
        PRIMARY KEY (Ssn),
        CONSTRAINT EMPSUPERFK
        FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn)
        ON DELETE SET NULL ON UPDATE CASCADE,
        CONSTRAINT EMPDEPTFK
        FOREIGN KEY (Dno) REFERENCES DEPARTMENT(Dnumber)
        ON DELETE SET DEFAULT    ON UPDATE CASCADE
 ) ;

CREATE TABLE DEPARTMENT
(
        Dname VARCHAR(15) NOT NULL,
        Dnumber INT NOT NULL,
        Mgr_ssn CHAR(9) NOT NULL DEFAULT '888665555',
        Mgr_start_date DATE,
        CONSTRAINT DEPTPK
        PRIMARY KEY (Dnumber),
        CONSTRAINT DEPTSK
        UNIQUE (Dname),
        CONSTRAINT DEPTMGRFK
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn)
        ON DELETE SET DEFAULT ON UPDATE CASCADE
) ;
```

The CHECK clause can also be used in conjunction with the CREATE DOMAIN statement. For example, we can write the following statement:

CREATE DOMAIN D_NUM AS INTEGER CHECK (D_NUM > 0 AND D_NUM < 21);

We can then use the created domain D_NUM as the attribute type for all attributes that refer to department numbers, such as Dnumber of DEPARTMENT, Dnum of PROJECT, Dno of EMPLOYEE, and so on.

Specifying Key and Referential Integrity Constraints

The PRIMARY KEY clause specifies one or more attributes that make up the primary key of a relation. If a primary key has a single attribute, the clause can follow the attribute directly. For example, the primary key of DEPARTMENT can be specified as follows:

Dnumber INT PRIMARY KEY ;

The UNIQUE clause specifies alternate (secondary) keys, as illustrated in the DEPARTMENT table declarations. The UNIQUE clause can also be specified directly for a secondary key if the secondary key is a single attribute, as in the following example:

Dname VARCHAR(15) UNIQUE ;

Referential integrity is specified via the FOREIGN KEY clause, a referential integrity constraint can be violated when tuples are inserted or deleted, or when a foreign key or primary key attribute value is modified. The default action that SQL takes for an integrity violation is to reject the update operation that will cause a violation, which is known as the RESTRICT option. However, the schema designer can specify an alternative action to be taken by attaching a referential triggered action clause to any foreign key constraint. The options include SET NULL, CASCADE , and SET DEFAULT. An option must be qualified with either ON DELETE or ON UPDATE. The database designer chooses ON DELETE SET NULL and ON UPDATE CASCADE for the foreign key Super_ssn of EMPLOYEE . This means that if the tuple for a supervising employee is deleted, the value of Super_ssn is automatically set to NULL for all employee tuples that were referencing the deleted employee tuple. On the other hand, if the Ssn value for a supervising employee is updated, the new value is cascaded to Super_ssn for all employee tuples referencing the updated employee tuple.

In general, the action taken by the DBMS for SET NULL or SET DEFAULT is the same for both ON DELETE and ON UPDATE: The value of the affected referencing attributes is changed to NULL for SET NULL and to the specified default value of the referencing attribute for SET DEFAULT. The action for CASCADE ON DELETE is to delete all the referencing tuples, whereas the action for CASCADE ON UPDATE is to change the value of the referencing foreign key attribute(s) to the updated (new) primary key value for all the referencing tuples.

Giving Names to Constraints

The names of all constraints within a particular schema must be unique. A constraint name is used to identify a particular constraint in case the constraint must be dropped later and replaced with another constraint.

Specifying Constraints on Tuples Using CHECK

In addition to key and referential integrity constraints, which are specified by special keywords, other table constraints can be specified through additional CHECK clauses at the end of a CREATE TABLE statement. These can be called tuple-based constraints because they apply to each tuple individually and are checked whenever a tuple is inserted or modified. For example, suppose that the DEPARTMENT table in had an additional attribute Dept_create_date, which stores the date when the department was created. Then we could add the following CHECK clause at the end of the CREATE TABLE statement for the DEPARTMENT table to make sure that a manager's start date is later than the department creation date.

CHECK (Dept_create_date <= Mgr_start_date);

The CHECK clause can also be used to specify more general constraints using the CREATE ASSERTION statement of SQL.

Basic Retrieval Queries in SQL

SQL has one basic statement for retrieving information from a database: the SELECT statement. The SELECT statement is not the same as the SELECT operation of relational algebra.

Before proceeding, we must point out an important distinction between SQL and the formal relational model: SQL allows a table (relation) to have two or more tuples that are identical in all their attribute values. Hence, in general, an SQL table is not a set of tuples, because a set does not allow two identical members; rather, it is a multiset (sometimes called a bag) of tuples. Some SQL relations are constrained to be sets because a key constraint has been declared or because the DISTINCT option has been used with the SELECT statement.

The SELECT-FROM-WHERE Structure of Basic SQL Queries

The basic form of the SELECT statement, sometimes called a mapping or a select-from-where block, is formed of the three clauses SELECT, FROM, and WHERE and has the following form:

SELECT <attribute list>
FROM <table list>
WHERE<condition>;

where

- <attribute list> is a list of attribute names whose values are to be retrieved by the query.
- <table list> is a list of the relation names required to process the query.
- <condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

In SQL, the basic logical comparison operators for comparing attribute values with one another and with literal constants are =, <, <=, >, >=, and <>. These correspond to the relational algebra operators =, <, ≤, >, ≥, and ≠, respectively. SQL has additional comparison operators that we will present gradually. We illustrate the basic SELECT statement in SQL with some sample queries.

- Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.

```
SELECT Bdate, Address
FROM EMPLOYEE
WHERE Fname ='John' AND Minit ='B' AND Lname ='Smith';
```

This query involves only the EMPLOYEE relation listed in the FROM clause. The query selects the individual EMPLOYEE tuples that satisfy the condition of the WHERE clause, then projects the result on the Bdate and Address attributes listed in the SELECT clause.

The SELECT clause of SQL specifies the attributes whose values are to be retrieved, which are called the projection attributes, and the WHERE clause specifies the Boolean condition that must be true for any retrieved tuple, which is known as the selection condition.

We can think of an implicit tuple variable or iterator in the SQL query ranging or looping over each individual tuple in the EMPLOYEE table and evaluating the condition in the WHERE clause. Only those tuples that satisfy the condition—that is, those tuples for which the condition evaluates to TRUE after substituting their corresponding attribute values—are selected.

- Retrieve the name and address of all employees who work for the 'Research'

department.
SELECT Fname , Lname , Address
FROM EMPLOYEE , DEPARTMENT
WHERE Dname ='Research' AND Dnumber = Dno;

In the WHERE clause, the condition Dname = 'Research' is a selection condition that chooses the particular tuple of interest in the DEPARTMENT table, because Dname is an attribute of DEPARTMENT. The condition Dnumber = Dno is called a join condition, because it combines two tuples: one from DEPARTMENT and one from EMPLOYEE, whenever the value of Dnumber in DEPARTMENT is equal to the value of Dno in EMPLOYEE.

A query that involves only selection and join conditions plus projection attributes is known as a select-project-join query. The next example is a select-project-join query with two join conditions.

- For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.
  SELECT Pnumber, Dnum, Lname, Address, Bdate
  FROM PROJECT, DEPARTMENT, EMPLOYEE
  WHERE Dnum = Dnumber AND Mgr_ssn = Ssn AND Plocation ='Stafford';

The join condition Dnum = Dnumber relates a project tuple to its controlling department tuple, whereas the join condition Mgr_ssn = Ssn relates the controlling department tuple to the employee tuple who manages that department. Each tuple in the result will be a combination of one project, one department, and one employee that satisfies the join conditions. The projection attributes are used to choose the attributes to be displayed from each combined tuple.

Ambiguous Attribute Names, Aliasing, Renaming, and Tuple Variables

In SQL, the same name can be used for two (or more) attributes as long as the attributes are in different relations. If this is the case, and a multitable query refers to two or more attributes with the same name, we must qualify the attribute name with the relation name to prevent ambiguity. This is done by prefixing the relation name to the attribute name and separating the two by a period.

To illustrate this, suppose that in previous query the Dno and Lname attributes of the EMPLOYEE relation were called Dnumber and Name, and the Dname attribute of DEPARTMENT was also called Name; then, to prevent ambiguity, query would be rephrased as shown below. We must prefix the attributes Name and Dnumber in to specify which ones we are referring to, because the same attribute names are used in both relations:

SELECT Fname,EMPLOYEE.Name, Address
FROM EMPLOYEE , DEPARTMENT
WHERE DEPARTMENT.Name ='Research' AND
DEPARTMENT.Dnumber=EMPLOYEE.Dnumber ;

Fully qualified attribute names can be used for clarity even if there is no ambiguity in attribute names. We can also create an alias for each table name to avoid repeated typing of long table names.

SELECT EMPLOYEE.Fname, EMPLOYEE.LName, EMPLOYEE.Address
FROM EMPLOYEE , DEPARTMENT

```
WHERE DEPARTMENT.DName ='Research' AND
DEPARTMENT.Dnumber = EMPLOYEE.Dno ;
```

The ambiguity of attribute names also arises in the case of queries that refer to the same relation twice, as in the following example.

- For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

```
SELECT E.Fname, E.Lname, S.Fname, S.Lname
FROM EMPLOYEE AS E , EMPLOYEE AS S
WHERE E.Super_ssn = S.Ssn ;
```

In this case, we are required to declare alternative relation names E and S , called aliases or tuple variables, for the EMPLOYEE relation. An alias can follow the keyword AS , as shown above, or it can directly follow the relation name—for example,
by writing EMPLOYEE E, EMPLOYEE S in the FROM clause. It is also possible to rename the relation attributes within the query in SQL by giving them aliases. For example, if we write

```
EMPLOYEE AS E ( Fn , Mi , Ln , Ssn , Bd , Addr , Sex , Sal , Sssn , Dno )
```

in the FROM clause, Fn becomes an alias for Fname, Mi for Minit, Ln for Lname, and so on. In the above query we can think of E and S as two different copies of the EMPLOYEE relation; the first E, represents employees in the role of supervisees or subordinates; the second S, represents employees in the role of supervisors.

Whenever one or more aliases are given to a relation, we can use these names to represent different references to that same relation. This permits multiple references to the same relation within a query. We can use this alias-naming mechanism in any SQL query to specify tuple variables for every table in the WHERE clause, whether or not the same relation needs to be referenced more than once.

```
SELECT E.Fname, E.LName , E.Address
FROM EMPLOYEE E , DEPARTMENT D
WHERE D.DName = ' Research ' AND D.Dnumber = E.Dno ;
```

Unspecified WHERE Clause and Use of the Asterisk

A missing WHERE clause indicates no condition on tuple selection; hence, all tuples of the relation specified in the FROM clause qualify and are selected for the query result. For example, Query below selects all EMPLOYEE Ssns

```
SELECT  Ssn
FROM EMPLOYEE ;
```

If more than one relation is specified in the FROM clause and there is no WHERE clause, then the CROSS PRODUCT —all possible tuple combinations—of these relations is selected. Query below selects all combinations of an EMPLOYEE Ssn and a DEPARTMENT Dname, regardless of whether the employee works for the department or not

```
SELECT Ssn , Dname
FROM  EMPLOYEE , DEPARTMENT ;
```

To retrieve all the attribute values of the selected tuples, we do not have to list the attribute names explicitly in SQL; we just specify an asterisk (*), which stands for all the attributes. For example, query below retrieves all the attribute values of any EMPLOYEE

who works in DEPARTMENT number 5:

```
SELECT  *
FROM EMPLOYEE
WHERE Dno =5;
```

Query below retrieves all the attributes of an EMPLOYEE and the attributes of the DEPARTMENT in which he or she works, for every employee of the 'Research' department

```
SELECT *
FROM EMPLOYEE , DEPARTMENT
WHERE Dname = ' Research ' AND Dno = Dnumber ;
```

Tables as Sets in SQL

As we mentioned earlier, SQL usually treats a table not as a set but rather as a multiset; duplicate tuples can appear more than once in a table, and in the result of a query. SQL does not automatically eliminate duplicate tuples in the results of queries, for the following reasons:

- Duplicate elimination is an expensive operation. One way to implement it is to sort the tuples first and then eliminate duplicates.
- The user may want to see duplicate tuples in the result of a query.
- When an aggregate function is applied to tuples, in most cases we do not want to eliminate duplicates.

An SQL table with a key is restricted to being a set, since the key value must be distinct in each tuple. If we do want to eliminate duplicate tuples from the result of an SQL query, we use the keyword DISTINCT in the SELECT clause, meaning that only distinct tuples should remain in the result. In general, a query with SELECT DISTINCT eliminates duplicates, whereas a query with SELECT ALL does not. Specifying SELECT with neither ALL nor DISTINCT —as in our previous examples— is equivalent to SELECT ALL .

For example, Query below retrieves the salary of every employee; if several employees have the same salary, that salary value will appear as many times in the result of the query.

```
SELECT ALL Salary
FROM  EMPLOYEE ;
```

If we are interested only in distinct salary values, we want each value to appear only once, regardless of how many employees earn that salary. By using the keyword DISTINCT as in Query below:

```
SELECT DISTINCT Salary
FROM  EMPLOYEE ;
```

SQL has directly incorporated some of the set operations from mathematical set theory, which are also part of relational algebra. There are set union (UNION), set difference (EXCEPT), and set intersection (INTERSECT) operations. The relations resulting from these set operations are sets of tuples; that is, duplicate tuples are eliminated from the result. The next example illustrates the use of UNION.

- Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

```
(SELECT DISTINCT Pnumber
FROM PROJECT , DEPARTMENT , EMPLOYEE
WHERE Dnum = Dnumber AND Mgr_ssn = Ssn AND Lname ='Smith' )
UNION
(SELECT DISTINCT Pnumber
FROM PROJECT , WORKS_ON , EMPLOYEE
WHERE Pnumber = Pno AND Essn = Ssn AND Lname ='Smith' );
```

The first SELECT query retrieves the projects that involve a 'Smith' as manager of the department that controls the project, and the second retrieves the projects that involve a 'Smith' as a worker on the project.

SQL also has corresponding multiset operations, which are followed by the keyword ALL (UNION ALL, EXCEPT ALL, INTERSECT ALL). Their results are multisets (duplicates are not eliminated).

Substring Pattern Matching and Arithmetic Operators

The first feature allows comparison conditions on only parts of a character string, using the LIKE comparison operator. This can be used for string pattern matching. Partial strings are specified using two reserved characters: % replaces an arbitrary number of zero or more characters, and the underscore (_) replaces a single character. For example, consider the following query.

- Retrieve all employees whose address is in Houston, Texas.

```
SELECT Fname , Lname
FROM EMPLOYEE
WHERE Address LIKE '%Houston,TX%';
```

- To retrieve all employees who were born during the 1950s.

```
SELECT Fname , Lname
FROM EMPLOYEE
WHERE Bdate LIKE '195 _ _ _ _ _ _ _';
```

If an underscore or % is needed as a literal character in the string, the character should be preceded by an escape character, which is specified after the string using the keyword ESCAPE.

For example, 'AB_CD\%EF' ESCAPE '\' represents the literal string 'AB_CD%EF' because \ is specified as the escape character. Any character not used in the string can be chosen as the escape character.

Another feature allows the use of arithmetic in queries. The standard arithmetic operators for addition (+), subtraction (–), multiplication (*), and division (/) can be applied to numeric values or attributes with numeric domains. For example, suppose that we want to see the effect of giving all employees who work on the 'ProductX' project a 10 percent raise;

```
SELECT E.Fname , E.Lname , 1.1 * E.Salary AS Increased_sal
FROM EMPLOYEE AS E , WORKS_ON AS W , PROJECT AS P
WHERE E.Ssn = W.Essn AND W.Pno = P.Pnumber AND P.Pname ='ProductX';
```

This example also shows how we can rename an attribute in the query result using AS in the SELECT clause.

Another comparison operator, which can be used for convenience, is BETWEEN , which is illustrated in Query below.

- Retrieve all employees in department 5 whose salary is between $30,000 and $40,000.
  SELECT *
  FROM EMPLOYEE
  WHERE (Salary BETWEEN 30000 AND 40000) AND Dno = 5;

The condition (Salary BETWEEN 30000 AND 40000) is equivalent to the condition ((Salary >= 30000) AND (Salary <= 40000)).

Ordering of Query Results

SQL allows the user to order the tuples in the result of a query by the values of one or more of the attributes that appear in the query result, by using the ORDER BY clause. This is illustrated below.

- Retrieve a list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by last name, then first name.
  SELECT D.Dname, E.Lname, E.Fname, P.Pname
  FROM DEPARTMENT D, EMPLOYEE E, WORKS_ON W, PROJECT P
  WHERE D.Dnumber = E.Dno AND E.Ssn = W.Essn AND W.Pno = P.Pnumber
  ORDER BY D.Dname , E.Lname , E.Fname ;

The default order is in ascending order of values. We can specify the keyword DESC if we want to see the result in a descending order of values. The keyword ASC can be used to specify ascending order explicitly. For example, if we want descending alphabetical order on Dname and ascending order on Lname, Fname, the ORDER BY clause can be written as

ORDER BY D.Dname DESC, E.Lname ASC, E.Fname ASC

Discussion and Summary of Basic SQL Retrieval Queries

A simple retrieval query in SQL can consist of up to four clauses, but only the first two — SELECT and FROM —are mandatory. The clauses are specified in the following order, with the clauses between square brackets [...] being optional:

SELECT <attribute list>
FROM <table list>
[WHERE <condition>]
[ORDER BY <attribute list>];

The SELECT clause lists the attributes to be retrieved, and the FROM clause specifies all relations (tables) needed in the simple query. The WHERE clause identifies the conditions for selecting the tuples from these relations, including join conditions if needed. ORDER BY specifies an order for displaying the results of a query.

INSERT, DELETE, and UPDATE Statements in SQL

In SQL, three commands can be used to modify the database: INSERT , DELETE , and UPDATE.

The INSERT Command

In its simplest form, INSERT is used to add a single tuple to a relation. We must specify the relation name and a list of values for the tuple. The values should be listed in the same order in which the corresponding attributes were specified in the CREATE TABLE command. For example, to add a new tuple to the EMPLOYEE relation:

INSERT INTO EMPLOYEE VALUES (‘Richard’, ‘K’, ‘Marini’, ‘653298653’, ‘1962-12-30’, ‘98 Oak Forest, Katy, TX’, ‘M’, 37000, ‘653298653’, 4);

A second form of the INSERT statement allows the user to specify explicit attribute names that correspond to the values provided in the INSERT command. This is useful if a relation has many attributes but only a few of those attributes are assigned values in the new tuple. However, the values must include all attributes with NOT NULL specification and no default value. Attributes with NULL allowed or DEFAULT values are the ones that can be left out. For example, to enter a tuple for a new EMPLOYEE for whom we know only the Fname, Lname, Dno, and Ssn attributes, we can use the following query:

INSERT INTO EMPLOYEE (Fname, Lname , Dno , Ssn) VALUES ('Richard', 'Marini', 4, '653298653');

Attributes not specified are set to their DEFAULT or to NULL.

A variation of the INSERT command inserts multiple tuples into a relation in conjunction with creating the relation and loading it with the result of a query. For example, to create a temporary table that has the employee last name, project name, and hours per week for each employee working on a project, we can write the statements below:

- CREATE TABLE WORKS_ON_INFO
  (
  Emp_name VARCHAR (15),
  Proj_name VARCHAR (15),
  Hours_per_week DECIMAL (3,1)
  );

- INSERT INTO WORKS_ON_INFO (Emp_name , Proj_name, Hours_per_week)
  SELECT E.Lname , P.Pname , W.Hours
  FROM PROJECT P , WORKS_ON W , EMPLOYEE E
  WHERE P.Pnumber = W.Pno AND W.Essn = E.Ssn ;

A table WORKS_ON_INFO is created and is loaded with the joined information retrieved from the database. Notice that the WORKS_ON_INFO table may not be up-to-date; that is, if we update any of the PROJECT, WORKS_ON, or EMPLOYEE relations then WORKS_ON_INFO may become outdated. We have to create a view to keep such a table up-to-date.

The DELETE Command

The DELETE command removes tuples from a relation. It includes a WHERE clause, similar to that used in an SQL query, to select the tuples to be deleted. Tuples are explicitly deleted from only one table at a time. However, the deletion may propagate to tuples in other relations if referential triggered actions are specified in the referential integrity constraints of the DDL. Depending on the number of tuples selected by the condition in the WHERE clause, zero, one, or several tuples can be deleted by a single DELETE command. A missing WHERE clause specifies that all tuples in the relation are to be deleted; however, the table remains in the database as an empty table. We must use the DROP TABLE command to remove the table definition. The DELETE commands shown below if applied independently to the database, will delete zero, one, four, and all tuples, respectively, from the EMPLOYEE relation:

- DELETE FROM EMPLOYEE WHERE Lname ='Brown';
- DELETE FROM EMPLOYEE WHERE Ssn ='123456789';
- DELETE FROM EMPLOYEE WHERE Dno =5;
- DELETE FROM EMPLOYEE ;

The UPDATE Command

The UPDATE command is used to modify attribute values of one or more selected tuples. As in the DELETE command, a WHERE clause in the UPDATE command selects the tuples to be modified from a single relation. However, updating a primary key value may propagate to the foreign key values of tuples in other relations if such a referential triggered action is specified in the referential integrity constraints of the DDL. An additional SET clause in the UPDATE command specifies the attributes to be modified and their new values.

- For example, to change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively, we use:

```
UPDATE PROJECT
SET Plocation = 'Bellaire', Dnum = 5
WHERE Pnumber =10;
```

Several tuples can be modified with a single UPDATE command. An example is to give all employees in the 'Research' department a 10 percent raise in salary. In this request, the modified Salary value depends on the original Salary value in each tuple, so two references to the Salary attribute are needed. In the SET clause, the reference to the Salary attribute on the right refers to the old Salary value before modification, and the one on the left refers to the new Salary value after modification:

```
UPDATE EMPLOYEE
SET Salary = Salary * 1.1
WHERE Dno = 5;
```

It is also possible to specify NULL or DEFAULT as the new attribute value. Notice that each UPDATE command explicitly refers to a single relation only. To modify multiple relations, we must issue several UPDATE commands.

More Complex SQL Retrieval Queries

Comparisons Involving NULL and Three-Valued Logic

SQL has various rules for dealing with NULL values. NULL is used to represent a missing value, but that it usually has one of three different interpretations—value unknown (exists but is not known), value not available (exists but is purposely withheld), or value not applicable (the attribute is undefined for this tuple). Consider the following examples to illustrate each of the meanings of NULL .

1. Unknown value. A person's date of birth is not known, so it is represented by NULL in the database.
2. Unavailable or withheld value. A person has a home phone but does not want it to be listed, so it is withheld and represented as NULL in the database.
3. Not applicable attribute. An attribute Last College Degree would be NULL for a person who has no college degrees because it does not apply to that person.

It is often not possible to determine which of the meanings is intended; for example, a NULL for the home phone of a person can have any of the three meanings. Hence, SQL does not distinguish between the different meanings of NULL .

In general, each individual NULL value is considered to be different from every other NULL value in the various database records. When a NULL is involved in a comparison operation, the result is considered to be UNKNOWN (it may be TRUE or it may be FALSE). Hence, SQL uses a three-valued logic with values TRUE, FALSE, and UNKNOWN instead

of the standard two-valued (Boolean) logic with values TRUE or FALSE.

In Tables (a) and (b), the rows and columns represent the values of the results of comparison conditions, which would typically appear in the WHERE clause of an SQL query. Each expression result would have a value of TRUE, FALSE , or UNKNOWN. The result of combining the two values using the AND logical connective is shown by the entries in Table (a). Table (b) shows the result of using the OR logical connective. For example, the result of (FALSE AND UNKNOWN) is FALSE, whereas the result of (FALSE OR UNKNOWN) is UNKNOWN. Table (c) shows the result of the NOT logical operation.

(a)

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

(b)

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

(c)

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

In select-project-join queries, the general rule is that only those combinations of tuples that evaluate the logical expression in the WHERE clause of the query to TRUE are selected. Tuple combinations that evaluate to FALSE or UNKNOWN are not selected. Rather than using = or <> to compare an attribute value to NULL, SQL uses the comparison operators IS or IS NOT. This is because SQL considers each NULL value as being distinct from every other NULL value, so equality comparison is not appropriate.

```
SELECT Fname , Lname
FROM EMPLOYEE
WHERE Super_ssn IS NULL ;
```

Nested Queries, Tuples, and Set/Multiset Comparisons

Some queries require that existing values in the database be fetched and then used in a comparison condition. Such queries can be conveniently formulated by using nested queries, which are complete select-from-where blocks within the WHERE clause of another query. That other query is called the outer query.

```
SELECT DISTINCT Pnumber
FROM PROJECT
WHERE
Pnumber IN
        ( SELECT Pnumber
        FROM PROJECT , DEPARTMENT , EMPLOYEE
        WHERE Dnum = Dnumber ANDMgr_ssn = Ssn AND Lname ='Smith' )
OR
```

```
Pnumber IN
        ( SELECT Pno
        FROM WORKS_ON , EMPLOYEE
        WHERE Essn = Ssn AND Lname ='Smith' );
```

The first nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as manager, while the second nested query selects the project numbers of projects that have an employee with last name 'Smith' involved as worker. In the outer query, we use the OR logical connective to retrieve a PROJECT tuple if the PNUMBER value of that tuple is in the result of either nested query.

If a nested query returns a single attribute and a single tuple, the query result will be a single (scalar) value. In such cases, it is permissible to use = instead of IN for the comparison operator.

SQL allows the use of tuples of values in comparisons by placing them within parentheses. To illustrate this, consider the following query:

```
SELECT DISTINCT Essn
FROM WORKS_ON
WHERE ( Pno , Hours ) IN
        ( SELECT Pno , Hours
        FROM WORKS_ON
        WHERE Essn ='123456789' );
```

This query will select the Essns of all employees who work the same (project, hours) combination on some project that employee 'John Smith' (whose Ssn = '123456789') works on. In addition to the IN operator, a number of other comparison operators can be used to compare a single value v (typically an attribute name) to a set or multiset v. The = ANY (or = SOME) operator returns TRUE if the value v is equal to some value in the set V and is hence equivalent to IN. The two keywords ANY and SOME have the same effect. Other operators that can be combined with ANY (or SOME) include >, >=, <, <=, and <>. The keyword ALL can also be combined with each of these operators. For example, the comparison condition (v > ALL V) returns TRUE if the value v is greater than all the values in the set (or multiset) V. An example is the following query, which returns the names of employees whose salary is greater than the salary of all the employees in department 5:

```
SELECT Lname , Fname
FROM EMPLOYEE
WHERE Salary > ALL
        ( SELECT Salary
        FROM EMPLOYEE
        WHERE Dno =5 );
```

In general, we can have several levels of nested queries. We can once again be faced with possible ambiguity among attribute names if attributes of the same name exist—one in a relation in the FROM clause of the outer query, and another in a relation in the FROM clause of the nested query. The rule is that a reference to an unqualified attribute refers to the relation declared in the innermost nested query.

- Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

```
SELECT E.Fname , E.Lname
```

```
FROM EMPLOYEE AS E
WHERE E.Ssn IN
        ( SELECT Essn
        FROM DEPENDENT AS D
        WHERE E.Fname = D.Dependent_name AND E.Sex = D.Sex );
```

In the nested query, we must qualify E.Sex because it refers to the Sex attribute of EMPLOYEE from the outer query, and DEPENDENT also has an attribute called Sex. If there were any unqualified references to Sex in the nested query, they would refer to the Sex attribute of DEPENDENT. However, we would not have to qualify the attributes Fname and Ssn of EMPLOYEE if they appeared in the nested query because the DEPENDENT relation does not have attributes called Fname and Ssn, so there is no ambiguity.

Correlated Nested Queries

Whenever a condition in the WHERE clause of a nested query references some attribute of a relation declared in the outer query, the two queries are said to be correlated. We can understand a correlated query better by considering that the nested query is evaluated once for each tuple (or combination of tuples) in the outer query.

In general, a query written with nested select-from-where blocks and using the = or IN comparison operators can always be expressed as a single block query. For example, above query may be written as in:

```
SELECT E.Fname , E.Lname
FROM EMPLOYEE AS E , DEPENDENT AS D
WHERE E.Ssn = D.Essn AND E.Sex = D.Sex
AND E.Fname = D.Dependent_name ;
```

The EXISTS and UNIQUE Functions in SQL

The EXISTS function in SQL is used to check whether the result of a correlated nested query is empty (contains no tuples) or not. The result of EXISTS is a Boolean value TRUE if the nested query result contains at least one tuple, or FALSE if the nested query result contains no tuples.

EXISTS and NOT EXISTS are typically used in conjunction with a correlated nested query. In general, EXISTS (Q) returns TRUE if there is at least one tuple in the result of the nested query Q , and it returns FALSE otherwise. On the other hand, NOT EXISTS (Q) returns TRUE if there are no tuples in the result of nested query Q , and it returns FALSE otherwise. Next, we illustrate the use of NOT EXISTS .

- Retrieve the names of employees who have no dependents.

```
SELECT Fname , Lname
FROM EMPLOYEE
WHERE NOT EXISTS ( SELECT *
        FROM DEPENDENT
        WHERE  Ssn = Essn );
```

In above query, the correlated nested query retrieves all DEPENDENT tuples related to a particular EMPLOYEE tuple. If none exist, the EMPLOYEE tuple is selected because the WHERE clause condition will evaluate to TRUE in this case. We can explain query as follows: For each EMPLOYEE tuple, the correlated nested query selects all DEPENDENT tuples whose Essn value matches the EMPLOYEE Ssn; if the result is empty, no dependents are related to the employee, so we select that EMPLOYEE tuple and retrieve its Fname and Lname.

- List the names of managers who have at least one dependent.

```
SELECT Fname , Lname
FROM EMPLOYEE
WHERE EXISTS ( SELECT *
        FROM  DEPENDENT
        WHERE Ssn = Essn )
AND
        EXISTS ( SELECT *
        FROM DEPARTMENT
        WHERE Ssn = Mgr_ssn );
```

In the query above, we specify two nested correlated queries; the first selects all DEPENDENT tuples related to an EMPLOYEE, and the second selects all DEPARTMENT tuples managed by the EMPLOYEE. If at least one of the first and at least one of the second exists, we select the EMPLOYEE tuple.

```
SELECT  Fname , Lname
FROM EMPLOYEE
WHERE NOT EXISTS
        ( ( SELECT Pnumber
        FROM  PROJECT
        WHERE Dnum =5)
        EXCEPT
        ( SELECT Pno
        FROM WORKS_ON
        WHERE Ssn = Essn ) );
```

In above query, the first subquery (which is not correlated with the outer query) selects all projects controlled by department 5, and the second subquery (which is correlated) selects all projects that the particular employee being considered works on. If the set difference of the first subquery result MINUS (EXCEPT) the second subquery result is empty, it means that the employee works on all the projects of department 5 and is therefore selected.

Explicit Sets and Renaming of Attributes in SQL

We have seen several queries with a nested query in the WHERE clause. It is also possible to use an explicit set of values in the WHERE clause, rather than a nested query. Such a set is enclosed in parentheses in SQL. Retrieve the Social Security numbers of all employees who work on project numbers 1, 2, or 3.

```
SELECT DISTINCT Essn
FROM WORKS_ON
WHERE Pno IN (1, 2, 3);
```

In SQL, it is possible to rename any attribute that appears in the result of a query by adding the qualifier AS followed by the desired new name. Hence, the AS construct can be used to alias both attribute and relation names, and it can be used in both the SELECT and FROM clauses. For example in the query below which is to retrieve the last name of each employee and his or her supervisor, while renaming the resulting attribute names as Employee_name and Supervisor_name. The new names will appear as column headers in the query result.

```
SELECT E.Lname AS Employee_name , S.Lname AS Supervisor_name
FROM EMPLOYEE AS E , EMPLOYEE AS S
WHERE E.Super_ssn = S.Ssn ;
```

Joined Tables in SQL and Outer Joins

The concept of a joined table (or joined relation) was incorporated into SQL to permit users to specify a table resulting from a join operation in the FROM clause of a query. This construct may be easier to comprehend than mixing together all the select and join conditions in the WHERE clause. For example, consider the query below, which retrieves the name and address of every employee who works for the 'Research' department. It may be easier to specify the join of the EMPLOYEE and DEPARTMENT relations first, and then to select the desired tuples and attributes. This can be written in SQL as follows :

```
SELECT Fname, Lname, Address
FROM  ( EMPLOYEE JOIN DEPARTMENT ON Dno = Dnumber )
WHERE  Dname ='Research';
```

The FROM clause contains a single joined table. The attributes of such a table are all the attributes of the first table, EMPLOYEE, followed by all the attributes of the second table, DEPARTMENT. The concept of a joined table also allows the user to specify different types of join, such as NATURAL JOIN and various types of OUTER JOIN. In a NATURAL JOIN on two relations R and S, no join condition is specified; an implicit EQUIJOIN condition for each pair of attributes with the same name from R and S is created. Each such pair of attributes is included only once in the resulting relation.

If the names of the join attributes are not the same in the base relations, it is possible to rename the attributes so that they match, and then to apply NATURAL JOIN. In this case, the AS construct can be used to rename a relation and all its attributes in the FROM clause. This is illustrated in query below, where the DEPARTMENT relation is renamed as DEPT and its attributes are renamed as Dname, Dno (to match the name of the desired join attribute Dno in the EMPLOYEE table), Mssn, and Msdate. The implied join condition for this NATURAL JOIN is EMPLOYEE.Dno = DEPT.Dno, because this is the only pair of attributes with the same name after renaming:

```
SELECT Fname , Lname , Address
FROM ( EMPLOYEE NATURAL JOIN
        ( DEPARTMENT AS DEPT ( Dname , Dno , Mssn , Msdate )))
WHERE Dname ='Research';
```

The default type of join in a joined table is called an inner join, where a tuple is included in the result only if a matching tuple exists in the other relation. If the user requires that all employees be included, an OUTER JOIN must be used explicitly. In SQL, this is handled by explicitly specifying the keyword OUTER JOIN in a joined table, as illustrated below:

```
SELECT E.Lname AS Employee_name, S.Lname AS Supervisor_name
FROM ( EMPLOYEE AS E LEFT OUTER JOIN
       EMPLOYEE AS S ON E.Super_ssn = S.Ssn);
```

In SQL, the options available for specifying joined tables include INNER JOIN (only pairs of tuples that match the join condition are retrieved, same as JOIN), LEFT OUTER JOIN (every tuple in the left table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the right table), RIGHT OUTER

JOIN (every tuple in the right table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the left table), and FULL OUTER JOIN. In the latter three options, the keyword OUTER may be omitted. If the join attributes have the same name, one can also specify the natural join variation of outer joins by using the keyword NATURAL before the operation (for example, NATURAL LEFT OUTER JOIN). The keyword CROSS JOIN is used to specify the CARTESIAN PRODUCT operation. It is also possible to nest join specifications; that is, one of the tables in a join may itself be a joined table.

```
SELECT Pnumber, Dnum, Lname, Address, Bdate
FROM (( PROJECT JOIN DEPARTMENT ON Dnum = Dnumber ) JOIN
        EMPLOYEE ON Mgr_ssn = Ssn )
WHERE Plocation ='Stafford';
```

Not all SQL implementations have implemented the new syntax of joined tables. In some systems, a different syntax was used to specify outer joins by using the comparison operators +=, =+, and +=+ for left, right, and full outer join, respectively, when specifying the join condition. For example, to specify the left outer join using this syntax, we could write the query as follows:

```
SELECT E.Lname , S.Lname
FROM EMPLOYEE E , EMPLOYEE S
WHERE E.Super_ssn + = S.Ssn ;
```

Aggregate Functions in SQL

Aggregate functions are used to summarize information from multiple tuples into a single tuple summary. Grouping is used to create sub-groups of tuples before summarization. A number of built-in aggregate functions exist: COUNT , SUM , MAX , MIN , and AVG. The COUNT function returns the number of tuples or values as specified in a query. The functions SUM, MAX, MIN, and AVG can be applied to a set or multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values. These functions can be used in the SELECT clause or in a HAVING clause.

- Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.
  SELECT SUM(Salary), MAX(Salary), MIN (Salary), AVG (Salary) FROM EMPLOYEE ;

If we want to get the preceding function values for employees of a specific department —say, the 'Research' department—we can write query, where the EMPLOYEE tuples are restricted by the WHERE clause to those employees who work for the 'Research' department.

- Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.
  SELECT SUM(Salary), MAX(Salary), MIN(Salary), AVG (Salary) FROM (EMPLOYEE JOIN DEPARTMENT ON Dno = Dnumber) WHERE Dname ='Research';

- Retrieve the total number of employees in the company.
  SELECT COUNT (*)
  FROM EMPLOYEE ;

- Retrieve the number of employees in the 'Research' department.
  SELECT COUNT (*)
  FROM EMPLOYEE, DEPARTMENT
  WHERE DNO = DNUMBER AND DNAME ='Research';

Here the asterisk (*) refers to the rows (tuples), so COUNT (*) returns the number of rows in the result of the query. We may also use the COUNT function to count values in a column rather than tuples, as in the next example.

```
SELECT COUNT ( DISTINCT Salary )
FROM EMPLOYEE ;
```

If we write COUNT(SALARY) instead of COUNT(DISTINCT SALARY), then duplicate values will not be eliminated. However, any tuples with NULL for SALARY will not be counted. In general, NULL values are discarded when aggregate functions are applied to a particular column(attribute).

We can specify a correlated nested query with an aggregate function, and then use the nested query in the WHERE clause of an outer query. For example, to retrieve the names of all employees who have two or more dependents, we can write the following:

```
SELECT Lname , Fname
FROM EMPLOYEE
WHERE ( SELECT COUNT ( * )
        FROM DEPENDENT
        WHERE
        Ssn = Essn ) >= 2;
```

The correlated nested query counts the number of dependents that each employee has; if this is greater than or equal to two, the employee tuple is selected.

Grouping: The GROUP BY and HAVING Clauses

In many cases we want to apply the aggregate functions to subgroups of tuples in a relation, where the subgroups are based on some attribute values. For example, we may want to find the average salary of employees in each department or the number of employees who work on each project. In these cases we need to partition the relation into nonoverlapping subsets (or groups) of tuples. Each group (partition) will consist of the tuples that have the same value of some attribute(s), called the grouping attribute(s). We can then apply the function to each such group independently to produce summary information about each group. SQL has a GROUP BY clause for this purpose. The GROUP BY clause specifies the grouping attributes, which should also appear in the SELECT clause, so that the value resulting from applying each aggregate function to a group of tuples appears along with the value of the grouping attribute(s).

- For each department, retrieve the department number, the number of employees in the department, and their average salary.
  SELECT Dno, COUNT (*), AVG (Salary)
  FROM EMPLOYEE GROUP BY Dno ;

In the above query, the EMPLOYEE tuples are partitioned into groups—each group having the same value for the grouping attribute Dno. Hence, each group contains the employees who work in the same department. The COUNT and AVG functions are applied to each such group of tuples. Notice that the SELECT clause includes only the grouping

attribute and the aggregate functions to be applied on each group of tuples. Figure below illustrates how grouping works on the above query.

Fname	Minit	Lname	Ssn	...	Salary	Super_ssn	Dno
John	B	Smith	123456789		30000	333445555	5
Franklin	T	Wong	333445555		40000	888665555	5
Ramesh	K	Narayan	666884444		38000	333445555	5
Joyce	A	English	453453453	...	25000	333445555	5
Alicia	J	Zelaya	999887777		25000	987654321	4
Jennifer	S	Wallace	987654321		43000	888665555	4
Ahmad	V	Jabbar	987987987		25000	987654321	4
James	E	Bong	888665555		55000	NULL	1

Dno	Count (*)	Avg (Salary)
5	4	33250
4	3	31000
1	1	55000

Grouping EMPLOYEE tuples by the value of Dno

If NULL s exist in the grouping attribute, then a separate group is created for all tuples with a NULL value in the grouping attribute. For example, if the EMPLOYEE table had some tuples that had NULL for the grouping attribute Dno , there would be a separate group for those tuples in the result.

- For each project, retrieve the project number, the project name, and the number of employees who work on that project.

```
SELECT  Pnumber, Pname, COUNT (*)
FROM PROJECT, WORKS_ON
WHERE Pnumber = Pno
GROUP BY Pnumber, Pname ;
```

Pname	Pnumber	...	Essn	Pno	Hours
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
Computerization	10		333445555	10	10.0
Computerization	10	...	999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

Pname	Count (*)
ProductY	3
Computerization	3
Reorganization	3
Newbenefits	3

After applying the HAVING clause condition

Above query shows how we can use a join condition in conjunction with GROUP BY. In this case, the grouping and functions are applied after the joining of the two relations. Sometimes we want to retrieve the values of these functions only for groups that satisfy certain conditions. For example, suppose that we want to modify the query so that only projects with more than two employees appear in the result. SQL provides a HAVING clause, which can appear in conjunction with a GROUP BY clause, for this purpose. HAVING

provides a condition on the summary information regarding the group of tuples associated with each value of the grouping attributes. Only the groups that satisfy the condition are retrieved in the result of the query.

- For each project on which more than two employees work, retrieve the project number, the project name, and the number of employees who work on the project.

```
SELECT Pnumber , Pname , COUNT ( * )
FROM PROJECT , WORKS_ON
WHERE Pnumber = Pno
GROUP BY Pnumber , Pname
HAVING COUNT ( * ) > 2 ;
```

Notice that while selection conditions in the WHERE clause limit the tuples to which functions are applied, the HAVING clause serves to choose whole groups.

- For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project:

```
SELECT Pnumber, Pname, COUNT ( * )
FROM PROJECT, WORKS_ON, EMPLOYEE
WHERE Pnumber = Pno AND Ssn = Essn AND Dno = 5
GROUP BY Pnumber , Pname ;
```

Here we restrict the tuples in the relation (and hence the tuples in each group) to those that satisfy the condition specified in the WHERE clause—namely, that they work in department number 5. Notice that we must be extra careful when two different conditions apply (one to the aggregate function in the SELECT clause and another to the function in the HAVING clause). For example, suppose that we want to count the total number of employees whose salaries exceed $40,000 in each department, but only for departments where more than five employees work. Here, the condition (SALARY > 40000) applies only to the COUNT function in the SELECT clause. Suppose that we write the following incorrect query:

```
SELECT Dname , COUNT ( * )
FROM DEPARTMENT , EMPLOYEE
WHERE Dnumber = Dno AND Salary > 40000
GROUP BY Dname
HAVING COUNT ( * ) > 5 ;
```

This is incorrect because it will select only departments that have more than five employees who each earn more than $40,000. The rule is that the WHERE clause is executed first, to select individual tuples or joined tuples; the HAVING clause is applied later, to select individual groups of tuples. Hence, the tuples are already restricted to employees who earn more than $40,000 before the function in the HAVING clause is applied. One way to write this query correctly is to use a nested query, as shown below.

```
SELECT Dnumber , COUNT ( * )
FROM DEPARTMENT, EMPLOYEE
WHERE Dnumber = Dno AND Salary > 40000 AND Dnumber
        ( SELECT Dno
        FROM EMPLOYEE
        GROUP BY Dno
```

```
HAVING COUNT ( * ) > 5)
group by Dnumber;
```

Discussion and Summary of SQL Queries

A retrieval query in SQL can consist of up to six clauses, but only the first two— SELECT and FROM —are mandatory. The query can span several lines, and is ended by a semicolon. Query terms are separated by spaces, and parentheses can be used to group relevant parts of a query in the standard way. The clauses are specified in the following order, with the clauses between square brackets [...] being optional:

```
SELECT <attribute and function list>
FROM <table list>
[ WHERE <condition> ]
[ GROUP BY <grouping attribute(s)> ]
[ HAVING <group condition> ]
[ ORDER BY <attribute list> ] ;
```

The SELECT clause lists the attributes or functions to be retrieved. The FROM clause specifies all relations (tables) needed in the query, including joined relations, but not those in nested queries. The WHERE clause specifies the conditions for selecting the tuples from these relations, including join conditions if needed. GROUP BY specifies grouping attributes, whereas HAVING specifies a condition on the groups being selected rather than on the individual tuples. The built-in aggregate functions COUNT, SUM, MIN, MAX, and AVG are used in conjunction with grouping, but they can also be applied to all the selected tuples in a query without a GROUP BY clause. Finally, ORDER BY specifies an order for displaying the result of a query.

In order to formulate queries correctly, it is useful to consider the steps that define the meaning or semantics of each query. A query is evaluated conceptually by first applying the FROM clause (to identify all tables involved in the query or to materialize any joined tables), followed by the WHERE clause to select and join tuples, and then by GROUP BY and HAVING. Conceptually, ORDER BY is applied at the end to sort the query result. If none of the last three clauses (GROUP BY, HAVING, and ORDER BY) are specified, we can think conceptually of a query as being executed as follows: For each combination of tuples—one from each of the relations specified in the FROM clause—evaluate the WHERE clause; if it evaluates to TRUE , place the values of the attributes specified in the SELECT clause from this tuple combination in the result of the query.

Specifying Constraints as Assertions and Actions as Triggers

In this section, we introduce two additional features of SQL: the CREATE ASSERTION statement and the CREATE TRIGGER statement. CREATE ASSERTION, can be used to specify additional types of constraints that are outside the scope of the built-in relational model constraints (primary and unique keys, entity integrity, and referential integrity). These built-in constraints can be specified within the CREATE TABLE statement of SQL. Then we introduce CREATE TRIGGER, which can be used to specify automatic actions that the database system will perform when certain events and conditions occur. This type of functionality is generally referred to as active databases.

Specifying General Constraints as Assertions in SQL

In SQL, users can specify general constraints—those that do not fall into any of the categories—via declarative assertions, using the CREATE ASSERTION statement of the DDL. Each assertion is given a constraint name and is specified via a condition similar to

the WHERE clause of an SQL query. For example, to specify the constraint that the salary of an employee must not be greater than the salary of the manager of the department that the employee works for in SQL, we can write the following assertion:

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS ( SELECT  *
FROM EMPLOYEE E, EMPLOYEE M, DEPARTMENT D
WHERE E.Salary > M.Salary AND E.Dno = D.Dnumber AND D.Mgr_ssn = M.Ssn ) );
```

The constraint name SALARY_CONSTRAINT is followed by the keyword CHECK, which is followed by a condition in parentheses that must hold true on every database state for the assertion to be satisfied. The constraint name can be used later to refer to the constraint or to modify or drop it. The DBMS is responsible for ensuring that the condition is not violated. Any WHERE clause condition can be used, but many constraints can be specified using the EXISTS and NOT EXISTS style of SQL conditions. Whenever some tuples in the database cause the condition of an ASSERTION statement to evaluate to FALSE, the constraint is violated. The constraint is satisfied by a database state if no combination of tuples in that database state violates the constraint.

The basic technique for writing such assertions is to specify a query that selects any tuples that violate the desired condition. By including this query inside a NOT EXISTS clause, the assertion will specify that the result of this query must be empty so that the condition will always be TRUE. Thus, the assertion is violated if the result of the query is not empty. In the preceding example, the query selects all employees whose salaries are greater than the salary of the manager of their department. If the result of the query is not empty, the assertion is violated.

Introduction to Triggers in SQL

Another important statement in SQL is CREATE TRIGGER. In many cases it is convenient to specify the type of action to be taken when certain events occur and when certain conditions are satisfied. For example, it may be useful to specify a condition that, if violated, causes some user to be informed of the violation. A manager may want to be informed if an employee's travel expenses exceed a certain limit by receiving a message whenever this occurs. The action that the DBMS must take in this case is to send an appropriate message to that user. The condition is thus used to monitor the database. Other actions may be specified, such as executing a specific stored procedure or triggering other updates. The CREATE TRIGGER statement is used to implement such actions in SQL.

Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database. Several events can trigger this rule: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. Suppose that the action to take would be to call an external stored procedure, which will notify the supervisor. The trigger could then be written as below.

```
CREATE TRIGGER SALARY_VIOLATION
BEFORE INSERT OR UPDATE OF SALARY , SUPERVISOR_SSN
ON EMPLOYEE FOR EACH ROW
WHEN ( NEW.SALARY > ( SELECT SALARY FROM EMPLOYEE
WHERE SSN = NEW.SUPERVISOR_SSN ) )
INFORM_SUPERVISOR (NEW.Supervisor_ssn, NEW.Ssn);
```

The trigger is given the name SALARY_VIOLATION, which can be used to remove or deactivate the trigger later. A typical trigger has three components:

1. The event(s): These are usually database update operations that are explicitly applied to the database. In this example the events are: inserting a new employee record,

changing an employee's salary, or changing an employee's supervisor. These events are specified after the keyword BEFORE in our example, which means that the trigger should be executed before the triggering operation is executed. An alternative is to use the keyword AFTER, which specifies that the trigger should be executed after the operation specified in the event is completed.

2. The condition that determines whether the rule action should be executed: Once the triggering event has occurred, an optional condition may be evaluated. If no condition is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only if it evaluates to true will the rule action be executed. The condition is specified in the WHEN clause of the trigger.
3. The action to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed. In this example, the action is to execute the stored procedure INFORM_SUPERVISOR.

Views (Virtual Tables) in SQL

In this section we introduce the concept of a view in SQL. We show how views are specified, and then we discuss the problem of updating views and how views can be implemented by the DBMS.

Concept of a View in SQL

A view in SQL terminology is a single table that is derived from other tables. These other tables can be base tables or previously defined views. A view does not necessarily exist in physical form; it is considered to be a virtual table, in contrast to base tables, whose tuples are always physically stored in the database. This limits the possible update operations that can be applied to views, but it does not provide any limitations on querying a view. We can think of a view as a way of specifying a table that we need to reference frequently, even though it may not exist physically.

For example, referring to the COMPANY database we may frequently issue queries that retrieve the employee name and the project names that the employee works on. Rather than having to specify the join of the three tables EMPLOYEE, WORKS_ON, and PROJECT every time we issue this query, we can define a view that is specified as the result of these joins. Then we can issue queries on the view, which are specified as single- table retrievals rather than as retrievals involving two joins on three tables. We call the EMPLOYEE, WORKS_ON, and PROJECT tables the defining tables of the view.

Specification of Views in SQL

In SQL, the command to specify a view is CREATE VIEW. The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view. If none of the view attributes results from applying functions or arithmetic operations, we do not have to specify new attribute names for the view, since they would be the same as the names of the attributes of the defining tables in the default case.

```
CREATE VIEW  WORKS_ON1
AS SELECT Fname, Lname, Pname, Hours
FROM EMPLOYEE , PROJECT , WORKS_ON
WHERE Ssn = Essn AND Pno = Pnumber;

CREATE VIEW DEPT_INFO(Dept_name,No_of_emps, Total_sal )
AS SELECT Dname , COUNT ( * ), SUM ( Salary )
FROM DEPARTMENT , EMPLOYEE
```

```
WHERE Dnumber = Dno
GROUP BY Dname ;
```

We did not specify any new attribute names for the view WORKS_ON1 in this case, WORKS_ON1 inherits the names of the view attributes from the defining tables EMPLOYEE, PROJECT, and WORKS_ON. View DEPT_INFO explicitly specifies new attribute names, using a one-to-one correspondence between the attributes specified in the CREATE VIEW clause and those specified in the SELECT clause of the query that defines the view.

WORKS_ON1

Fname	Lname	Pname	Hours

DEPT_INFO

Dept_name	No_of_emps	Total_sal

We can now specify SQL queries on a view—or virtual table—in the same way we specify queries involving base tables. For example, to retrieve the last name and first name of all employees who work on the 'ProductX' project, we can utilize the WORKS_ON1 view and specify the query as follows:

1.
```
SELECT Fname, Lname
FROM WORKS_ON1
WHERE Pname ='ProductX';
```

The same query would require the specification of two joins if specified on the base relations directly; one of the main advantages of a view is to simplify the specification of certain queries.

A view is supposed to be always up-to-date; if we modify the tuples in the base tables on which the view is defined, the view must automatically reflect these changes. Hence, the view is not realized or materialized at the time of view definition but rather at the time when we specify a query on the view. It is the responsibility of the DBMS and not the user to make sure that the view is kept up-to-date. If we do not need a view any more, we can use the DROP VIEW command to dispose of it. For example, to get rid of the view WORKS_ON1, we can use the SQL statement:

```
DROP VIEW WORKS_ON1 ;
```

View Implementation, View Update, and Inline Views

The problem of efficiently implementing a view for querying is complex. Two main approaches have been suggested. One strategy, called **query modification**, involves modifying or transforming the view query (submitted by the user) into a query on the underlying base tables. For example, the query QV1 would be automatically modified to the following query by the DBMS:

```
SELECT Fname, Lname
FROM EMPLOYEE, PROJECT, WORKS_ON
WHERE Ssn = Essn AND Pno = Pnumber
AND Pname ='ProductX';
```

The second strategy, called **view materialization**, involves physically creating a temporary view table when the view is first queried and keeping that table on the assumption that other queries on the view will follow. In this case, an efficient strategy for

automatically updating the view table when the base tables are updated must be developed in order to keep the view up-to-date. Techniques using the concept of incremental update have been developed for this purpose, where the DBMS can determine what new tuples must be inserted, deleted, or modified in a materialized view table when a database update is applied to one of the defining base tables.

Updating of views is complicated and can be ambiguous. In general, an update on a view defined on a single table without any aggregate functions can be mapped to an update on the underlying base table under certain conditions. For a view involving joins, an update operation may be mapped to update operations on the underlying base relations in multiple ways. Hence, it is often not possible for the DBMS to determine which of the updates is intended. To illustrate potential problems with updating a view defined on multiple tables, consider the WORKS_ON1 view, and suppose that we issue the command to update the PNAME attribute of 'John Smith' from 'ProductX' to 'ProductY'. This view update is shown below:

```
UPDATE WORKS_ON1
SET Pname = 'ProductY'
WHERE Lname ='Smith' AND Fname ='John'
AND Pname ='ProductX';
```

This query can be mapped into several updates on the base relations to give the desired update effect on the view. In addition, some of these updates will create additional side effects that affect the result of other queries. For example, here are two possible updates, (a) and (b), on the base relations corresponding to the view

a)
```
UPDATE WORKS_ON
SET Pno = ( SELECT Pnumber
            FROM PROJECT
            WHERE Pname ='ProductY' )
WHERE Essn IN ( SELECT Ssn
                FROM EMPLOYEE
                WHERE Lname ='Smith' AND Fname ='John' )
              AND
                Pno = ( SELECT Pnumber
                FROM PROJECT
                WHERE Pname ='ProductX' );
```

b)
```
UPDATE PROJECT SET Pname = 'ProductY'
WHERE Pname = 'ProductX';
```

Update (a) relates 'John Smith' to the 'ProductY' PROJECT tuple instead of the 'ProductX' PROJECT tuple and is the most likely desired update. However, (b) would also give the desired update effect on the view, but it accomplishes this by changing the name of the 'ProductX' tuple in the PROJECT relation to 'ProductY'. It is quite unlikely that the user who specified the view update wants the update to be interpreted as in (b), since it also has the side effect of changing all the view tuples with Pname = 'ProductX'.

Some view updates may not make much sense; for example, modifying the Total_sal attribute of the DEPT_INFO view does not make sense because Total_sal is defined to be the sum of the individual employee salaries. This request is shown below :

```
UPDATE DEPT_INFO
SET Total_sal = 100000
```

WHERE Dname ='Research';

Generally, a view update is feasible when only one possible update on the base relations can accomplish the desired update effect on the view. Whenever an update on the view can be mapped to more than one update on the underlying base relations, we must have a certain procedure for choosing one of the possible updates as the most likely one.

In summary, we can make the following observations:

- A view with a single defining table is updatable if the view attributes contain the primary key of the base relation, as well as all attributes with the NOT NULL constraint that do not have default values specified.
- Views defined on multiple tables using joins are generally not updatable.
- Views defined using grouping and aggregate functions are not updatable.

In SQL, the clause WITH CHECK OPTION must be added at the end of the view definition if a view is to be updated. It is also possible to define a view table in the FROM clause of an SQL query. This is known as an **in-line view**.

Schema Change Statements in SQL

In this section, we give an overview of the schema evolution commands available in SQL, which can be used to alter a schema by adding or dropping tables, attributes, constraints, and other schema elements. This can be done while the database is operational and does not require recompilation of the database schema.

The DROP Command

The DROP command can be used to drop named schema elements, such as tables, domains, or constraints. One can also drop a schema. For example, if a whole schema is no longer needed, the DROP SCHEMA command can be used. There are two drop behavior options: CASCADE and RESTRICT. For example, to remove the COMPANY database schema and all its tables, domains, and other elements, the CASCADE option is used as follows:

DROP SCHEMA COMPANY CASCADE ;

If the RESTRICT option is chosen in place of CASCADE, the schema is dropped only if it has no elements in it; otherwise, the DROP command will not be executed.

To use the RESTRICT option, the user must first individually drop each element in the schema, then drop the schema itself. If a base relation within a schema is no longer needed, the relation and its definition can be deleted by using the DROP TABLE command. For example, if we no longer wish to keep track of dependents of employees in the COMPANY database of, we can get rid of the DEPENDENT relation by issuing the following command:

DROP TABLE DEPENDENT CASCADE ;

If the RESTRICT option is chosen instead of CASCADE, a table is dropped only if it is not referenced in any constraints (for example, by foreign key definitions in another relation) or views or by any other elements. With the CASCADE option, all such constraints, views, and other elements that reference the table being dropped are also dropped automatically from the schema, along with the table itself.

Notice that the DROP TABLE command not only deletes all the records in the table if successful, but also removes the table definition from the catalog. If it is desired to delete only the records but to leave the table definition for future use, then the DELETE command should be used instead of DROP TABLE. The DROP command can also be used to drop other types of named schema elements, such as constraints or domains.

The ALTER Command

The definition of a base table or of other named schema elements can be changed by using the ALTER command. For base tables, the possible alter table actions include adding or dropping a column (attribute), changing a column definition, and adding or dropping table constraints. For example, to add an attribute for keeping track of jobs of employees to the EMPLOYEE base relation in the COMPANY schema, we can use the command :

ALTER TABLE COMPANY. EMPLOYEE ADD COLUMN Job VARCHAR (12);

We must still enter a value for the new attribute Job for each individual EMPLOYEE tuple. This can be done either by specifying a default clause or by using the UPDATE command individually on each tuple. If no default clause is specified, the new attribute will have NULLs in all the tuples of the relation immediately after the command is executed; hence, the NOT NULL constraint is not allowed in this case.

To drop a column, we must choose either CASCADE or RESTRICT for drop behavior. If CASCADE is chosen, all constraints and views that reference the column are dropped automatically from the schema, along with the column. If RESTRICT is chosen, the command is successful only if no views or constraints reference the column. For example, the following command removes the attribute Address from the EMPLOYEE base table:

ALTER TABLE COMPANY. EMPLOYEE DROP COLUMN Address CASCADE ;

It is also possible to alter a column definition by dropping an existing default clause or by defining a new default clause. The following examples illustrate this clause:

ALTER TABLE COMPANY. DEPARTMENT ALTER COLUMN Mgr_ssn DROP DEFAULT ;

ALTER TABLE COMPANY. DEPARTMENT ALTER COLUMN Mgr_ssn SET DEFAULT '333445555';

One can also change the constraints specified on a table by adding or dropping a named constraint. To be dropped, a constraint must have been given a name when it was specified. For example, to drop the constraint named EMPSUPERFK from the EMPLOYEE relation, we write:

ALTER TABLE COMPANY. EMPLOYEE DROP CONSTRAINT EMPSUPERFK CASCADE;

Once this is done, we can redefine a replacement constraint by adding a new constraint to the relation, if needed. This is specified by using the ADD keyword in the ALTER TABLE statement followed by the new constraint, which can be named or unnamed and can be of any of the table constraint types discussed.

STORED PROCEDURES

When SQL statements are issued from a remote application, the records in the result of the query need to be transferred from the database system back to the application. If we use a cursor to remotely access the results of an SQL statement, the DBMS has resources such as locks and memory tied up while the application is processing the records retrieved through the cursor. In contrast, a stored procedure is a program that is executed through a single SQL statement that can be locally executed and completed within the process space of the database server. The results can be packaged into one big result and returned to the application, or the application logic can be performed directly at the server, without having to transmit the results to the client at all. Stored procedures are also beneficial for software engineering reasons. Once a stored procedure is registered with the database server, different users can re-use the stored procedure, eliminating duplication of efforts in writing SQL queries. In addition, application programmers do not need to know the the database schema if we encapsulate all database access into stored procedures.

Creating a Simple Stored Procedure

Let us look at the example stored procedure written in SQL shown below; this stored procedure has the name 'ShowNumberOfOrders'.

```
CREATE PROCEDURE ShowNumberOfOrders
SELECT C.cid, C.cname, COUNT(*)
FROM Customers C, Orders O
WHERE C.cid = O.cid
GROUP BY C.cid, C.cname
```

Stored procedures can also have parameters. These parameters have to be valid SQL types, and have one of three different modes: IN, OUT, or INOUT. IN parameters are arguments to' the stored procedure. OUT parameters are returned from the stored procedure; it assigns values to all OUT parameters that the user can process. INOUT parameters combine the properties of IN and OUT parameters: They contain values to be passed to the stored procedures, and the stored procedure can set their values as return values.

Let us look at an example of a stored procedure with arguments. The stored procedure shown in Figure 6.9 has two arguments: book_isbn and addedQty. It updates the available number of copies of a book with the quantity from a new shipment.

```
CREATE PROCEDURE AddInventory (
IN book_isbn CHAR(10),
IN addedQty INTEGER)
UPDATE Books
SET qty_in_stock = qty_jn_stock + addedQty
WHERE book_isbn = isbn
```

Stored procedures do not have to be written in SQL; they can be written in any host language.

Calling Stored Procedures

Stored procedures can be called in interactive SQL with the CALL statement:

CALL storedProcedureName(argument1, argument2, ... , argumentN);

In Embedded SQL, the arguments to a stored procedure are usually variables in the host language. For example, the stored procedure AddInventory would be called as follows:

```
EXEC SQL BEGIN DECLARE SECTION
        char isbn[10];
        long qty;
EXEC SQL END DECLARE SECTION
// set isbn and qty to some values
EXEC SQL CALL AddInventory(:isbn,:qty);
```

Calling Stored Procedures from JDBC

We can call stored procedures from JDBC using the CallableStatment class. CallableStatement is a subclass of PreparedStatement and provides the same functionality.

```
CallableStatement cstmt = con. prepareCall(" {call ShowNumberOfOrders}");
ResultSet rs = cstmt.executeQueryO
```

Calling Stored Procedures from SQLJ

The stored procedure 'ShowNumberOfOrders' is called as follows using SQLJ:

```
// create the cursor class
#sql Iterator CustomerInfo(int cid, String cname, int count);
```

```
// create the cursor
CustomerInfo customerinfo;
// call the stored procedure
#sql customerinfo = {CALL ShowNumberOfOrders};
while (customerinfo.next( )) {
        System.out.println(customerinfo.cid() + "," +  customerinfo.count()) ;
}
```

SQL/PSM

In this section, we briefly discuss the SQL/PSM standard, which is representative of most vendor specific languages. In PSM, we define modules, which are collections of stored procedures, temporary relations, and other declarations. In SQL/PSM, we declare a stored procedure as follows:

```
CREATE PROCEDURE name (parameter1, ..., parameterN)
local variable declarations
procedure code;
```

We can declare a function similarly as follows:

```
CREATE FUNCTION name (parameter1, ..., parameterN)
RETURNS sqlDataType
local variable declarations
function code;
```

We start out with an example of a SQL/PSM function that illustrates the main SQL/PSM constructs. The function takes as input a customer identified by her cid and a year. The function returns the rating of the customer, which is defined as follows: Customers who have bought more than ten books during the year are rated 'two'; customer who have purchased between 5 and 10 books are rated 'one', otherwise the customer is rated 'zero'. The following SQL/PSM code computes the rating for a given customer and year.

```
CREATE PROCEDURE RateCustomer
        (IN custId INTEGER, IN year INTEGER)
        RETURNS INTEGER
DECLARE rating INTEGER;
DECLARE numOrders INTEGER;
SET numOrders = (SELECT COUNT(*) FROM Orders O WHERE O.tid = custId);
IF (numOrders> 10) THEN rating=2;
ELSEIF (numOrders>5) THEN rating=1;
ELSE rating=0;
END IF;
RETURN rating;
```

Let us use this example to give a short overview of some SQL/PSM constructs:

- We can declare local variables using the DECLARE statement. In our example, we declare two local variables: 'rating', and 'numOrders'.
- PSM/SQL functions return values via the RETURN statement. In our example, we return the value of the local variable 'rating'.
- We can assign values to variables with the SET statement. In our example, we assigned the return value of a query to the variable 'numOrders'.

SQL/PSM has branches and loops. Branches have the following form:

```
IF (condition) THEN statements;
```

ELSEIF statements;
. . .
ELSEIF statements;
ELSE statements;
END IF

Loops are of the form
LOOP
statements:
END LOOP

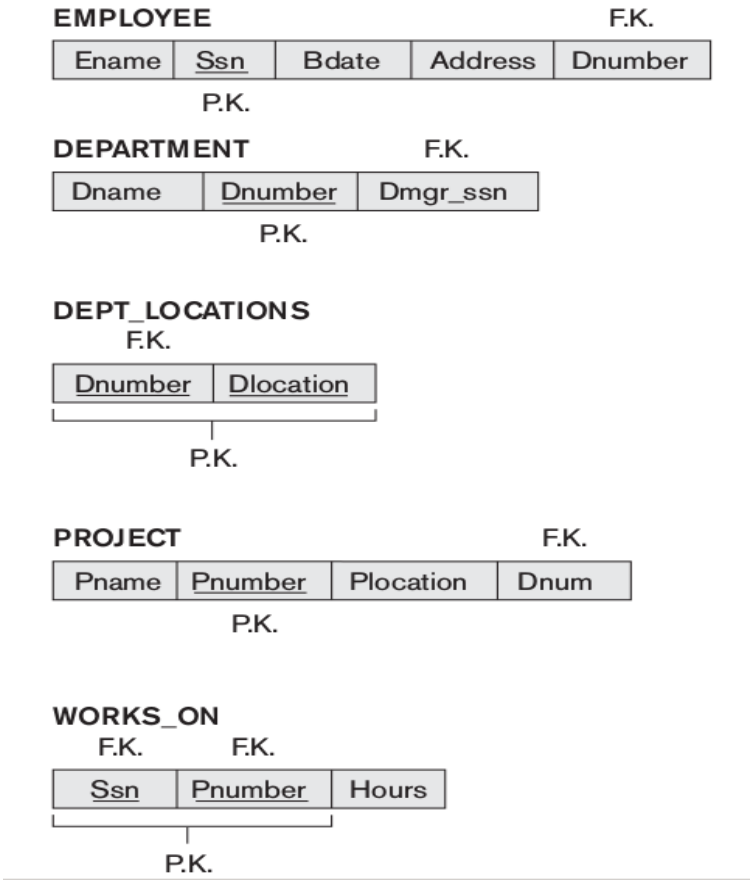
Informal Design Guidelines for Relation Schemas

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.



In general, the easier it is to explain the semantics of the relation, the better the relation schema design will be. To illustrate this, consider Figure above, a simplified version

of the COMPANY relational database schema. The meaning of the EMPLOYEE relation schema is quite simple: Each tuple represents an employee, with values for the employee's name (Ename), Social Security number (Ssn), birth date (Bdate), and address (Address), and the number of the department that the employee works for (Dnumber). The Dnumber attribute is a foreign key that represents an implicit relationship between EMPLOYEE and DEPARTMENT.

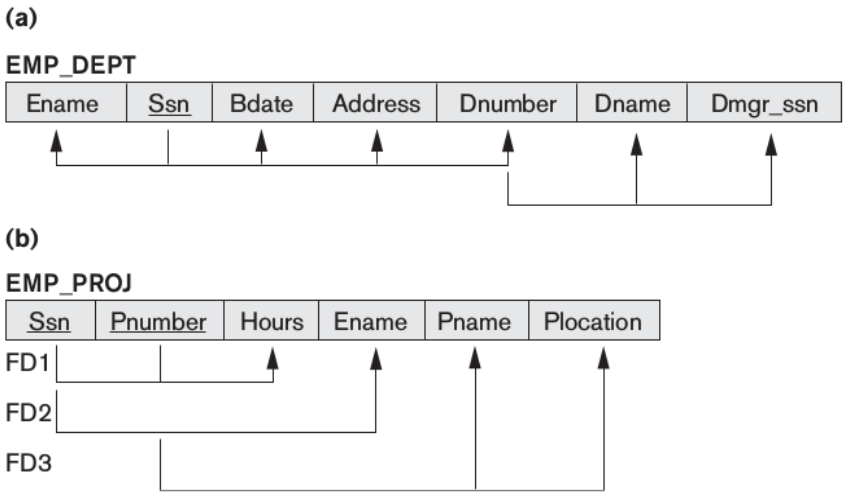
The semantics of the DEPARTMENT and PROJECT schemas are also straightforward: Each DEPARTMENT tuple represents a department entity, and each PROJECT tuple represents a project entity. The attribute Dmgr_ssn of DEPARTMENT relates a department to the employee who is its manager, while Dnum of PROJECT relates a project to its controlling department; both are foreign key attributes. The ease with which the meaning of a relation's attributes can be explained is an informal measure of how well the relation is designed.

The semantics of the other two relation schemas in Figure are slightly more complex. Each tuple in DEPT_LOCATIONS gives a department number(Dnumber) and one of the locations of the department (Dlocation). Each tuple in WORKS_ON gives an employee Social Security number (Ssn), the project number of one of the projects that the employee works on (Pnumber), and the number of hours per week that the employee works on that project (Hours).

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. Intuitively, if a relation schema corresponds to one entity type or one relationship type, it is straightforward to interpret and to explain its meaning. Otherwise, if the relation corresponds to a mixture of multiple entities and relationships, semantic ambiguities will result and the relation cannot be easily explained.

Examples of Violating Guideline 1. The relation schemas in Figures below also have clear semantics. A tuple in the EMP_DEPT relation schema represents a single employee but includes additional information—namely, the name (Dname) of the department for which the employee works and the Social Security number (Dmgr_ssn) of the department manager. For the EMP_PROJ relation, each tuple relates an employee to a project but also includes the employee name (Ename), project name (Pname), and project location (Plocation). Although there is nothing wrong logically with these two relations, they violate Guideline 1 by mixing attributes from distinct real-world entities: EMP_DEPT mixes attributes of employees and departments, and EMP_PROJ mixes attributes of employees and projects and the WORKS_ON relationship.



Redundant Information in Tuples and Update Anomalies

One goal of schema design is to minimize the storage space used by the base relations. 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 with that for an EMP_DEPT base relation, 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.
  - It is difficult to insert a new department that has no employees as yet in the EMP_DEPT relation. The only way to do this is to place NULL values in the attributes for employee. This violates the entity integrity for EMP_DEPT because Ssn is its primary key. Moreover, when the first employee is assigned to that department, we do not need this tuple with NULL values any more.
- Deletion Anomalies. 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.
- 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. The second guideline is consistent with and, in a way, a restatement of the first guideline.

NULL Values in Tuples

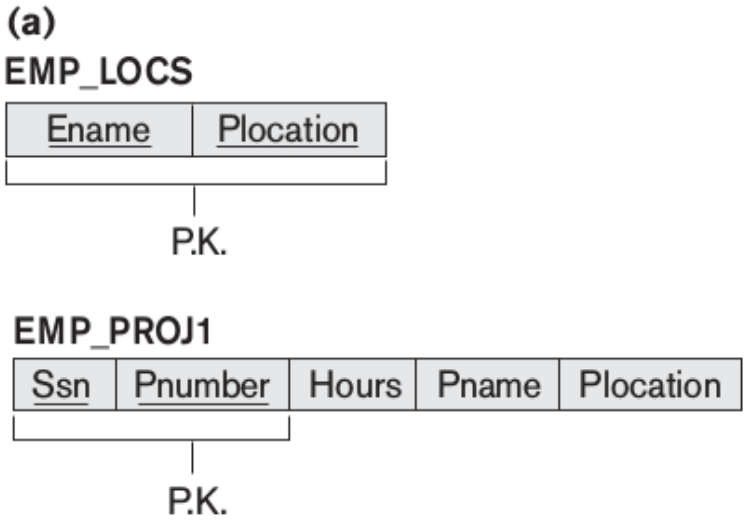
If many of the attributes do not apply to all tuples in the relation, we end up with many NULLs in those tuples. This can waste space at the storage level and may also lead to problems with understanding the meaning of the attributes and with specifying JOIN operations at the logical level. Another problem 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. Having the same representation for all NULLs compromises the different meanings they may have. Therefore, we may state another guideline.

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. For example, if only 15 percent of employees have individual offices, there is little justification for including an attribute Office_number in the EMPLOYEE relation; rather, a relation EMP_OFFICES(Essn, Office_number) can be created to include tuples for only the employees with individual offices.

Generation of Spurious Tuples

Consider the two relation schemas EMP_LOCS and EMP_PROJ1 in Figure below, which can be used instead of the single EMP_PROJ relation. 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.

Additional tuples that were not in EMP_PROJ are called spurious tuples because they represent spurious information that is not valid. Decomposing EMP_PROJ into EMP_LOCS and EMP_PROJ1 is undesirable because when we JOIN them back using NATURAL JOIN, we do not get the correct original information. This is because in this case Plocation is the attribute that relates EMP_LOCS and EMP_PROJ1, and Plocation is neither a primary key nor a foreign key in either EMP_LOCS or EMP_PROJ1.

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. Avoid relations that contain matching attributes that are not (foreign key, primary key) combinations because joining on such attributes may produce spurious tuples.

Functional Dependencies

So far we have dealt with the informal measures of database design. We now introduce a formal tool for analysis of relational schemas that enables us to detect and describe some of the above-mentioned problems in precise terms. The single most important concept in relational schema design theory is that of a functional dependency.

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 A_1, A_2, ..., A_n; let us think of the whole database as being described by a single universal relation schema R = {A_1, A_2, ..., A_n}.

Definition. A functional dependency, denoted by X → 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 t_1 and t_2 in r that have $t_1[X] = t_2[X]$, they must also have $t_1[Y] = t_2[Y]$. This means that the values of the Y component of a tuple in r depend on, or are determined by, the values of the X component; alternatively, the values of the X component of a tuple uniquely (or functionally) determine the values of the Y component. We also say that there is a functional dependency from X to Y, or that Y is functionally dependent on X. The abbreviation for functional dependency is FD. The set of attributes X is called the left-hand side of the FD, and Y is called the right-hand side.

- If a constraint on R states that there cannot be more than one tuple with a given X-value in any relation state r(R)—that is, X is a candidate key of R—this implies that X → 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 → R.
- If X → Y in R, this does not say whether or not Y → X in R.

A functional dependency is a property of the semantics or meaning of the attributes. The database designers will use their understanding of the semantics of the attributes of R—that is, how they relate to one another—to specify the functional dependencies that should hold on all relation states (extensions) r of R. Whenever the semantics of two sets of attributes in R indicate that a functional dependency should hold, we specify the dependency as a constraint.

Consider the relation schema EMP_PROJ; from the semantics of the attributes and the relation, we know that the following functional dependencies should hold:

a) Ssn → Ename
b) Pnumber →{ Pname , Plocation }
c) { Ssn, Pnumber } → 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), and (c) a combination of Ssn and Pnumber values uniquely determines the number of hours the employee currently works on the project per week (Hours).

A functional dependency is a property of the relation schema R, not of a particular legal relation state r of R. Therefore, an FD cannot be inferred automatically from a given relation extension r but must be defined explicitly by someone who knows the semantics of the attributes of R. For example, Table below shows a particular state of the TEACH relation schema. Although at first glance we may think that Text→Course, we cannot confirm this unless we know that it is true for all possible legal states of TEACH. It is, however, sufficient to demonstrate a single counterexample to disprove a functional dependency. For example,

because 'Smith' teaches both 'Data Structures' and 'Data Management,' we can conclude that Teacher does not functionally determine Course.

TEACH

Teacher	Course	Text
Smith	Data Structures	Bartram
Smith	Data Management	Martin
Hall	Compilers	Hoffman
Brown	Data Structures	Horowitz

See the illustrative example in Table below. Here, the following FDs may hold because the four tuples in the current extension have no violation of these constraints: B → C; C → B; {A,B} → C; {A,B} → D; and {C,D} → B. However, the following do not hold because we already have violations of them in the given extension: A → B (tuples 1 and 2 violate this constraint); B → A (tuples 2 and 3 violate this constraint); D → C (tuples 3 and 4 violate it).

A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d2
a2	b2	c2	d3
a3	b3	c4	d3

Normal Forms Based on Primary Keys

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 .

Following either of these approaches, it is then useful to evaluate the relations for goodness and decompose them further as needed to achieve higher normal forms, using the normalization theory.

Normalization of Relations

The normalization process, as first proposed by Codd, takes a relation schema through a series of tests to certify whether it satisfies a certain normal form. The process, which proceeds in a top-down fashion by evaluating each relation against the criteria for normal forms and decomposing relations as necessary, can thus be considered as relational design by analysis. Initially, Codd proposed three normal forms, which he called first, second, and third normal form. A stronger definition of 3NF—called Boyce-Codd normal form (BCNF)—was proposed later by Boyce and Codd. All these normal forms are based on a single analytical tool: the functional dependencies among the attributes of a relation. Later, a fourth normal

form (4NF) and a fifth normal form (5NF) were proposed, based on the concepts of multivalued dependencies and join dependencies, respectively.

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 to make the design have successively better quality. Thus, 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.

Normal forms, when considered in isolation from other factors, do not guarantee a good database design. It is generally not sufficient to check separately that each relation schema in the database is, say, in BCNF or 3NF. Rather, the process of normalization through decomposition must also confirm the existence of additional properties that the relational schemas, taken together, should possess. These would include two properties:

- The nonadditive join or lossless join property, which guarantees that the spurious tuple generation problem discussed, does not occur with respect to the relation schemas created after decomposition.
- The dependency preservation property, which ensures that each functional dependency is represented in some individual relation resulting after decomposition.

The nonadditive join property is extremely critical and must be achieved at any cost, whereas the dependency preservation property, although desirable, is some-times sacrificed.

Practical Use of Normal Forms

Normalization is carried out in practice so that the resulting designs are of high quality and meet the desirable properties stated previously. The database design as practiced in industry today pays particular attention to normalization only up to 3NF, BCNF, or at most 4NF.

Another point worth noting is that the database designers need not normalize to the highest possible normal form. Relations may be left in a lower normalization status, such as 2NF, for performance reasons. Doing so incurs the corresponding penalties of dealing with the anomalies.

Definition. **Denormalization** is the process of storing the join of higher normal form relations as a base relation, which is in a lower normal form.

Definitions of Keys and Attributes Participating in Keys

Definition. A **superkey** of a relation schema $R = \{A_1, A_2, ..., A_n\}$ is a set of attributes $S \subseteq R$ with the property that no two tuples t_1 and t_2 in any legal relation state r of R will have $t_1[S] = t_2[S]$. 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 $K = \{A_1, A_2, ..., A_k\}$ of R, then $K - \{A_i\}$ is not a key of R for any A_i, $1 \leq i \leq k$. If a relation schema has more than one key, each is called a **candidate key**.

Definition. An attribute of relation schema R is called a prime attribute of R if it is a member of some candidate key of R. An attribute is called nonprime if it is not a prime attribute—that is, if it is not a member of any candidate key. Both Ssn and Pnumber are

prime attributes of WORKS_ON, whereas other attributes of WORKS_ON are nonprime.

First Normal Form

First normal form was defined to disallow multivalued attributes, composite attributes, and their combinations. It 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. Hence, 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a single tuple. The only attribute values permitted by 1NF are single atomic (or indivisible) values.

Consider the DEPARTMENT relation schema, whose primary key is Dnumber, and suppose that we extend it by including the Dlocations attribute as shown in (a). We assume that each department can have a number of locations. The DEPARTMENT schema and a sample relation state are shown in Figure. 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 .
- 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.

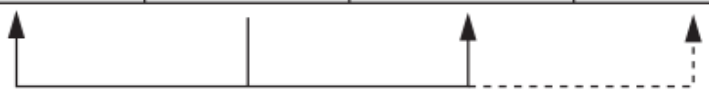
In either case, the DEPARTMENT relation in Figure is not in 1NF. There are three main techniques to achieve first normal form for such a relation:

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}. This decomposes the non-1NF relation into two 1NF relations.
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(c). In this case, the primary key becomes the combination { Dnumber, Dlocation }. This solution has the disadvantage of introducing redundancy in the relation.
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.

(a)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations



(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

Of the three solutions above, the first is generally considered best because it does not suffer from redundancy and it is completely general, having no limit placed on a maximum number of values. In fact, if we choose the second solution, it will be decomposed further during subsequent normalization steps into the first solution.

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. Figure below shows how the EMP_PROJ relation could appear if nesting is allowed. 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

(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

each project. The schema of this EMP_PROJ relation can be represented as follows:

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

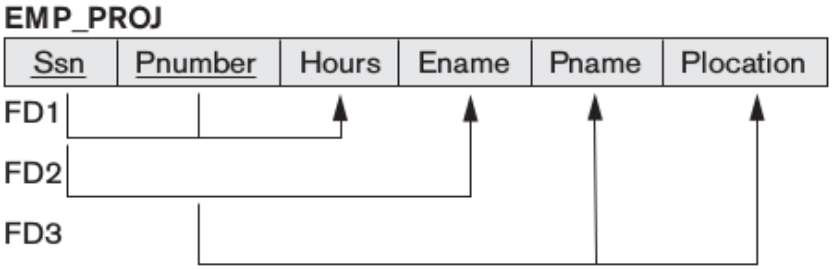
The set braces{ } identify the attribute PROJS as multivalued, and we list the component attributes that form PROJS between parentheses ( ). Notice that Ssn is the primary key of the EMP_PROJ relation in Figures (a) and (b), while Pnumber is the partial key of the nested relation; that is, within each tuple, the nested relation must have unique values of Pnumber.

To normalize this into 1NF, we remove the nested relation attributes into a new relation and propagate the primary key into it; the primary key of the new relation will combine the partial key with the primary key of the original relation. Decomposition and primary key propagation yield the schemas EMP_PROJ1 and EMP_PROJ2 , as shown in Figure (c).

Second Normal Form

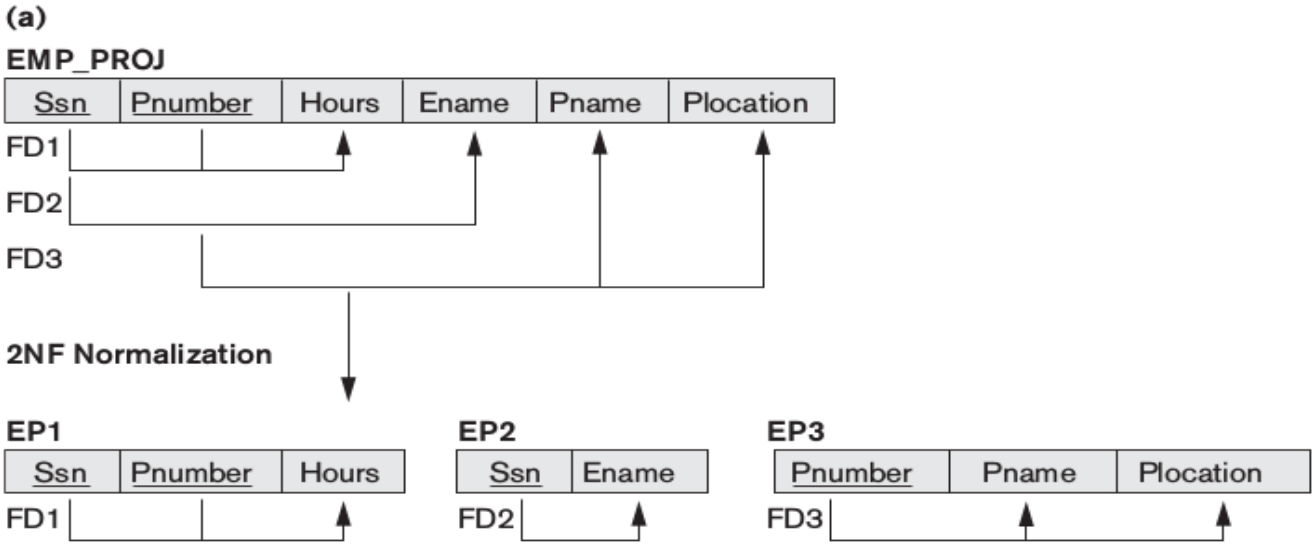
Second normal form (2NF) is based on the concept of full functional dependency. A functional dependency X → Y is a full functional dependency if removal of any attribute A from X means that the dependency does not hold any more; that is, for any attribute A Є X, (X – {A}) does not functionally determine Y. A functional dependency X → Y is a partial dependency if some attribute A Є X can be removed from X and the dependency still holds; that is, for some A Є X, (X – {A}) → Y.

In Figure below, { 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.



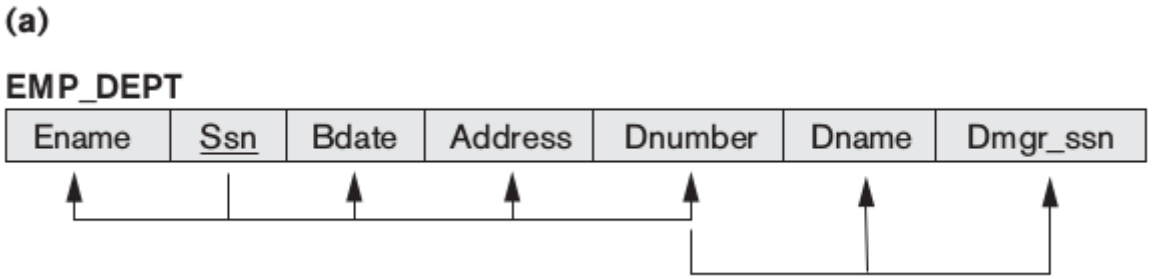
Definition. A relation schema R is in 2NF if every nonprime attribute A in R is fully functionally dependent on the primary key of R. The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all. The EMP_PROJ relation in Figure above 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.

If a relation schema is not in 2NF, it can be second normalized or 2NF normalized into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent. Therefore, the functional dependencies FD1, FD2, and FD3 in Figure above lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure below, each of which is in 2NF.



Third Normal Form

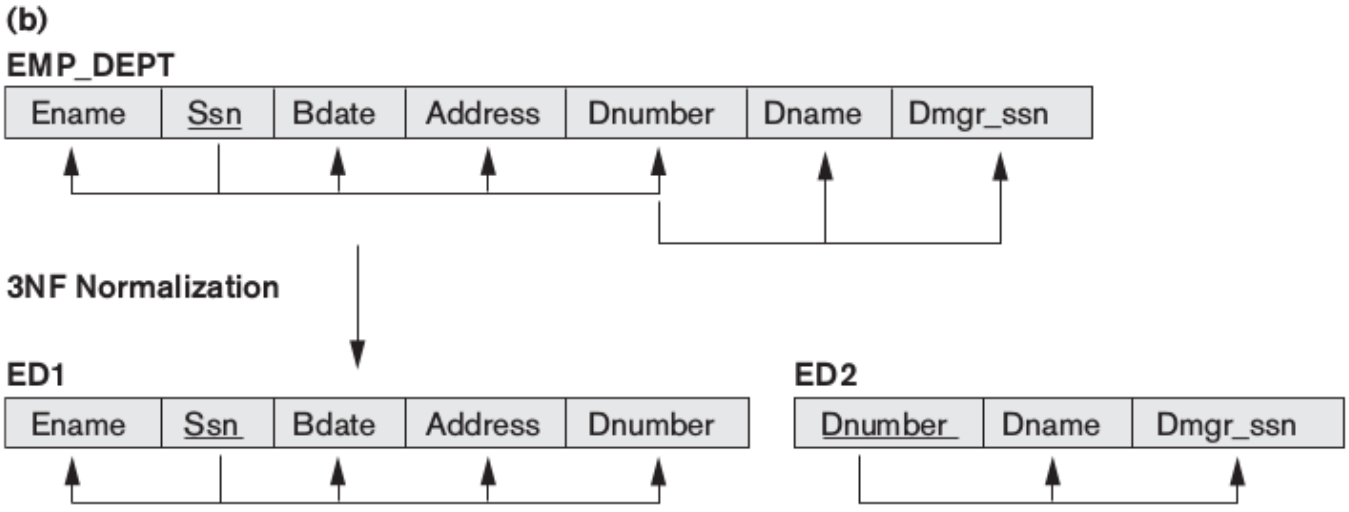
Third normal form (3NF) is based on the concept of transitive dependency. A functional dependency X → Y in a relation schema R is a transitive dependency if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R, and both X → Z and Z → Y hold. The dependency Ssn → Dmgr_ssn is transitive through Dnumber in EMP_DEPT in Figure below, because both the dependencies Ssn → Dnumber and



Dnumber → Dmgr_ssn hold and Dnumber is neither a key itself nor a subset of the key of EMP_DEPT. Intuitively, we can see that the dependency of Dmgr_ssn on Dnumber is undesirable in EMP_DEPT since Dnumber is not a key of EMP_DEPT .

Definition. According to Codd's original definition, a relation schema R is in 3NF if it satisfies 2NF and no nonprime attribute of R is transitively dependent on the primary key. The relation schema EMP_DEPT in Figure above is in 2NF, since no partial dependencies on a key exist. However, EMP_DEPT is not in 3NF because of the transitive dependency of Dmgr_ssn (and also Dname) on Ssn via Dnumber.

We can normalize EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2 shown in Figure below. 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

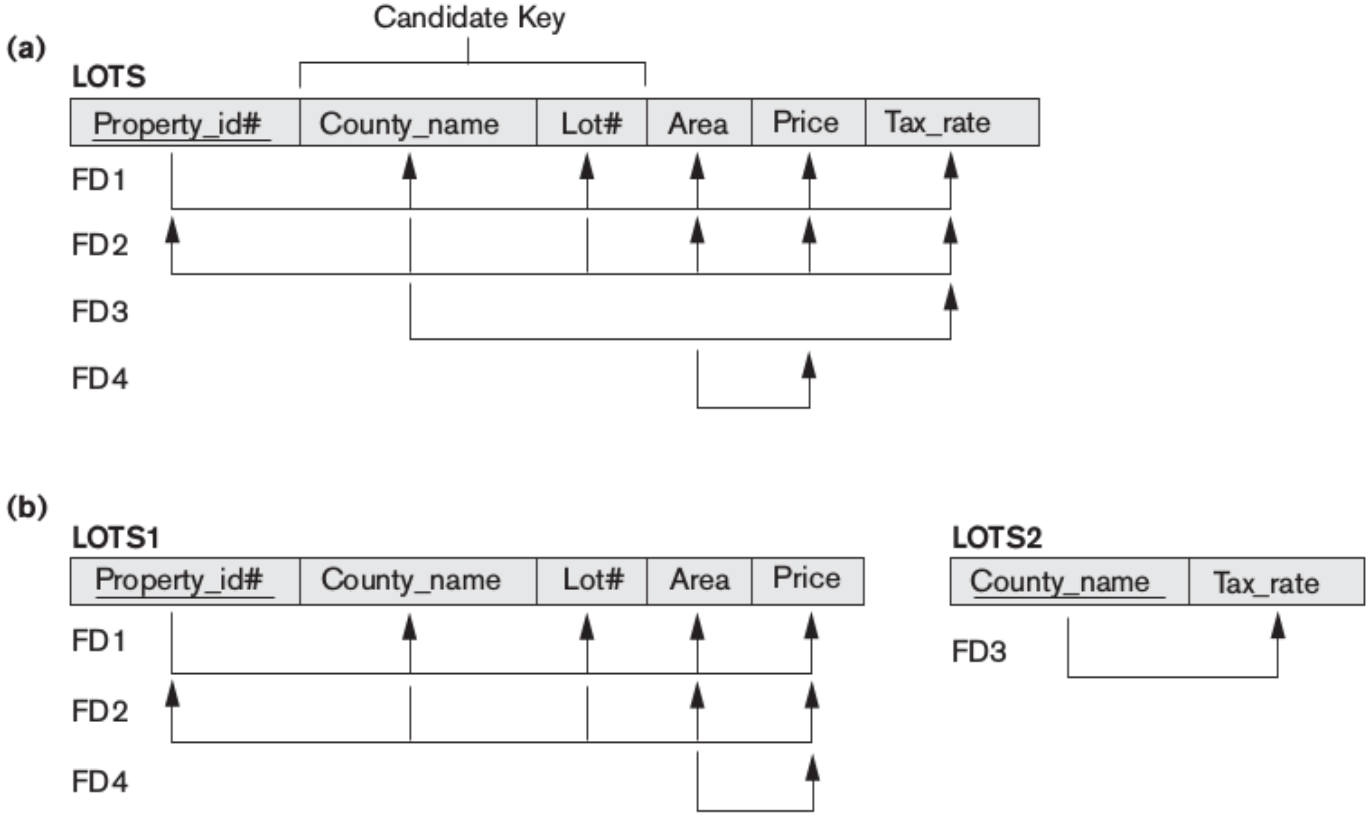
In general, we want to design our relation schemas so that they have neither partial nor transitive dependencies because these types of dependencies cause the update anomalies. The steps for normalization into 3NF relations that we have discussed so far disallow partial and transitive dependencies on the primary key.

The normalization procedure described so far is useful for analysis in practical situations for a given database where primary keys have already been defined. These definitions, however, do not take other candidate keys of a relation, if any, into account. As a

general definition of prime attribute, an attribute that is part of any candidate key will be considered as prime. Partial and full functional dependencies and transitive dependencies will now be considered with respect to all candidate keys of a relation.

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 key. If the key contains a single attribute, the test need not be applied at all. Consider the relation schema LOTS shown in Figure below, 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 hold. We choose Property_id# as the primary key. Suppose that the following two additional functional dependencies hold in LOTS :

FD3: County_name → Tax_rate

FD4: Area → Price

In words, the dependency FD3 says that the tax rate is fixed for a given county (does not vary lot by lot within the same county), while FD4 says that the price of a lot is determined by its area regardless of which county it is in. The LOTS relation schema violates the general definition of 2NF because Tax_rate is partially dependent on the candidate key { County_name, Lot# }, due to FD3.

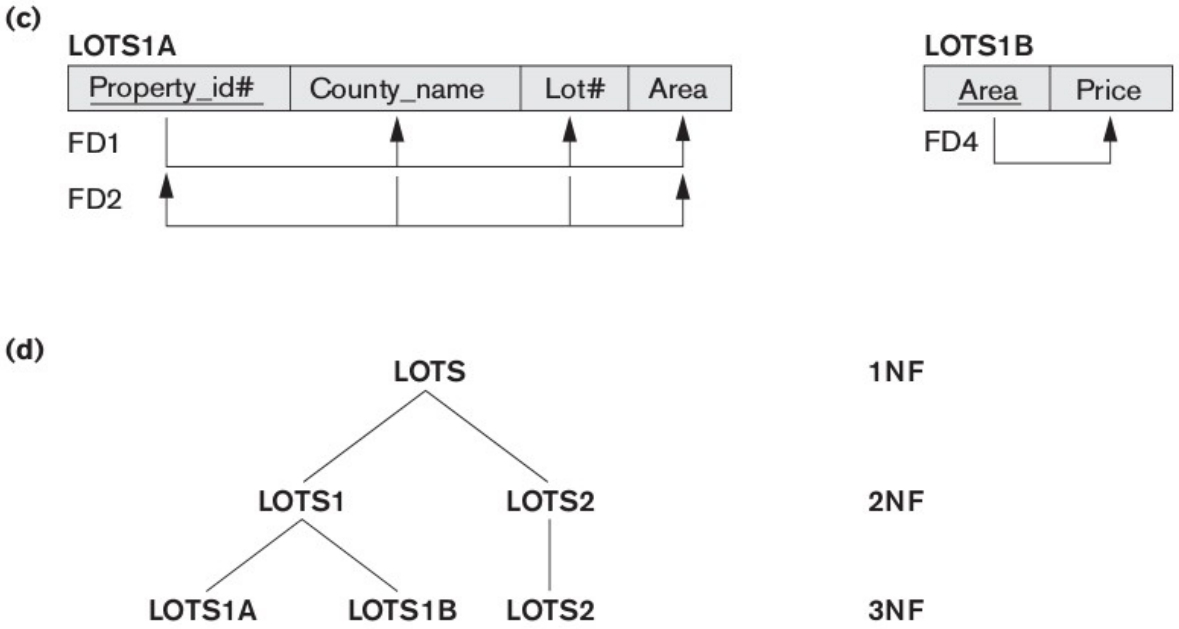
To normalize LOTS into 2NF, we decompose it into the two relations LOTS1 and LOTS2, shown in Figure(b). We construct LOTS1 by removing the attribute Tax_rate that violates 2NF from LOTS and placing it with County_name (the left-hand side of FD3 that

causes the partial dependency) into another relation LOTS2. Both LOTS1 and LOTS2 are in 2NF. Notice that FD4 does not violate 2NF and is carried over to LOTS1.

General Definition of Third Normal Form

Definition. 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 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 below. We construct LOTS1A by removing the attribute Price that violates 3NF from LOTS1 and placing it with Area (the left hand 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.



Interpreting the General Definition of Third Normal Form

A relation schema R violates the general definition of 3NF if a functional dependency X → A holds in R that does not meet either condition—meaning that it violates both conditions (a) and (b) of 3NF. This can occur due to two types of problematic functional dependencies:

- A nonprime attribute determines another nonprime attribute. Here we typically have a transitive dependency that violates 3NF.
- A proper subset of a key of R functionally determines a nonprime attribute. Here we have a partial dependency that violates 3NF (and also 2NF).

Therefore, we can state a general alternative definition of 3NF as follows:

Alternative Definition. A relation schema R is in 3NF if every nonprime attribute of R meets both of the following conditions:

- It is fully functionally dependent on every key of R.
- It is nontransitively dependent on every key of R.