chapter 15

# Basics of Functional Dependencies and Normalization for Relational Databases

In Chapters 3 through 6, we presented various aspects of the relational model and the languages associated with it. Each *relation schema* consists of a number of attributes, and the *relational database schema* consists of a number of relation schemas. So far, we have assumed that attributes are grouped to form a relation schema by using the common sense of the database designer or by mapping a database schema design from a conceptual data model such as the ER or Enhanced-ER (EER) data model. These models make the designer identify entity types and relationship types and their respective attributes, which leads to a natural and logical grouping of the attributes into relations when the mapping procedures discussed in Chapter 9 are followed. However, we still need some formal way of analyzing why one grouping of attributes into a relation schema may be better than another. While discussing database design in Chapters 7 through 10, we did not develop any measure of appropriateness or *goodness* to measure the quality of the design, other than the intuition of the designer. In this chapter we discuss some of the theory that has been developed with the goal of evaluating relational schemas for design quality—that is, to measure formally why one set of groupings of attributes into relation schemas is better than another.

There are two levels at which we can discuss the *goodness* of relation schemas. The first is the **logical** (or **conceptual**) **level**—how users interpret the relation schemas and the meaning of their attributes. Having good relation schemas at this level enables users to understand clearly the meaning of the data in the relations, and hence to formulate their queries correctly. The second is the **implementation** (or

**physical storage**) **level**—how the tuples in a base relation are stored and updated. This level applies only to schemas of base relations—which will be physically stored as files—whereas at the logical level we are interested in schemas of both base relations and views (virtual relations). The relational database design theory developed in this chapter applies mainly to *base relations,* although some criteria of appropriateness also apply to views, as shown in Section 15.1.

As with many design problems, database design may be performed using two approaches: bottom-up or top-down. A **bottom-up design methodology** (also called *design by synthesis*) considers the basic relationships *among individual attributes* as the starting point and uses those to construct relation schemas. This approach is not very popular in practice[1] because it suffers from the problem of having to collect a large number of binary relationships among attributes as the starting point. For practical situations, it is next to impossible to capture binary relationships among all such pairs of attributes. In contrast, a **top-down design methodology** (also called *design by analysis*) starts with a number of groupings of attributes into relations that exist together naturally, for example, on an invoice, a form, or a report. The relations are then analyzed individually and collectively, leading to further decomposition until all desirable properties are met. The theory described in this chapter is applicable to both the top-down and bottom-up design approaches, but is more appropriate when used with the top-down approach.

Relational database design ultimately produces a set of relations. The implicit goals of the design activity are *information preservation* and *minimum redundancy*. Information is very hard to quantify—hence we consider information preservation in terms of maintaining all concepts, including attribute types, entity types, and relationship types as well as generalization/specialization relationships, which are described using a model such as the EER model. Thus, the relational design must preserve all of these concepts, which are originally captured in the conceptual design after the conceptual to logical design mapping. Minimizing redundancy implies minimizing redundant storage of the same information and reducing the need for multiple updates to maintain consistency across multiple copies of the same information in response to real-world events that require making an update.

We start this chapter by informally discussing some criteria for good and bad relation schemas in Section 15.1. In Section 15.2, we define the concept of *functional dependency,* a formal constraint among attributes that is the main tool for formally measuring the appropriateness of attribute groupings into relation schemas. In Section 15.3, we discuss normal forms and the process of normalization using functional dependencies. Successive normal forms are defined to meet a set of desirable constraints expressed using functional dependencies. The normalization procedure consists of applying a series of tests to relations to meet these increasingly stringent requirements and decompose the relations when necessary. In Section 15.4, we dis-

---

[1]An exception in which this approach is used in practice is based on a model called the *binary relational model.* An example is the NIAM methodology (Verheijen and VanBekkum, 1982).

cuss more general definitions of normal forms that can be directly applied to any given design and do not require step-by-step analysis and normalization. Sections 15.5 to 15.7 discuss further normal forms up to the fifth normal form. In Section 15.6 we introduce the multivalued dependency (MVD), followed by the join dependency (JD) in Section 15.7. Section 15.8 summarizes the chapter.

Chapter 16 continues the development of the theory related to the design of good relational schemas. We discuss desirable properties of relational decomposition—nonadditive join property and functional dependency preservation property. A general algorithm that tests whether or not a decomposition has the nonadditive (or *lossless*) join property (Algorithm 16.3 is also presented). We then discuss properties of functional dependencies and the concept of a minimal cover of dependencies. We consider the bottom-up approach to database design consisting of a set of algorithms to design relations in a desired normal form. These algorithms assume as input a given set of functional dependencies and achieve a relational design in a target normal form while adhering to the above desirable properties. In Chapter 16 we also define additional types of dependencies that further enhance the evaluation of the *goodness* of relation schemas.

If Chapter 16 is not covered in a course, we recommend a quick introduction to the desirable properties of decomposition and the discussion of Property NJB in Section 16.2.

# 15.1 Informal Design Guidelines for Relation Schemas

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

- Making sure that the semantics of the attributes is clear in the schema
- Reducing the redundant information in tuples
- Reducing the NULL values in tuples
- Disallowing the possibility of generating spurious tuples

These measures are not always independent of one another, as we will see.

## 15.1.1 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 Chapter 3 we discussed how a relation can be interpreted as a set of facts. If the conceptual design described in Chapters 7 and 8 is done carefully and the mapping procedure in Chapter 9 is followed systematically, the relational schema design should have a clear meaning.

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 15.1, a simplified version of the COMPANY relational database schema in Figure 3.5, and Figure 15.2, which presents an example of populated relation states of this 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.

**Figure 15.1**
A simplified COMPANY relational database schema.

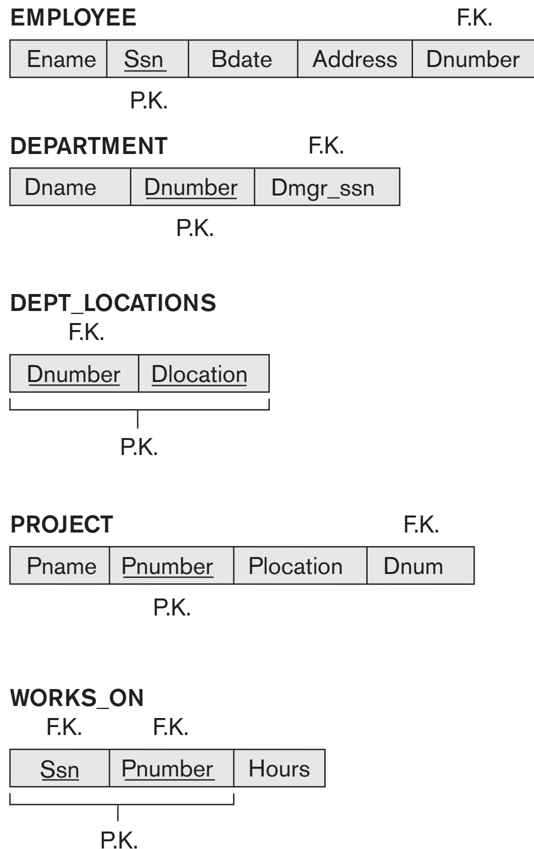


**Figure 15.2**
Sample database state for the relational database schema in Figure 15.1.

**EMPLOYEE**

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

**DEPARTMENT**

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

**DEPT_LOCATIONS**

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

**WORKS_ON**

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

**PROJECT**

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

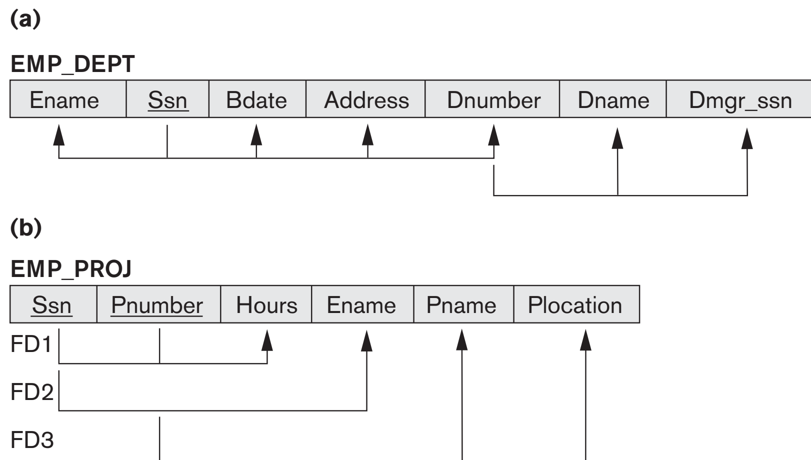
The semantics of the other two relation schemas in Figure 15.1 are slightly more complex. Each tuple in DEPT_LOCATIONS gives a department number (Dnumber) and *one of* the locations of the department (Dlocation). Each tuple in WORKS_ON gives an employee Social Security number (Ssn), the project number 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). However, both schemas have a well-defined and unambiguous interpretation. The schema DEPT_LOCATIONS represents a multivalued attribute of DEPARTMENT, whereas WORKS_ON represents an M:N relationship between EMPLOYEE and PROJECT. Hence, all the relation schemas in Figure 15.1 may be considered as easy to explain and therefore good from the standpoint of having clear semantics. We can thus formulate the following informal design guideline.

## 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 15.3(a) and 15.3(b) also have clear semantics. (The reader should ignore the lines under the relations for now; they are used to illustrate functional dependency notation, discussed in Section 15.2.) A tuple in the EMP_DEPT relation schema in Figure 15.3(a) 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 in Figure 15.3(b), each tuple relates an employee to a project but also includes

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



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. Hence, they fare poorly against the above measure of design quality. They may be used as views, but they cause problems when used as base relations, as we discuss in the following section.

### 15.1.2 Redundant Information in Tuples and Update Anomalies

One goal of schema design is to minimize the storage space used by the base relations (and hence the corresponding files). Grouping attributes into relation schemas has a significant effect on storage space. For example, compare the space used by the two base relations EMPLOYEE and DEPARTMENT in Figure 15.2 with that for an EMP_DEPT base relation in Figure 15.4, which is the result of applying the NATURAL JOIN operation to EMPLOYEE and DEPARTMENT. In EMP_DEPT, the attribute values pertaining to a particular department (Dnumber, Dname, Dmgr_ssn) are repeated for *every employee who works for that department.* In contrast, each department's information appears only once in the DEPARTMENT relation in Figure 15.2. Only the department number (Dnumber) is repeated in the EMPLOYEE relation for each employee who works in that department as a foreign key. Similar comments apply to the EMP_PROJ relation (see Figure 15.4), which augments the WORKS_ON relation with additional attributes from EMPLOYEE and PROJECT.

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.[2]

**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. In the design of Figure 15.2, we do not have to worry about this consistency problem because we enter only the department number in the employee tuple; all other attribute values of department 5 are recorded only once in the database, as a single tuple in the DEPARTMENT relation.
- 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

[2]These anomalies were identified by Codd (1972a) to justify the need for normalization of relations, as we shall discuss in Section 15.3.

Redundancy

**EMP_DEPT**

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

Redundancy Redundancy

**EMP_PROJ**

| Ssn | Pnumber | Hours | Ename | Pname | Plocation |
|---|---|---|---|---|---|
| 123456789 | 1 | 32.5 | Smith, John B. | ProductX | Bellaire |
| 123456789 | 2 | 7.5 | Smith, John B. | ProductY | Sugarland |
| 666884444 | 3 | 40.0 | Narayan, Ramesh K. | ProductZ | Houston |
| 453453453 | 1 | 20.0 | English, Joyce A. | ProductX | Bellaire |
| 453453453 | 2 | 20.0 | English, Joyce A. | ProductY | Sugarland |
| 333445555 | 2 | 10.0 | Wong, Franklin T. | ProductY | Sugarland |
| 333445555 | 3 | 10.0 | Wong, Franklin T. | ProductZ | Houston |
| 333445555 | 10 | 10.0 | Wong, Franklin T. | Computerization | Stafford |
| 333445555 | 20 | 10.0 | Wong, Franklin T. | Reorganization | Houston |
| 999887777 | 30 | 30.0 | Zelaya, Alicia J. | Newbenefits | Stafford |
| 999887777 | 10 | 10.0 | Zelaya, Alicia J. | Computerization | Stafford |
| 987987987 | 10 | 35.0 | Jabbar, Ahmad V. | Computerization | Stafford |
| 987987987 | 30 | 5.0 | Jabbar, Ahmad V. | Newbenefits | Stafford |
| 987654321 | 30 | 20.0 | Wallace, Jennifer S. | Newbenefits | Stafford |
| 987654321 | 20 | 15.0 | Wallace, Jennifer S. | Reorganization | Houston |
| 888665555 | 20 | Null | Borg, James E. | Reorganization | Houston |

**Figure 15.4**
Sample states for EMP_DEPT and EMP_PROJ resulting from applying NATURAL JOIN to the relations in Figure 15.2. These may be stored as base relations for performance reasons.

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. This problem does not occur in the design of Figure 15.2 because a department is entered in the DEPARTMENT relation whether or not any employees work for it, and whenever an employee is assigned to that department, a corresponding tuple is inserted in EMPLOYEE.

**Deletion Anomalies.** The problem of deletion anomalies is related to the second insertion anomaly situation just discussed. If we delete from EMP_DEPT an employee tuple that happens to represent the last employee working for a particular department, the information concerning that department is lost from the database. This problem does not occur in the database of Figure 15.2 because DEPARTMENT tuples are stored separately.

**Modification Anomalies.** In EMP_DEPT, if we change the value of one of the attributes of a particular department—say, the manager of department 5—we must update the tuples of *all* employees who work in that department; otherwise, the database will become inconsistent. If we fail to update some tuples, the same department will be shown to have two different values for manager in different employee tuples, which would be wrong.[3]

It is easy to see that these three anomalies are undesirable and cause difficulties to maintain consistency of data as well as require unnecessary updates that can be avoided; hence, we can state the next guideline as follows.

### Guideline 2

Design the base relation schemas so that no insertion, deletion, or modification anomalies are present in the relations. If any anomalies are present,[4] note them clearly and make sure that the programs that update the database will operate correctly.

The second guideline is consistent with and, in a way, a restatement of the first guideline. We can also see the need for a more formal approach to evaluating whether a design meets these guidelines. Sections 15.2 through 15.4 provide these needed formal concepts. It is important to note that these guidelines may sometimes *have to be violated* in order to *improve the performance* of certain queries. If EMP_DEPT is used as a stored relation (known otherwise as a *materialized view*) in addition to the base relations of EMPLOYEE and DEPARTMENT, the anomalies in EMP_DEPT must be noted and accounted for (for example, by using triggers or stored procedures that would make automatic updates). This way, whenever the base relation is updated, we do not end up with inconsistencies. In general, it is advisable to use anomaly-free base relations and to specify views that include the joins for placing together the attributes frequently referenced in important queries.

## 15.1.3 NULL Values in Tuples

In some schema designs we may group many attributes together into a "fat" relation. If many of the attributes do not apply to all tuples in the relation, we end up with many NULLs in those tuples. This can waste space at the storage level and may

---

[3]This is not as serious as the other problems, because all tuples can be updated by a single SQL query.

[4]Other application considerations may dictate and make certain anomalies unavoidable. For example, the EMP_DEPT relation may correspond to a query or a report that is frequently required.

also lead to problems with understanding the meaning of the attributes and with specifying JOIN operations at the logical level.[5] Another problem with NULLs is how to account for them when aggregate operations such as COUNT or SUM are applied. SELECT and JOIN operations involve comparisons; if NULL values are present, the results may become unpredictable.[6] Moreover, NULLs can have multiple interpretations, such as the following:

- The attribute *does not apply* to this tuple. For example, Visa_status may not apply to U.S. students.
- The attribute value for this tuple is *unknown*. For example, the Date_of_birth may be unknown for an employee.
- The value is *known but absent*; that is, it has not been recorded yet. For example, the Home_Phone_Number for an employee may exist, but may not be available and recorded yet.

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.

Using space efficiently and avoiding joins with NULL values are the two overriding criteria that determine whether to include the columns that may have NULLs in a relation or to have a separate relation for those columns (with the appropriate key columns). 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.

## 15.1.4 Generation of Spurious Tuples

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

---

[5]This is because inner and outer joins produce different results when NULLs are involved in joins. The users must thus be aware of the different meanings of the various types of joins. Although this is reasonable for sophisticated users, it may be difficult for others.

[6]In Section 5.5.1 we presented comparisons involving NULL values where the outcome (in three-valued logic) are TRUE, FALSE, and UNKNOWN.

(a)

**EMP_LOCS**

| Ename | Plocation |
|---|---|

P.K.

**EMP_PROJ1**

| Ssn | Pnumber | Hours | Pname | Plocation |
|---|---|---|---|---|

P.K.

(b)

**EMP_LOCS**

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

**EMP_PROJ1**

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

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

EMP_PROJ relation in Figure 15.4, which are obtained by applying the appropriate PROJECT ($\pi$) operations to EMP_PROJ (ignore the dashed lines in Figure 15.5(b) for now).

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

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

*
*
*

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

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

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. We can now informally state another design guideline.

## 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.

This informal guideline obviously needs to be stated more formally. In Section 16.2 we discuss a formal condition called the nonadditive (or lossless) join property that guarantees that certain joins do not produce spurious tuples.

### 15.1.5 Summary and Discussion of Design Guidelines

In Sections 15.1.1 through 15.1.4, we informally discussed situations that lead to problematic relation schemas and we proposed informal guidelines for a good relational design. The problems we pointed out, which can be detected without additional tools of analysis, are as follows:

- Anomalies that cause redundant work to be done during insertion into and modification of a relation, and that may cause accidental loss of information during a deletion from a relation
- Waste of storage space due to NULLs and the difficulty of performing selections, aggregation operations, and joins due to NULL values
- Generation of invalid and spurious data during joins on base relations with matched attributes that may not represent a proper (foreign key, primary key) relationship

In the rest of this chapter we present formal concepts and theory that may be used to define the *goodness* and *badness* of *individual* relation schemas more precisely. First we discuss functional dependency as a tool for analysis. Then we specify the three normal forms and Boyce-Codd normal form (BCNF) for relation schemas. The strategy for achieving a good design is to decompose a badly designed relation appropriately. We also briefly introduce additional normal forms that deal with additional dependencies. In Chapter 16, we discuss the properties of decomposition in detail, and provide algorithms that design relations bottom-up by using the functional dependencies as a starting point.

## 15.2 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. In this section we formally define the concept, and in Section 15.3 we see how it can be used to define normal forms for relation schemas.

### 15.2.1 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 $A_1, A_2, \ldots, A_n$; let us think of the whole database as being described by a single **universal**

relation schema $R = \{A_1, A_2, \ldots, A_n\}$.[7] We do not imply that we will actually store the database as a single universal table; we use this concept only in developing the formal theory of data dependencies.[8]

> **Definition.** A **functional dependency**, denoted by $X \rightarrow Y$, between two sets of attributes $X$ and $Y$ that are subsets of $R$ specifies a *constraint* on the possible tuples that can form a relation state $r$ of $R$. The constraint is that, for any two tuples $t_1$ 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** or **f.d.** The set of attributes $X$ is called the **left-hand side** of the FD, and $Y$ is called the **right-hand side**.

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

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

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. Relation extensions $r(R)$ that satisfy the functional dependency constraints are called **legal relation states** (or **legal extensions**) of $R$. Hence, the main use of functional dependencies is to describe further a relation schema $R$ by specifying constraints on its attributes that must hold *at all times.* Certain FDs can be specified without referring to a specific relation, but as a property of those attributes given their commonly understood meaning. For example, {State, Driver_license_number} $\rightarrow$ Ssn should hold for any adult in the United States and hence should hold whenever these attributes appear in a relation. It is also possible that certain functional dependencies may cease to

---

[7]This concept of a universal relation is important when we discuss the algorithms for relational database design in Chapter 16.

[8]This assumption implies that every attribute in the database should have a distinct name. In Chapter 3 we prefixed attribute names by relation names to achieve uniqueness whenever attributes in distinct relations had the same name.

exist in the real world if the relationship changes. For example, the FD Zip_code → Area_code used to exist as a relationship between postal codes and telephone number codes in the United States, but with the proliferation of telephone area codes it is no longer true.

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

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

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, Figure 15.7 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.

Given a populated relation, one cannot determine which FDs hold and which do not unless the meaning of and the relationships among the attributes are known. All one can say is that a certain FD *may* exist if it holds in that particular extension. One cannot guarantee its existence until the meaning of the corresponding attributes is clearly understood. One can, however, emphatically state that a certain FD *does not*

**TEACH**

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

**Figure 15.7**
A relation state of TEACH with a *possible* functional dependency TEXT → COURSE. However, TEACHER → COURSE is ruled out.

*hold* if there are tuples that show the violation of such an FD. See the illustrative example relation in Figure 15.8. Here, the following FDs *may hold* because the four tuples in the current extension have no violation of these constraints: $B \rightarrow C$; $C \rightarrow B$; $\{A, B\} \rightarrow C$; $\{A, B\} \rightarrow D$; and $\{C, D\} \rightarrow B$. However, the following *do not* hold because we already have violations of them in the given extension: $A \rightarrow B$ (tuples 1 and 2 violate this constraint); $B \rightarrow A$ (tuples 2 and 3 violate this constraint); $D \rightarrow C$ (tuples 3 and 4 violate it).

Figure 15.3 introduces a **diagrammatic notation** for displaying FDs: Each FD is displayed as a horizontal line. The left-hand-side attributes of the FD are connected by vertical lines to the line representing the FD, while the right-hand-side attributes are connected by the lines with arrows pointing toward the attributes.

We denote by $F$ the set of functional dependencies that are specified on relation schema $R$. Typically, the schema designer specifies the functional dependencies that are *semantically obvious*; usually, however, numerous other functional dependencies hold in *all* legal relation instances among sets of attributes that can be derived from and satisfy the dependencies in $F$. Those other dependencies can be *inferred* or *deduced* from the FDs in $F$. We defer the details of inference rules and properties of functional dependencies to Chapter 16.

## 15.3 Normal Forms Based on Primary Keys

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

- Perform a conceptual schema design using a conceptual model such as ER or EER and map the conceptual design into a set of relations
- Design the relations based on external knowledge derived from an existing implementation of files or forms or reports

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 presented in this chapter and the next. We focus in

**Figure 15.8**
A relation *R* (A, B, C, D) with its extension.

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

this section on the first three normal forms for relation schemas and the intuition behind them, and discuss how they were developed historically. More general definitions of these normal forms, which take into account all candidate keys of a relation rather than just the primary key, are deferred to Section 15.4.

We start by informally discussing normal forms and the motivation behind their development, as well as reviewing some definitions from Chapter 3 that are needed here. Then we discuss the first normal form (1NF) in Section 15.3.4, and present the definitions of second normal form (2NF) and third normal form (3NF), which are based on primary keys, in Sections 15.3.5 and 15.3.6, respectively.

### 15.3.1 Normalization of Relations

The normalization process, as first proposed by Codd (1972a), takes a relation schema through a series of tests to *certify* whether it satisfies a certain **normal form**. 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; these are briefly discussed in Sections 15.6 and 15.7.

**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 discussed in Section 15.1.2. It can be considered as a "filtering" or "purification" process to make the design have successively better quality. Unsatisfactory relation schemas that do not meet certain conditions—the **normal form tests**—are decomposed into smaller relation schemas that meet the tests and hence possess the desirable properties. 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 in Section 15.1.4 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 sometimes sacrificed, as we discuss in Section 16.1.2. We defer the presentation of the formal concepts and techniques that guarantee the above two properties to Chapter 16.

### 15.3.2 Practical Use of Normal Forms

Most practical design projects acquire existing designs of databases from previous designs, designs in legacy models, or from existing files. Normalization is carried out in practice so that the resulting designs are of high quality and meet the desirable properties stated previously. Although several higher normal forms have been defined, such as the 4NF and 5NF that we discuss in Sections 15.6 and 15.7, the practical utility of these normal forms becomes questionable when the constraints on which they are based are rare, and hard to understand or to detect by the database designers and users who must discover these constraints. Thus, 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, such as those discussed at the end of Section 15.1.2. 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.

### 15.3.3 Definitions of Keys and Attributes Participating in Keys

Before proceeding further, let's look again at the definitions of keys of a relation schema from Chapter 3.

**Definition.** A **superkey** of a relation schema $R = \{A_1, A_2, \ldots, A_n\}$ 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 $A_i$, $1 \leq i \leq k$. In Figure 15.1, {Ssn} is a key for EMPLOYEE, whereas {Ssn}, {Ssn, Ename}, {Ssn, Ename, Bdate}, and any set of attributes that includes Ssn are all superkeys.

If a relation schema has more than one key, each is called a **candidate key**. One of the candidate keys is *arbitrarily* designated to be the **primary key**, and the others are called secondary keys. In a practical relational database, each relation schema must have a primary key. If no candidate key is known for a relation, the entire relation can be treated as a default superkey. In Figure 15.1, {Ssn} is the only candidate key for EMPLOYEE, so it is also the primary 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.

In Figure 15.1, both Ssn and Pnumber are prime attributes of WORKS_ON, whereas other attributes of WORKS_ON are nonprime.

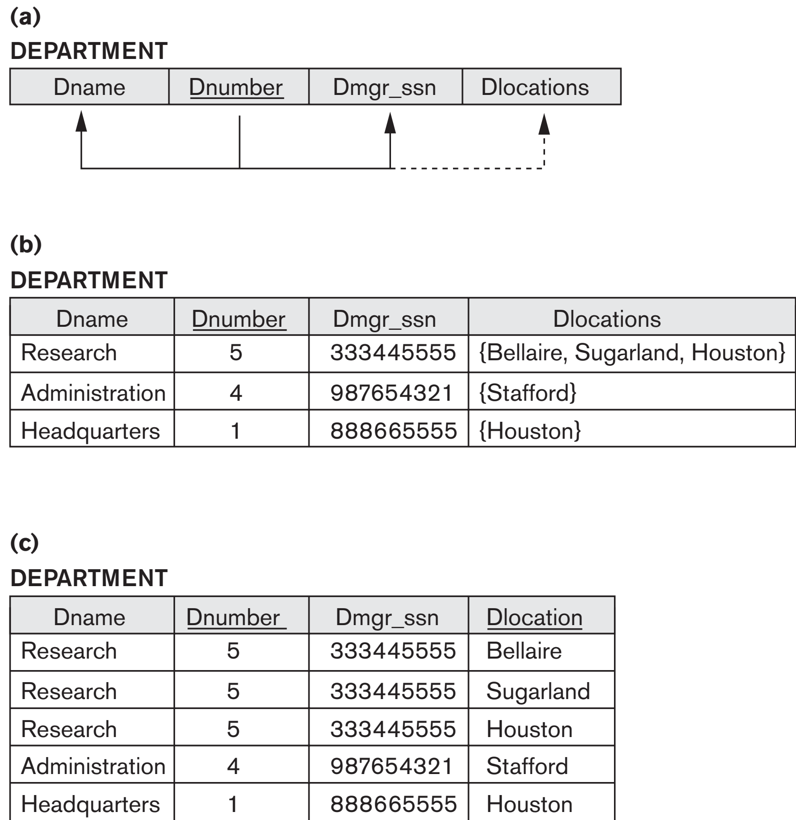
We now present the first three normal forms: 1NF, 2NF, and 3NF. These were proposed by Codd (1972a) as a sequence to achieve the desirable state of 3NF relations by progressing through the intermediate states of 1NF and 2NF if needed. As we shall see, 2NF and 3NF attack different problems. However, for historical reasons, it is customary to follow them in that sequence; hence, by definition a 3NF relation *already satisfies* 2NF.

### 15.3.4 First Normal Form

**First normal form (1NF)** is now considered to be part of the formal definition of a relation in the basic (flat) relational model; historically, it was defined to disallow multivalued attributes, composite attributes, and their combinations. 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.* In other words, 1NF disallows *relations within relations* or *relations as attribute values within tuples.* The only attribute values permitted by 1NF are single **atomic** (or **indivisible**) **values**.

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

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


**(a)**

**DEPARTMENT**

| 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 |

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

- The domain of Dlocations contains sets of values and hence is nonatomic. In this case, Dnumber $\rightarrow$ Dlocations because each set is considered a single member of the attribute domain.[9]

In either case, the DEPARTMENT relation in Figure 15.9 is not in 1NF; in fact, it does not even qualify as a relation according to our definition of relation in Section 3.1. 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}, as shown in Figure 15.2. A distinct tuple in DEPT_LOCATIONS exists for *each location* of a department. This decomposes the non-1NF relation into two 1NF relations.

[9]In this case we can consider the domain of Dlocations to be the **power set** of the set of single locations; that is, the domain is made up of all possible subsets of the set of single locations.

2. Expand the key so that there will be a separate tuple in the original DEPARTMENT relation for each location of a DEPARTMENT, as shown in Figure 15.9(c). 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. It further introduces spurious semantics about the ordering among the location values that is not originally intended. Querying on this attribute becomes more difficult; for example, consider how you would write the query: *List the departments that have 'Bellaire' as one of their locations* in this design.

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

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

The set braces { } identify the attribute PROJS as multivalued, and we list the component attributes that form PROJS between parentheses ( ). Interestingly, recent trends for supporting complex objects (see Chapter 11) and XML data (see Chapter 12) attempt to allow and formalize nested relations within relational database systems, which were disallowed early on by 1NF.

Notice that Ssn is the primary key of the EMP_PROJ relation in Figures 15.10(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 15.10(c).

This procedure can be applied recursively to a relation with multiple-level nesting to **unnest** the relation into a set of 1NF relations. This is useful in converting an unnormalized relation schema with many levels of nesting into 1NF relations. The

**(a)**

**EMP_PROJ**

| | | Projs | |
|---|---|---|---|
| Ssn | Ename | Pnumber | Hours |

**(b)**

**EMP_PROJ**

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

**(c)**

**EMP_PROJ1**

| Ssn | Ename |
|---|---|

**EMP_PROJ2**

| Ssn | Pnumber | Hours |
|---|---|---|

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

existence of more than one multivalued attribute in one relation must be handled carefully. As an example, consider the following non-1NF relation:

PERSON (Ss#, {Car_lic#}, {Phone#})

This relation represents the fact that a person has multiple cars and multiple phones. If strategy 2 above is followed, it results in an all-key relation:

PERSON_IN_1NF (Ss#, Car_lic#, Phone#)

To avoid introducing any extraneous relationship between Car_lic# and Phone#, all possible combinations of values are represented for every Ss#, giving rise to redundancy. This leads to the problems handled by multivalued dependencies and 4NF, which we will discuss in Section 15.6. The right way to deal with the two multivalued attributes in PERSON shown previously is to decompose it into two separate relations, using strategy 1 discussed above: P1(Ss#, Car_lic#) and P2(Ss#, Phone#).

## 15.3.5 Second Normal Form

**Second normal form (2NF)** is based on the concept of *full functional dependency.* A functional dependency $X \rightarrow Y$ is a **full functional dependency** if removal of any attribute $A$ from $X$ means that the dependency does not hold any more; that is, for any attribute $A \varepsilon X$, $(X - \{A\})$ does *not* functionally determine $Y$. A functional dependency $X \rightarrow Y$ is a **partial dependency** if some attribute $A \varepsilon X$ can be removed from $X$ and the dependency still holds; that is, for some $A \varepsilon X$, $(X - \{A\}) \rightarrow Y$. In Figure 15.3(b), {Ssn, Pnumber} → Hours is a full dependency (neither Ssn → Hours nor Pnumber → Hours holds). However, the dependency {Ssn, Pnumber} → Ename is partial because Ssn → Ename holds.

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

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

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 15.3(b) lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure 15.11(a), each of which is in 2NF.

## 15.3.6 Third Normal Form

**Third normal form (3NF)** is based on the concept of *transitive dependency*. A functional dependency $X \rightarrow Y$ in a relation schema $R$ is a **transitive dependency** if there exists a set of attributes $Z$ in $R$ that is neither a candidate key nor a subset of any key of $R$,[10] and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold. The dependency Ssn → Dmgr_ssn is transitive through Dnumber in EMP_DEPT in Figure 15.3(a), because both the

[10] This is the general definition of transitive dependency. Because we are concerned only with primary keys in this section, we allow transitive dependencies where $X$ is the primary key but $Z$ may be (a subset of) a candidate key.

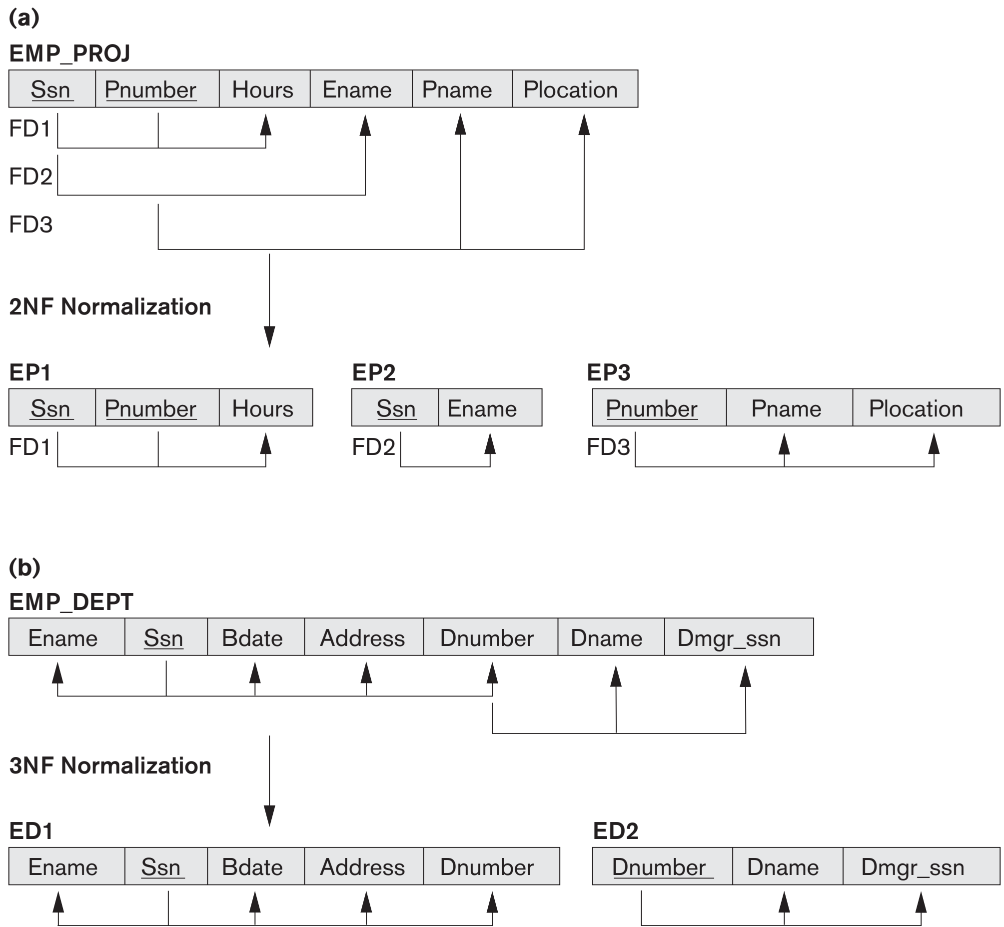



**Figure 15.11**
Normalizing into 2NF and 3NF. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.

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 15.3(a) 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 15.11(b). Intuitively, we see that ED1 and ED2 represent independent entity facts about employees and departments. A NATURAL JOIN operation on ED1 and ED2 will recover the original relation EMP_DEPT without generating spurious tuples.

Intuitively, we can see that any functional dependency in which the left-hand side is part (a proper subset) of the primary key, or any functional dependency in which the left-hand side is a nonkey attribute, is a *problematic* FD. 2NF and 3NF normalization remove these problem FDs by decomposing the original relation into new relations. In terms of the normalization process, it is not necessary to remove the partial dependencies before the transitive dependencies, but historically, 3NF has been defined with the assumption that a relation is tested for 2NF first before it is tested for 3NF. Table 15.1 informally summarizes the three normal forms based on primary keys, the tests used in each case, and the corresponding *remedy* or normalization performed to achieve the normal form.

## 15.4 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 discussed in Section 15.1.2. 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

**Table 15.1** Summary of Normal Forms Based on Primary Keys and Corresponding Normalization

| Normal Form | Test | Remedy (Normalization) |
|---|---|---|
| First (1NF) | Relation should have no multivalued attributes or nested relations. | Form new relations for each multivalued attribute or nested relation. |
| Second (2NF) | For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key. | Decompose and set up a new relation for each partial key with its dependent attribute(s). Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it. |
| Third (3NF) | Relation should not have a nonkey attribute functionally determined by another nonkey attribute (or by a set of nonkey attributes). That is, there should be no transitive dependency of a nonkey attribute on the primary key. | Decompose and set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s). |

any, into account. In this section we give the more general definitions of 2NF and 3NF that take *all* candidate keys of a relation into account. Notice that this does not affect the definition of 1NF since it is independent of keys and functional dependencies. 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.

### 15.4.1 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*.[11]

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are *part of* the primary key. If the primary key contains a single attribute, the test need not be applied at all. Consider the relation schema LOTS shown in Figure 15.12(a), which describes parcels of land for sale in various counties of a state. Suppose that there are two candidate keys: Property_id# and {County_name, Lot#}; that is, lot numbers are unique only within each county, but Property_id# numbers are unique across counties for the entire state.

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

FD3: County_name → Tax_rate
FD4: Area → Price

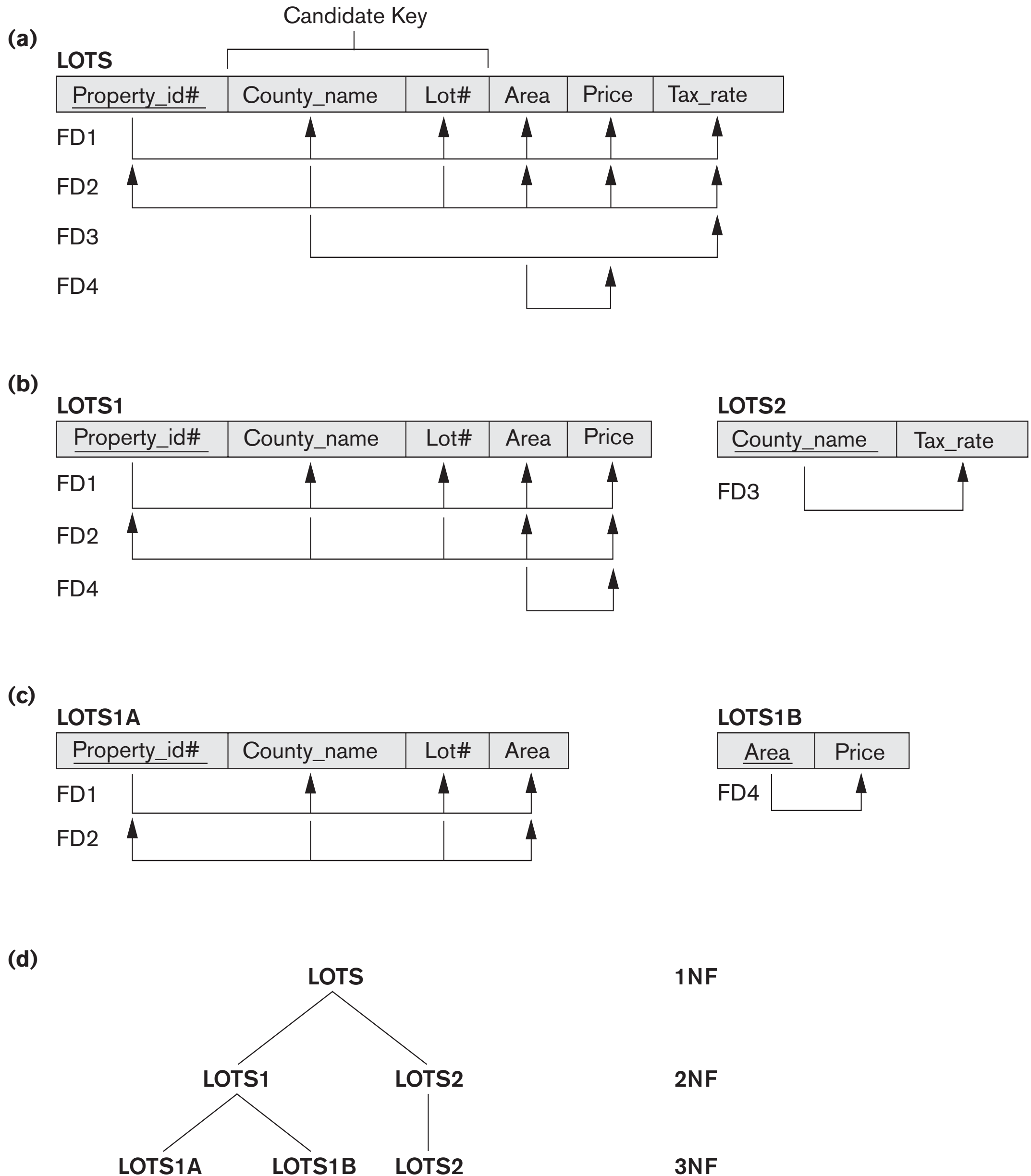
In words, the dependency FD3 says that the tax rate is fixed for a given county (does not vary lot by lot within the same county), while FD4 says that the price of a lot is determined by its area regardless of which county it is in. (Assume that this is the price of the lot for tax purposes.)

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 15.12(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.

---

[11]This definition can be restated as follows: A relation schema *R* is in 2NF if every nonprime attribute *A* in *R* is fully functionally dependent on *every* key of *R*.

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



### 15.4.2 General Definition of Third Normal Form

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

According to this definition, LOTS2 (Figure 15.12(b)) is in 3NF. However, FD4 in LOTS1 violates 3NF because Area is not a superkey and Price is not a prime attribute in LOTS1. To normalize LOTS1 into 3NF, we decompose it into the relation schemas LOTS1A and LOTS1B shown in Figure 15.12(c). We construct LOTS1A by removing the attribute Price that violates 3NF from LOTS1 and placing it with Area (the 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.
- This general definition can be applied *directly* to test whether a relation schema is in 3NF; it does *not* have to go through 2NF first. If we apply the above 3NF definition to LOTS with the dependencies FD1 through FD4, we find that *both* FD3 and FD4 violate 3NF. Therefore, we could decompose LOTS into LOTS1A, LOTS1B, and LOTS2 directly. Hence, the transitive and partial dependencies that violate 3NF can be removed *in any order*.

### 15.4.3 Interpreting the General Definition of Third Normal Form

A relation schema $R$ violates the general definition of 3NF if a functional dependency $X \rightarrow A$ 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:

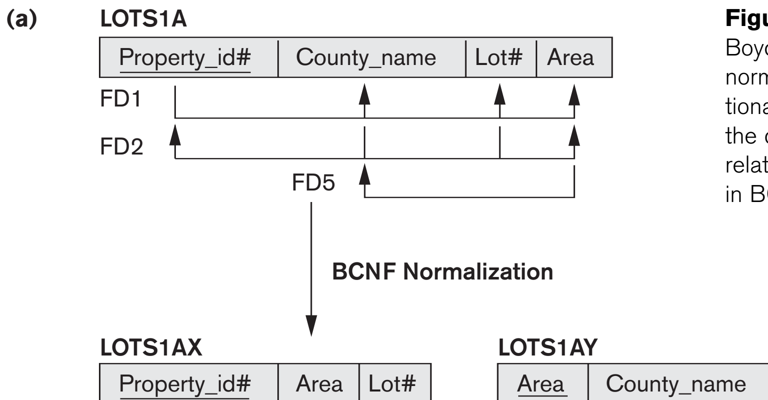
- It is fully functionally dependent on every key of $R$.
- It is nontransitively dependent on every key of $R$.

## 15.5 Boyce-Codd Normal Form

**Boyce-Codd normal form (BCNF)** was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF. That is, every relation in BCNF is also in 3NF; however, a relation in 3NF is *not necessarily* in BCNF. Intuitively, we can see the need for a stronger normal form than 3NF by going back to the LOTS relation schema in Figure 15.12(a) with its four functional dependencies FD1 through FD4. Suppose that we have thousands of lots in the relation but the lots are from only two counties: DeKalb and Fulton. Suppose also that lot sizes in DeKalb County are only 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0 acres, whereas lot sizes in Fulton County are restricted to 1.1, 1.2, ..., 1.9, and 2.0 acres. In such a situation we would have the additional functional dependency FD5: Area → County_name. If we add this to the other dependencies, the relation schema LOTS1A still is in 3NF because County_name is a prime attribute.

The area of a lot that determines the county, as specified by FD5, can be represented by 16 tuples in a separate relation *R*(Area, County_name), since there are only 16 possible Area values (see Figure 15.13). This representation reduces the redundancy of repeating the same information in the thousands of LOTS1A tuples. BCNF is a *stronger normal form* that would disallow LOTS1A and suggest the need for decomposing it.

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



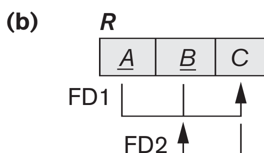




**Figure 15.13**
Boyce-Codd normal form. (a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition. (b) A schematic relation with FDs; it is in 3NF, but not in BCNF.

The formal definition of BCNF differs from the definition of 3NF in that condition (b) of 3NF, which allows $A$ to be prime, is absent from BCNF. That makes BCNF a stronger normal form compared to 3NF. In our example, FD5 violates BCNF in LOTS1A because AREA is not a superkey of LOTS1A. Note that FD5 satisfies 3NF in LOTS1A because County_name is a prime attribute (condition b), but this condition does not exist in the definition of BCNF. We can decompose LOTS1A into two BCNF relations LOTS1AX and LOTS1AY, shown in Figure 15.13(a). This decomposition loses the functional dependency FD2 because its attributes no longer coexist in the same relation after decomposition.

In practice, most relation schemas that are in 3NF are also in BCNF. Only if $X \rightarrow A$ holds in a relation schema $R$ with $X$ not being a superkey *and* $A$ being a prime attribute will $R$ be in 3NF but not in BCNF. The relation schema $R$ shown in Figure 15.13(b) illustrates the general case of such a relation. Ideally, relational database design should strive to achieve BCNF or 3NF for every relation schema. Achieving the normalization status of just 1NF or 2NF is not considered adequate, since they were developed historically as stepping stones to 3NF and BCNF.

As another example, consider Figure 15.14, which shows a relation TEACH with the following dependencies:

FD1: {Student, Course} $\rightarrow$ Instructor
FD2:[12] Instructor $\rightarrow$ Course

Note that {Student, Course} is a candidate key for this relation and that the dependencies shown follow the pattern in Figure 15.13(b), with Student as $A$, Course as $B$, and Instructor as $C$. Hence this relation is in 3NF but not BCNF. Decomposition of this relation schema into two schemas is not straightforward because it may be

**Figure 15.14**
A relation TEACH that is in 3NF but not BCNF.

**TEACH**

| Student | Course | Instructor |
|---|---|---|
| Narayan | Database | Mark |
| Smith | Database | Navathe |
| Smith | Operating Systems | Ammar |
| Smith | Theory | Schulman |
| Wallace | Database | Mark |
| Wallace | Operating Systems | Ahamad |
| Wong | Database | Omiecinski |
| Zelaya | Database | Navathe |
| Narayan | Operating Systems | Ammar |

[12]This dependency means that *each instructor teaches one course* is a constraint for this application.

decomposed into one of the three following possible pairs:

1. {Student, Instructor} and {Student, Course}.
2. {Course, Instructor} and {Course, Student}.
3. {Instructor, Course} and {Instructor, Student}.

All three decompositions *lose* the functional dependency FD1. The *desirable decomposition* of those just shown is 3 because it will not generate spurious tuples after a join.

A test to determine whether a decomposition is nonadditive (or lossless) is discussed in Section 16.2.4 under Property NJB. In general, a relation not in BCNF should be decomposed so as to meet this property.

We make sure that we meet this property, because nonadditive decomposition is a must during normalization. We may have to possibly forgo the preservation of all functional dependencies in the decomposed relations, as is the case in this example. Algorithm 16.5 does that and could be used above to give decomposition 3 for TEACH, which yields two relations in BCNF as:

(Instructor, Course) and (Instructor, Student)

Note that if we designate (Student, Instructor) as a primary key of the relation TEACH, the FD Instructor → Course causes a partial (non-full-functional) dependency of Course on a part of this key. This FD may be removed as a part of second normalization yielding exactly the same two relations in the result. This is an example of a case where we may reach the same ultimate BCNF design via alternate paths of normalization.

## 15.6 Multivalued Dependency and Fourth Normal Form

So far we have discussed the concept of functional dependency, which is by far the most important type of dependency in relational database design theory, and normal forms based on functional dependencies. However, in many cases relations have constraints that cannot be specified as functional dependencies. In this section, we discuss the concept of *multivalued dependency* (MVD) and define *fourth normal form,* which is based on this dependency. A more formal discussion of MVDs and their properties is deferred to Chapter 16. Multivalued dependencies are a consequence of first normal form (1NF) (see Section 15.3.4), which disallows an attribute in a tuple to have a *set of values*, and the accompanying process of converting an unnormalized relation into 1NF. If we have two or more multivalued *independent* attributes in the same relation schema, we get into a problem of having to repeat every value of one of the attributes with every value of the other attribute to keep the relation state consistent and to maintain the independence among the attributes involved. This constraint is specified by a multivalued dependency.

For example, consider the relation EMP shown in Figure 15.15(a). A tuple in this EMP relation represents the fact that an employee whose name is Ename works on the project whose name is Pname and has a dependent whose name is Dname. An employee may work on several projects and may have several dependents, and the employee's projects and dependents are independent of one another.[13] To keep the relation state consistent, and to avoid any spurious relationship between the two independent attributes, we must have a separate tuple to represent every combination of an employee's dependent and an employee's project. This constraint is spec-

**Figure 15.15**
Fourth and fifth normal forms.
(a) The EMP relation with two MVDs: Ename →→ Pname and Ename →→ Dname.
(b) Decomposing the EMP relation into two 4NF relations EMP_PROJECTS and EMP_DEPENDENTS.
(c) The relation SUPPLY with no MVDs is in 4NF but not in 5NF if it has the JD($R_1$, $R_2$, $R_3$).
(d) Decomposing the relation SUPPLY into the 5NF relations $R_1$, $R_2$, $R_3$.

**(a) EMP**

| Ename | Pname | Dname |
|---|---|---|
| Smith | X | John |
| Smith | Y | Anna |
| Smith | X | Anna |
| Smith | Y | John |

**(b) EMP_PROJECTS**

| Ename | Pname |
|---|---|
| Smith | X |
| Smith | Y |

**EMP_DEPENDENTS**

| Ename | Dname |
|---|---|
| Smith | John |
| Smith | Anna |

**(c) SUPPLY**

| Sname | Part_name | Proj_name |
|---|---|---|
| Smith | Bolt | ProjX |
| Smith | Nut | ProjY |
| Adamsky | Bolt | ProjY |
| Walton | Nut | ProjZ |
| Adamsky | Nail | ProjX |
| Adamsky | Bolt | ProjX |
| Smith | Bolt | ProjY |

**(d) $R_1$**

| Sname | Part_name |
|---|---|
| Smith | Bolt |
| Smith | Nut |
| Adamsky | Bolt |
| Walton | Nut |
| Adamsky | Nail |

**$R_2$**

| Sname | Proj_name |
|---|---|
| Smith | ProjX |
| Smith | ProjY |
| Adamsky | ProjY |
| Walton | ProjZ |
| Adamsky | ProjX |

**$R_3$**

| Part_name | Proj_name |
|---|---|
| Bolt | ProjX |
| Nut | ProjY |
| Bolt | ProjY |
| Nut | ProjZ |
| Nail | ProjX |

[13]In an ER diagram, each would be represented as a multivalued attribute or as a weak entity type (see Chapter 7).

ified as a multivalued dependency on the EMP relation, which we define in this section. Informally, whenever two *independent* 1:N relationships *A*:*B* and *A*:*C* are mixed in the same relation, *R*(*A*, *B*, *C*), an MVD may arise.[14]

### 15.6.1 Formal Definition of Multivalued Dependency

**Definition.** A multivalued dependency $X \twoheadrightarrow Y$ specified on relation schema *R*, where *X* and *Y* are both subsets of *R*, specifies the following constraint on any relation state *r* of *R*: If two tuples $t_1$ and $t_2$ exist in *r* such that $t_1[X] = t_2[X]$, then two tuples $t_3$ and $t_4$ should also exist in *r* with the following properties,[15] where we use *Z* to denote $(R - (X \cup Y))$:[16]

- $t_3[X] = t_4[X] = t_1[X] = t_2[X]$.
- $t_3[Y] = t_1[Y]$ and $t_4[Y] = t_2[Y]$.
- $t_3[Z] = t_2[Z]$ and $t_4[Z] = t_1[Z]$.

Whenever $X \twoheadrightarrow Y$ holds, we say that *X* **multidetermines** *Y*. Because of the symmetry in the definition, whenever $X \twoheadrightarrow Y$ holds in *R*, so does $X \twoheadrightarrow Z$. Hence, $X \twoheadrightarrow Y$ implies $X \twoheadrightarrow Z$, and therefore it is sometimes written as $X \twoheadrightarrow Y|Z$.

An MVD $X \twoheadrightarrow Y$ in R is called a **trivial MVD** if (a) *Y* is a subset of *X*, or (b) $X \cup Y = R$. For example, the relation EMP_PROJECTS in Figure 15.15(b) has the trivial MVD Ename $\twoheadrightarrow$ Pname. An MVD that satisfies neither (a) nor (b) is called a **nontrivial MVD**. A trivial MVD will hold in *any* relation state *r* of *R*; it is called trivial because it does not specify any significant or meaningful constraint on *R*.

If we have a *nontrivial MVD* in a relation, we may have to repeat values redundantly in the tuples. In the EMP relation of Figure 15.15(a), the values 'X' and 'Y' of Pname are repeated with each value of Dname (or, by symmetry, the values 'John' and 'Anna' of Dname are repeated with each value of Pname). This redundancy is clearly undesirable. However, the EMP schema is in BCNF because *no* functional dependencies hold in EMP. Therefore, we need to define a fourth normal form that is stronger than BCNF and disallows relation schemas such as EMP. Notice that relations containing nontrivial MVDs tend to be **all-key relations**—that is, their key is all their attributes taken together. Furthermore, it is rare that such all-key relations with a combinatorial occurrence of repeated values would be designed in practice. However, recognition of MVDs as a potential problematic dependency is essential in relational design.

We now present the definition of **fourth normal form (4NF)**, which is violated when a relation has undesirable multivalued dependencies, and hence can be used to identify and decompose such relations.

[14]This MVD is denoted as $A \twoheadrightarrow B|C$.

[15]The tuples $t_1$, $t_2$, $t_3$, and $t_4$ are not necessarily distinct.

[16]*Z* is shorthand for the attributes in *R* after the attributes in $(X \cup Y)$ are removed from *R*.

**Definition.** A relation schema $R$ is in 4**NF** with respect to a set of dependencies $F$ (that includes functional dependencies and multivalued dependencies) if, for every *nontrivial* multivalued dependency $X \twoheadrightarrow Y$ in $F^{+}$[17] $X$ is a superkey for $R$.

We can state the following points:

- An all-key relation is always in BCNF since it has no FDs.
- An all-key relation such as the EMP relation in Figure 15.15(a), which has no FDs but has the MVD Ename $\twoheadrightarrow$ Pname | Dname, is not in 4NF.
- A relation that is not in 4NF due to a nontrivial MVD must be decomposed to convert it into a set of relations in 4NF.
- The decomposition removes the redundancy caused by the MVD.

The process of normalizing a relation involving the nontrivial MVDs that is not in 4NF consists of decomposing it so that each MVD is represented by a separate relation where it becomes a trivial MVD. Consider the EMP relation in Figure 15.15(a). EMP is not in 4NF because in the nontrivial MVDs Ename $\twoheadrightarrow$ Pname and Ename $\twoheadrightarrow$ Dname, and Ename is not a superkey of EMP. We decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS, shown in Figure 15.15(b). Both EMP_PROJECTS and EMP_DEPENDENTS are in 4NF, because the MVDs Ename $\twoheadrightarrow$ Pname in EMP_PROJECTS and Ename $\twoheadrightarrow$ Dname in EMP_DEPENDENTS are trivial MVDs. No other nontrivial MVDs hold in either EMP_PROJECTS or EMP_DEPENDENTS. No FDs hold in these relation schemas either.

## 15.7 Join Dependencies and Fifth Normal Form

In our discussion so far, we have pointed out the problematic functional dependencies and showed how they were eliminated by a process of repeated binary decomposition to remove them during the process of normalization to achieve 1NF, 2NF, 3NF and BCNF. These binary decompositions must obey the NJB property from Section 16.2.4 that we referenced while discussing the decomposition to achieve BCNF. Achieving 4NF typically involves eliminating MVDs by repeated binary decompositions as well. However, in some cases there may be no nonadditive join decomposition of $R$ into *two* relation schemas, but there may be a nonadditive join decomposition into *more than two* relation schemas. Moreover, there may be no functional dependency in $R$ that violates any normal form up to BCNF, and there may be no nontrivial MVD present in $R$ either that violates 4NF. We then resort to another dependency called the *join dependency* and, if it is present, carry out a *multiway decomposition* into fifth normal form (5NF). It is important to note that such a dependency is a very peculiar semantic constraint that is very difficult to detect in practice; therefore, normalization into 5NF is very rarely done in practice.

---

[17] $F^{+}$ refers to the cover of functional dependencies $F$, or all dependencies that are implied by $F$. This is defined in Section 16.1.

**Definition.** A **join dependency** (**JD**), denoted by JD($R_1$, $R_2$, ..., $R_n$), specified on relation schema $R$, specifies a constraint on the states $r$ of $R$. The constraint states that every legal state $r$ of $R$ should have a nonadditive join decomposition into $R_1$, $R_2$, ..., $R_n$. Hence, for every such $r$ we have

$$* (\pi_{R_1}(r), \pi_{R_2}(r), ..., \pi_{R_n}(r)) = r$$

Notice that an MVD is a special case of a JD where $n = 2$. That is, a JD denoted as JD($R_1$, $R_2$) implies an MVD $(R_1 \cap R_2) \twoheadrightarrow (R_1 - R_2)$ (or, by symmetry, $(R_1 \cap R_2) \twoheadrightarrow (R_2 - R_1)$). A join dependency JD($R_1$, $R_2$, ..., $R_n$), specified on relation schema $R$, is a **trivial** JD if one of the relation schemas $R_i$ in JD($R_1$, $R_2$, ..., $R_n$) is equal to $R$. Such a dependency is called trivial because it has the nonadditive join property for any relation state $r$ of $R$ and thus does not specify any constraint on $R$. We can now define fifth normal form, which is also called *project-join normal form.*

**Definition.** A relation schema $R$ is in **fifth normal form (5NF)** (or **project-join normal form (PJNF)**) with respect to a set $F$ of functional, multivalued, and join dependencies if, for every nontrivial join dependency JD($R_1$, $R_2$, ..., $R_n$) in $F^+$ (that is, implied by $F$),[18] every $R_i$ is a superkey of $R$.

For an example of a JD, consider once again the SUPPLY all-key relation in Figure 15.15(c). Suppose that the following additional constraint always holds: Whenever a supplier $s$ supplies part $p$, *and* a project $j$ uses part $p$, *and* the supplier $s$ supplies *at least one* part to project $j$, *then* supplier $s$ will also be supplying part $p$ to project $j$. This constraint can be restated in other ways and specifies a join dependency JD($R_1$, $R_2$, $R_3$) among the three projections $R_1$(Sname, Part_name), $R_2$(Sname, Proj_name), and $R_3$(Part_name, Proj_name) of SUPPLY. If this constraint holds, the tuples below the dashed line in Figure 15.15(c) must exist in any legal state of the SUPPLY relation that also contains the tuples above the dashed line. Figure 15.15(d) shows how the SUPPLY relation *with the join dependency* is decomposed into three relations $R_1$, $R_2$, and $R_3$ that are each in 5NF. Notice that applying a natural join to *any two* of these relations *produces spurious tuples,* but applying a natural join to *all three together* does not. The reader should verify this on the sample relation in Figure 15.15(c) and its projections in Figure 15.15(d). This is because only the JD exists, but no MVDs are specified. Notice, too, that the JD($R_1$, $R_2$, $R_3$) is specified on *all* legal relation states, not just on the one shown in Figure 15.15(c).

Discovering JDs in practical databases with hundreds of attributes is next to impossible. It can be done only with a great degree of intuition about the data on the part of the designer. Therefore, the current practice of database design pays scant attention to them.

## 15.8 Summary

In this chapter we discussed several pitfalls in relational database design using intuitive arguments. We identified informally some of the measures for indicating

[18]Again, $F^+$ refers to the cover of functional dependencies $F$, or all dependencies that are implied by $F$. This is defined in Section 16.1.

whether a relation schema is *good* or *bad*, and provided informal guidelines for a good design. These guidelines are based on doing a careful conceptual design in the ER and EER model, following the mapping procedure in Chapter 9 correctly to map entities and relationships into relations. Proper enforcement of these guidelines and lack of redundancy will avoid the insertion/deletion/update anomalies, and generation of spurious data. We recommended limiting NULL values, which cause problems during SELECT, JOIN, and aggregation operations. Then we presented some formal concepts that allow us to do relational design in a top-down fashion by analyzing relations individually. We defined this process of design by analysis and decomposition by introducing the process of normalization.

We defined the concept of functional dependency, which is the basic tool for analyzing relational schemas, and discussed some of its properties. Functional dependencies specify semantic constraints among the attributes of a relation schema. Next we described the normalization process for achieving good designs by testing relations for undesirable types of *problematic* functional dependencies. We provided a treatment of successive normalization based on a predefined primary key in each relation, and then relaxed this requirement and provided more general definitions of second normal form (2NF) and third normal form (3NF) that take all candidate keys of a relation into account. We presented examples to illustrate how by using the general definition of 3NF a given relation may be analyzed and decomposed to eventually yield a set of relations in 3NF.

We presented Boyce-Codd normal form (BCNF) and discussed how it is a stronger form of 3NF. We also illustrated how the decomposition of a non-BCNF relation must be done by considering the nonadditive decomposition requirement. Then we introduced the fourth normal form based on multivalued dependencies that typically arise due to mixing independent multivalued attributes into a single relation. Finally, we introduced the fifth normal form, which is based on join dependency, and which identifies a peculiar constraint that causes a relation to be decomposed into several components so that they always yield the original relation back after a join. In practice, most commercial designs have followed the normal forms up to BCNF. Need for decomposing into 5NF rarely arises in practice, and join dependencies are difficult to detect for most practical situations, making 5NF more of theoretical value.

Chapter 16 presents synthesis as well as decomposition algorithms for relational database design based on functional dependencies. Related to decomposition, we discuss the concepts of *nonadditive* (or *lossless*) *join* and *dependency preservation,* which are enforced by some of these algorithms. Other topics in Chapter 16 include a more detailed treatment of functional and multivalued dependencies, and other types of dependencies.

## Review Questions

**15.1.** Discuss attribute semantics as an informal measure of goodness for a relation schema.

**15.2.** Discuss insertion, deletion, and modification anomalies. Why are they considered bad? Illustrate with examples.

**15.3.** Why should NULLs in a relation be avoided as much as possible? Discuss the problem of spurious tuples and how we may prevent it.

**15.4.** State the informal guidelines for relation schema design that we discussed. Illustrate how violation of these guidelines may be harmful.

**15.5.** What is a functional dependency? What are the possible sources of the information that defines the functional dependencies that hold among the attributes of a relation schema?

**15.6.** Why can we not infer a functional dependency automatically from a particular relation state?

**15.7.** What does the term *unnormalized relation* refer to? How did the normal forms develop historically from first normal form up to Boyce-Codd normal form?

**15.8.** Define first, second, and third normal forms when only primary keys are considered. How do the general definitions of 2NF and 3NF, which consider all keys of a relation, differ from those that consider only primary keys?

**15.9.** What undesirable dependencies are avoided when a relation is in 2NF?

**15.10.** What undesirable dependencies are avoided when a relation is in 3NF?

**15.11.** In what way do the generalized definitions of 2NF and 3NF extend the definitions beyond primary keys?

**15.12.** Define Boyce-Codd normal form. How does it differ from 3NF? Why is it considered a stronger form of 3NF?

**15.13.** What is multivalued dependency? When does it arise?

**15.14.** Does a relation with two or more columns always have an MVD? Show with an example.

**15.15.** Define fourth normal form. When is it violated? When is it typically applicable?

**15.16.** Define join dependency and fifth normal form.

**15.17.** Why is 5NF also called project-join normal form (PJNF)?

**15.18.** Why do practical database designs typically aim for BCNF and not aim for higher normal forms?

## Exercises

**15.19.** Suppose that we have the following requirements for a university database that is used to keep track of students' transcripts:

a. The university keeps track of each student's name (Sname), student number (Snum), Social Security number (Ssn), current address (Sc_addr) and

phone (Sc_phone), permanent address (Sp_addr) and phone (Sp_phone), birth date (Bdate), sex (Sex), class (Class) ('freshman', 'sophomore', ... , 'graduate'), major department (Major_code), minor department (Minor_code) (if any), and degree program (Prog) ('b.a.', 'b.s.', ... , 'ph.d.'). Both Ssn and student number have unique values for each student.

b. Each department is described by a name (Dname), department code (Dcode), office number (Doffice), office phone (Dphone), and college (Dcollege). Both name and code have unique values for each department.

c. Each course has a course name (Cname), description (Cdesc), course number (Cnum), number of semester hours (Credit), level (Level), and offering department (Cdept). The course number is unique for each course.

d. Each section has an instructor (Iname), semester (Semester), year (Year), course (Sec_course), and section number (Sec_num). The section number distinguishes different sections of the same course that are taught during the same semester/year; its values are 1, 2, 3, ..., up to the total number of sections taught during each semester.

e. A grade record refers to a student (Ssn), a particular section, and a grade (Grade).

Design a relational database schema for this database application. First show all the functional dependencies that should hold among the attributes. Then design relation schemas for the database that are each in 3NF or BCNF. Specify the key attributes of each relation. Note any unspecified requirements, and make appropriate assumptions to render the specification complete.

**15.20.** What update anomalies occur in the EMP_PROJ and EMP_DEPT relations of Figures 15.3 and 15.4?

**15.21.** In what normal form is the LOTS relation schema in Figure 15.12(a) with respect to the restrictive interpretations of normal form that take *only the primary key* into account? Would it be in the same normal form if the general definitions of normal form were used?

**15.22.** Prove that any relation schema with two attributes is in BCNF.

**15.23.** Why do spurious tuples occur in the result of joining the EMP_PROJ1 and EMP_ LOCS relations in Figure 15.5 (result shown in Figure 15.6)?

**15.24.** Consider the universal relation $R = \{A, B, C, D, E, F, G, H, I, J\}$ and the set of functional dependencies $F = \{ \{A, B\} \rightarrow \{C\}, \{A\} \rightarrow \{D, E\}, \{B\} \rightarrow \{F\}, \{F\} \rightarrow \{G, H\}, \{D\} \rightarrow \{I, J\} \}$. What is the key for $R$? Decompose $R$ into 2NF and then 3NF relations.

**15.25.** Repeat Exercise 15.24 for the following different set of functional dependencies $G = \{\{A, B\} \rightarrow \{C\}, \{B, D\} \rightarrow \{E, F\}, \{A, D\} \rightarrow \{G, H\}, \{A\} \rightarrow \{I\}, \{H\} \rightarrow \{J\} \}$.

**15.26.** Consider the following relation:

| A | B | C | TUPLE# |
|---|---|---|---|
| 10 | b1 | c1 | 1 |
| 10 | b2 | c2 | 2 |
| 11 | b4 | c1 | 3 |
| 12 | b3 | c4 | 4 |
| 13 | b1 | c1 | 5 |
| 14 | b3 | c4 | 6 |

a. Given the previous extension (state), which of the following dependencies *may hold* in the above relation? If the dependency cannot hold, explain why *by specifying the tuples that cause the violation*.

i. $A \rightarrow B$, ii. $B \rightarrow C$, iii. $C \rightarrow B$, iv. $B \rightarrow A$, v. $C \rightarrow A$

b. Does the above relation have a potential candidate key? If it does, what is it? If it does not, why not?

**15.27.** Consider a relation $R(A, B, C, D, E)$ with the following dependencies:

$AB \rightarrow C$, $CD \rightarrow E$, $DE \rightarrow B$

Is $AB$ a candidate key of this relation? If not, is $ABD$? Explain your answer.

**15.28.** Consider the relation $R$, which has attributes that hold schedules of courses and sections at a university; $R$ = {Course_no, Sec_no, Offering_dept, Credit_hours, Course_level, Instructor_ssn, Semester, Year, Days_hours, Room_no, No_of_students}. Suppose that the following functional dependencies hold on $R$:

{Course_no} → {Offering_dept, Credit_hours, Course_level}
{Course_no, Sec_no, Semester, Year} → {Days_hours, Room_no, No_of_students, Instructor_ssn}
{Room_no, Days_hours, Semester, Year} → {Instructor_ssn, Course_no, Sec_no}

Try to determine which sets of attributes form keys of $R$. How would you normalize this relation?

**15.29.** Consider the following relations for an order-processing application database at ABC, Inc.

ORDER (<u>O#</u>, Odate, Cust#, Total_amount)
ORDER_ITEM(<u>O#</u>, <u>I#</u>, Qty_ordered, Total_price, Discount%)

Assume that each item has a different discount. The Total_price refers to one item, Odate is the date on which the order was placed, and the Total_amount is the amount of the order. If we apply a natural join on the relations ORDER_ITEM and ORDER in this database, what does the resulting relation schema look like? What will be its key? Show the FDs in this resulting relation. Is it in 2NF? Is it in 3NF? Why or why not? (State assumptions, if you make any.)

**15.30.** Consider the following relation:

CAR_SALE(Car#, Date_sold, Salesperson#, Commission%, Discount_amt)

Assume that a car may be sold by multiple salespeople, and hence {Car#, Salesperson#} is the primary key. Additional dependencies are

Date_sold → Discount_amt and
Salesperson# → Commission%

Based on the given primary key, is this relation in 1NF, 2NF, or 3NF? Why or why not? How would you successively normalize it completely?

**15.31.** Consider the following relation for published books:

BOOK (Book_title, Author_name, Book_type, List_price, Author_affil, Publisher)

Author_affil refers to the affiliation of author. Suppose the following dependencies exist:

Book_title → Publisher, Book_type
Book_type → List_price
Author_name → Author_affil

a. What normal form is the relation in? Explain your answer.

b. Apply normalization until you cannot decompose the relations further. State the reasons behind each decomposition.

**15.32.** This exercise asks you to convert business statements into dependencies. Consider the relation DISK_DRIVE (Serial_number, Manufacturer, Model, Batch, Capacity, Retailer). Each tuple in the relation DISK_DRIVE contains information about a disk drive with a unique Serial_number, made by a manufacturer, with a particular model number, released in a certain batch, which has a certain storage capacity and is sold by a certain retailer. For example, the tuple Disk_drive ('1978619', 'WesternDigital', 'A2235X', '765234', 500, 'CompUSA') specifies that WesternDigital made a disk drive with serial number 1978619 and model number A2235X, released in batch 765234; it is 500GB and sold by CompUSA.

Write each of the following dependencies as an FD:

a. The manufacturer and serial number uniquely identifies the drive.

b. A model number is registered by a manufacturer and therefore can't be used by another manufacturer.

c. All disk drives in a particular batch are the same model.

d. All disk drives of a certain model of a particular manufacturer have exactly the same capacity.

**15.33.** Consider the following relation:

*R* (Doctor#, Patient#, Date, Diagnosis, Treat_code, Charge)

In the above relation, a tuple describes a visit of a patient to a doctor along with a treatment code and daily charge. Assume that diagnosis is determined (uniquely) for each patient by a doctor. Assume that each treatment code has a fixed charge (regardless of patient). Is this relation in 2NF? Justify your answer and decompose if necessary. Then argue whether further normalization to 3NF is necessary, and if so, perform it.

**15.34.** Consider the following relation:

CAR_SALE (Car_id, Option_type, Option_listprice, Sale_date, Option_discountedprice)

This relation refers to options installed in cars (e.g., cruise control) that were sold at a dealership, and the list and discounted prices of the options.

If CarID $\rightarrow$ Sale_date and Option_type $\rightarrow$ Option_listprice and CarID, Option_type $\rightarrow$ Option_discountedprice, argue using the generalized definition of the 3NF that this relation is not in 3NF. Then argue from your knowledge of 2NF, why it is not even in 2NF.

**15.35.** Consider the relation:

BOOK (Book_Name, Author, Edition, Year)

with the data:

| Book_Name | Author | Edition | Copyright_Year |
|---|---|---|---|
| DB_fundamentals | Navathe | 4 | 2004 |
| DB_fundamentals | Elmasri | 4 | 2004 |
| DB_fundamentals | Elmasri | 5 | 2007 |
| DB_fundamentals | Navathe | 5 | 2007 |

a. Based on a common-sense understanding of the above data, what are the possible candidate keys of this relation?

b. Justify that this relation has the MVD { Book } $\twoheadrightarrow$ { Author } | { Edition, Year }.

c. What would be the decomposition of this relation based on the above MVD? Evaluate each resulting relation for the highest normal form it possesses.

**15.36.** Consider the following relation:

TRIP (Trip_id, Start_date, Cities_visited, Cards_used)

This relation refers to business trips made by company salespeople. Suppose the TRIP has a single Start_date, but involves many Cities and salespeople may use multiple credit cards on the trip. Make up a mock-up population of the table.

a. Discuss what FDs and/or MVDs exist in this relation.

b. Show how you will go about normalizing it.

## Laboratory Exercise

*Note*: The following exercise use the DBD (Data Base Designer) system that is described in the laboratory manual. The relational schema $R$ and set of functional dependencies $F$ need to be coded as lists. As an example, $R$ and $F$ for this problem is coded as:

$R = [a, b, c, d, e, f, g, h, i, j]$
$F = [[[a, b],[c]],$
$[[a],[d, e]],$
$[[b],[f]],$
$[[f],[g, h]],$
$[[d],[i, j]]]$

Since DBD is implemented in Prolog, use of uppercase terms is reserved for variables in the language and therefore lowercase constants are used to code the attributes. For further details on using the DBD system, please refer to the laboratory manual.

**15.37.** Using the DBD system, verify your answers to the following exercises:

a. 15.24 (3NF only)

b. 15.25

c. 15.27

d. 15.28

## Selected Bibliography

Functional dependencies were originally introduced by Codd (1970). The original definitions of first, second, and third normal form were also defined in Codd (1972a), where a discussion on update anomalies can be found. Boyce-Codd normal form was defined in Codd (1974). The alternative definition of third normal form is given in Ullman (1988), as is the definition of BCNF that we give here. Ullman (1988), Maier (1983), and Atzeni and De Antonellis (1993) contain many of the theorems and proofs concerning functional dependencies.

Additional references to relational design theory are given in Chapter 16.

ACHARYA INSTITUTE OF GRADUATE STUDIES

(NAAC Re-Accredited and Affiliated to Bangalore City University)

Soldevanahalli, Bengaluru-560107

UNIT-4

# Relational Database Language

## INTRODUCTION

Most of the problems faced at the time of implementation of any system are outcome of a poor database design. In many cases it happens that system has to be continuously modified in multiple respects due to changing requirements of users. It is very important that a proper planning has to be done.
A relation in a relational database is based on a relational schema, which consists of number of attributes.
A relational database is made up of a number of relations and corresponding relational database schema.
The goal of a relational database design is to generate a set of relation schema that allows us to store information without unnecessary redundancy and also to retrieve information easily.
One approach to design schemas that are in an appropriate normal form. The normal forms are used to ensure that various types of anomalies and inconsistencies are not introduced into the database.

## WHAT IS RDBMS?

RDBMS stands for Relational Database Management System. RDBMS data is structured in database tables, fields and records. Each RDBMS table consists of database table rows. Each database table row consists of one or more database table fields.
RDBMS store the data into collection of tables, which might be related by common fields (database table columns). RDBMS also provide relational operators to manipulate the data stored into the database tables. Most RDBMS use sql as database querylanguage.The most popular RDBMS are MS SQL Server, DB2, Oracle and MySQL.
The relational model is an example of record-based model. Record based models are so named because the database is structured in fixed format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record types. The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model.
The relational model was designed by the IBM research scientist and mathematician, Dr.E.F.Cod

The relational model originated from a paper authored by Dr.codd entitled "A Relational Model of Data for Large Shared Data Banks", written in 1970. This paper included the following concepts that apply to database management systems forrelational databases.

The relation is the only data structure used in the relational data model to represent both entities and relationships between them.

Tuples, Attributes and Domain.

Rows of the relation are referred to as **tuples** of the relation and columns are its **attributes**. Each attribute of the column are drawn from the set of values known as **domain**. The domain of an attribute contains the set of values that the attribute may assume.
From the historical perspective, the relational data model is relatively new .The first database systems were based on either network or hierarchical models .The relational data model has established itself as the primary data model for commercial data processing applications. Its success in this domain has led to its applications outside data processing in systems for computer aided design and other environments.

**DIFFERENCE BETWEEN DBMS & RDBMS**

A DBMS has to be persistent, that is it should be accessible when the programcreated the data ceases to exist or even the application that created the data restarted.A DBMS also has to provide some uniform methods independent of a specific application for accessing the information that is stored.
RDBMS is a Relational Data Base Management System Relational DBMS. This adds the additional condition that the system supports a tabular structure for the data, with enforced relationships between the tables. This excludes the databases that don't support a tabular structure or don't enforce relationships between tables. You can say DBMS does not impose any constraints or security with regard to data manipulation it is user or the programmer responsibility to ensure the ACID PROPERTY of the database whereas the RDBMS is more with this regard because RDBMS define the integrity constraint for the purpose of holding ACID PROPERTY.

The DBMS interfaces with application programs so that the data contained in the database can be used by multiple applications and users. The DBMS allows these users to access and manipulate the data contained in the database in a convenient and effective manner. In addition the DBMS exerts centralized control of the database, prevents unauthorized users from accessing the data and ensures privacy of data In this chapter we study the query language : Structured Query Language (SQL) which uses a combination of Relational algebra and Relational calculus.

It is a data sub language used to organize, manage and retrieve data fromrelational database, which is managed by Relational Database Management System (RDBMS).

Vendors of DBMS like Oracle, IBM, DB2, Sybase, and Ingress use SQL asprogramming language for their database.

SQL originated with the system R project in 1974 at IBM's San Jose ResearchCentre.

Original version of SQL was SEQUEL which was an Application Program Interface(API) to the system R project.

The predecessor of SEQUEL was named SQUARE.SQL-92 is the current standard and is the current version.The SQL language can be used in two ways :

- ✦ Interactively or
- ✦ Embedded inside another program

The SQL is used interactively to directly operate a database and produce the desired results. The user enters SQL command that is immediately executed. Most databases have a tool that allows interactive execution of the SQL language. These include SQL Base's SQL Talk, Oracle's SQL Plus, and Microsoft's SQL server 7 QueryAnalyzer.

The second way to execute a SQL command is by embedding it in another language such as Cobol, Pascal, BASIC, C, Visual Basic, Java, etc. The result of embedded SQL command is passed to the variables in the host program, which in turnwill deal with them. The combination of SQL with a fourth-generation language brings together the best of two worlds and allows creation of user interfaces and database access in one application.

**SUBDIVISIONS OF SQL**

Regardless of whether SQL is embedded or used interactively, it can be dividedinto three groups of commands, depending on their purpose.

- Data Definition Language (DDL).
- Data Manipulation Language (DML).
- Data Control Language (DCL).

**Data Definition Language :**

Data Definition Language is a part of SQL that is responsible for the creation, updation and deletion of tables. It is responsible for creation of views and indexes also.The list of DDL commands is given below :

- CREATE
- TABLEALTER
- TABLE DROP
- TABLE CREATE VIEW
- CREATE INDEX

**Data Manipulation Language :**

Data manipulation commands manipulate (insert, delete, update and retrieve)data. The DML language includes commands that run queries and changes in data. It includes the following commands :

- SELECT
- UPDATE
- DELETE
- INSERT

**Data Control Language :**

The commands that form data control language are related to the security of the database performing tasks of assigning privileges so users can access certain objects in the database.

The DCL commands are :

- GRANT
- REVOKE
- COMMIT
- ROLLBACK

**DATA DEFINITION LANGUAGE**

The SQL DDL provides commands for defining relation schemas, deletingrelations, creating indices, and modifying relation schemas.

The SQL DDL allows the specification of not only a set of relations but alsoinformation about each relation including :

- The schema for each relation.
- The domain of values associated with each attribute.
- The integrity constraints.
- The set of indices to be maintained for each relation.
- The security and authorization information for each relation.
- The physical storage structure of each relation on disk.

**Domain/Data Types in SQL :**

The SQL - 92 standard supports a variety of built-in domain types, including thefollowing :

- Numeric data types include
  - Integer numbers of various sizes
    INT or INTEGER
    SMALLINT
  - Real numbers of various precision
    REAL
    DOUBLE
    PRECISIONFLOAT
    (n)
  - Formatted numbers can be represented by using
    DECIMAL (i, j) or
    DEC (i, j)
    NUMERIC (i, j) or NUMBER (i, j)

where, i - the precision, is the total number of decimal digits andj - the scale,
is the number of digits after the decimal point.
The default for scale is zero and the default for precision is implementation defined.

(2) Character string data types - are either fixed - length or varying - length.

CHAR (n) or CHARACTER (n) - is fixed length character string with user specified length n.

VARCHAR (n) - is a variable length character string, with user - specified maximum length n. The full form of CHARACTER VARYING (n), is equivalent.

(3) Date and Time data types :

There are new data types for date and time in SQL-92.

DATE - It is a calendar date containing year, month and day typically in the form
yyyy : mm : dd

TIME - It is the time of day, in hours, minutes and seconds, typically in the form
HH : MM : SS.

Varying length character strings, date and time were not part of the SQL - 89 standard.

In this section we will study the three Data Definition Language Commands :

CREATE TABLE
ALTER TABLE
DROP TABLE

**1. CREATE TABLE Command :**

The CREATE TABLE COMMAND is used to specify a new relation by giving it aname and specifying its attributes and constraints.

The attributes are specified first, and each attribute is given a name, a data type tospecify 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.

**Syntax of create table command :** CREATE TABLE table_name (
Column_name 1 data type [NOT NULL],
:
:
Column_name n data_type [NOT NULL]);The variables are defined as follows
:
If NOT NULL is not specified, the column can have NULL values.
table_name - is the name for the table.
column_name 1 to column_name n - are the valid column names or attributes.
NOT NULL – It specifies that column is mandatory. This feature allows you to prevent data from being entered into table without certain columns having data in them.

**Examples of CREATE TABLE Command :**

(1) Create Table Employee

| | | |
|---|---|---|
| (E_name | varchar2 (20) | NOT NULL, |
| B_Date | Date, | |
| Salary | Decimal (10, 12) | |
| Address | Varchar2 (50); | |

(2) Create Table Student

| | |
|---|---|
| (Student_id | Varchar2 (20) Not Null, |
| Last_Name | Varchar2 (20) Not Null, |
| First_name | Varchar2 (20), |
| BDate | Date, |
| State | Varchar2 (20), |
| City Varchar2 (20)); | |

(3) Create Table Course

| | |
|---|---|
| (Course_id | Varchar2 (5), |
| Department_id | Varchar2 (20), |
| Title | Varchar2 (20), |
| Description | Varchar2 (20)); |

**Constraints in CREATE TABLE Command :**
CREATE TABLE Command lets you enforce several kinds of constraints on atable : primary key, foreign key and check condition, unique condition.
A constraint clause can constrain a single column or group of columns in a table.
There are two ways to specify constraints :

- As part of the column definition i.e. a *column constraint*.
- Or at the end of the create table command i.e. a *table constraint*.
- Clauses that constrain several columns are the table constraints.

**The Primary Key :**
A table's primary key is the set of columns that uniquely identifies each row in thetable. CREATE TABLE command specifies the primary key as follows :

create table table_name (
Column_name 1 data_type [not null],
:

:

Column_name n data type [NOT NULL],

[Constraint constraint_name]

[Primary key (Column_name A, Column_name B… Column_name X)]);

Variables are defined as follows :
table_name is the name for the table.
column_name 1 through column_name n are the valid column names
data_type is valid datatype
constraint which is optional
constraint_name identifies the primary
key
column_name A through column_name X are the table's columns that compose the primary key.

**Example :**

```
Create table Employee
    (E_name          Varchar2 (20),
    B_Date           Date,
    Salary           Decimal (10, 2),
    Address          Varchar2 (80),
    Constraint
                     PK_Employe
    ePrimary key (Ename));
Create table_student
    (Student_id      Varchar2 (20),
    Last_name        Varchar2 (20)      NOT NULL,
    First_name       Varchar2 (20),
    B_Date           Date,
    State            Varchar2 (20),
    City             Varchar2 (20),
    Constraint       PK_Student
    Primary key      Student_id));
Create Table_Course
    (Course_id       Varchar2 (5),
    Department_id    Varchar2 (20),
    Title            Varchar2 (20),
    Description      Varchar2 (20),
    Constraint       PK_Course
    Primary key (Course_id,
    Department_id));
```

**Note :** We do not specify NOT NULL constraint for those columns which form the primary key, since those are the mandatory columns by default. Primary keys are subject to several restrictions.

(i) A column that is a part of the primary key cannot be NULL.

(ii) A column that is defined as LONG, or LONG RAW (ORACLE data types)cannot be a part of primary key.

(iii) The maximum number of columns in the primary key is 16.

**Foreign Key :** A foreign key is a combination of columns with values based on the primary key values from another table. A foreign key constraint also known as a referential integrity constraint, specifies that the values of the foreign key correspond toactual values of primary key in other table.

Create table command specifies the foreign key as follows :

Create Table table_name

(Column_name 1 data type [NOT NULL],

primary key values from another table. A foreign key constraint also known as a referential integrity constraint, specifies that the values of the foreign key correspond toactual values of primary key in other table.

Create table command specifies the foreign key as follows :

Create Table table_name

(Column_name 1 data type [NOT NULL],

:

Column_name N data type [NOT NULL],[constraint constraint_name

Foreign key (column_name F1 … Column_name FN) references referenced-table (column_name P1, … column_name $P_N$)]);

table_name - is the name for the table.

Column_name 1 through column_name N are the valid columns.

constraint_name is the name given to foreign key.

referenced_table - is the name of the table referenced by the foreign key declaration.

column_name $F_1$ through column_name $F_N$ are the columns that compose the foreignkey.

Column_name $P_1$ through column_name $P_N$ are the columns that compose the primarykey in referenced-table.

**Examples :**

```
Create table_department
      (Department_id        Varchar2 (20),
      Department_name       Varchar2 (20),
      Constraint            PK_Department
      Primary key           (Department_id));
Create table_course
      (Course_id            Varchar2 (20),
      Department_id         Varchar2 (20),
      Title                 Varchar2 (20),
      Description           Varchar2 (20),
      Constraint            PK_course
      Primary key (Course_id,
      Department_id),Constraint       FK -
      course
Foreign key (Department_id) references Department (Department_id));
```

Thus, primary key of course table is (Course_id, Department_id).

The primary key of Department table is (Department_id).

Foreign key of course table is (Department_id) which references the department table.

When you define a foreign key, the DBMS verifies the following :

(1) A primary key has been defined for table referenced by the foreign key.

(2) The number of columns composing the foreign key matches the number of primary key columns in the referenced table.

(3) The datatype and width of each foreign key columns matches the datatype and width of each primary key column in the referenced table.

**Unique Constraint or Candidate key :**

A candidate key is a combination of one or more columns, the values of which uniquely identify each row of the table. Create table command specifies the unique constraint as follows :

```
CREATE  TABLE  table_name
      (column_name 1      data_type            [NOT NULL],
           :
           :
      column_name n       data_type            [NOT
      NULL],[constraint   constraint_name
      Unique  (Column_name A,……… Column_nameX)]);
```

**Example :**

```
Create table student
      (Student_id         Varchar2 (20),
      Last_name           Varchar2 (20),        NOT
      NULL,First_name     Varchar2 (20),        NOT
      NULL,BDate          Date,
      State               Varchar2 (20),
      City                Varchar2 (20),
      Constraint          UK-student
      Unique              (last_name, first_name),
```

```
Constraint      PK-student
Primary key     (Student_id));
```

A unique constraint is not a substitute for a primary key. Two differences between primary key and unique constraints are :

(1) A table can have only one primary key, but it can have many unique constraints.
(2) When a primary key is defined, the columns that compose the primary key are automatically mandatory. When a unique constraint is declared, the columns that compose the unique constraint are not automatically defined to be mandatory, you must also specify that the column is NOT NULL.

**Check Constraint :**

Using CHECK constraint SQL can specify the data validation for column during table creation. CHECK clause is a Boolean condition that is either TRUE or FALSE. If the condition evaluates to TRUE, the column value is accepted by database, if the condition evaluates to FALSE, database will return an error code.

The check constraint is declared in CREATE TABLE statement using the syntax :

Column_name datatype [constraint constraint_name] [CHECK (Condition)]

The variables are defined as follows :

Column_name - is the column name
data_type - is the column's data type
constraint_name - is the name given to check constraint condition is the legal SQL
Condition that returns a Boolean value.

**Examples :**

```
Create table_worker
    (NameVarchar2 (25)                    NOT
    NULL,AgeNumber      Constraint CK_worker
    CHECK (Age Between 18 AND 65) );
Create table_instructor
    (Instructor_id      Varchar2 (20),
    Department_id       Varchar2 (20)      NOT
                                           NULL,
    Name                Varchar2 (25),
    Position            Varchar2 (25)
    Constraint          CK_instructor
CHECK (Position in ('ASSISTANT PROFESSOR', 'ASSOCIATE PROFESSOR', 'PROFESSOR')),
    Address             Varchar2 (25),
    Constraint          PK_instructor
```

```
Create table student
    (Student_id       Varchar2 (20),
    Last_name         Varchar2 (20),        NOT
    NULL,First_name   Varchar2 (20),        NOT
    NULL,BDate        Date,
    State             Varchar2 (20),
    City              Varchar2 (20),
    Constraint        UK-student
    Unique            (last_name, first_name),
    Constraint        PK-student
    Primary key       (Student_id));
```

A unique constraint is not a substitute for a primary key. Two differences between primary key and unique constraints are :

(3) A table can have only one primary key, but it can have many unique constraints.

(4) When a primary key is defined, the columns that compose the primary key are automatically mandatory. When a unique constraint is declared, the columns that compose the unique constraint are not automatically defined to be mandatory, you must also specify that the column is NOT NULL.

**Check Constraint :**

Using CHECK constraint SQL can specify the data validation for column during table creation. CHECK clause is a Boolean condition that is either TRUE or FALSE. If the condition evaluates to TRUE, the column value is accepted by database, if the condition evaluates to FALSE, database will return an error code.

The check constraint is declared in CREATE TABLE statement using the syntax :

Column_name datatype [constraint constraint_name] [CHECK (Condition)]

The variables are defined as follows :

Column_name - is the column name
data_type - is the column's data type
constraint_name - is the name given to check constraint condition is the legal SQL
Condition that returns a Boolean value.

**Examples :**

```
Create table_worker
    (NameVarchar2 (25)                      NOT
    NULL,AgeNumber       Constraint CK_worker
    CHECK (Age Between 18 AND 65) );
Create table_instructor
    (Instructor_id    Varchar2 (20),
    Department_id     Varchar2 (20)         NOT
                                            NULL,
    Name              Varchar2 (25),
    Position          Varchar2 (25)
    Constraint        CK_instructor
CHECK (Position in ('ASSISTANT PROFESSOR', 'ASSOCIATE PROFESSOR', 'PROFESSOR')),
    Address           Varchar2 (25),
    Constraint        PK_instructor
```

```
Primary key        (Instructor_id));
```

If the position of the instructor is not one of the three legal values, DBMS willreturn an error code indicating that a check constraint has been violated.

More than one column can have check constraint.

```
Create table_Patient

    (Patient_id        Varchar2 (25)        Primary key,
        Body_Temp      Number (4, 1)
        Constraint     Patient_BT
        CHECK (Body_Temp >= 60.0 and
        Body_Temp  <= 110.0),
        Insurance_StatusChar(1)
        Constraint          Patient_IS
        CHECK (Insurance-Status in ('Y', 'y', 'N', 'n')));
```

One column can have more than one CHECK constraint.

```
Create table_Loan - application
        (loan_app_no          number (6)       primary
        key,Name              Varchar2 (20),
        Amount_requestednumber (9, 2)          NOT NULL,
        Amount_approvednumber (9, 2)
        Constraint
                              Amount_approved_lim
        itCheck (Amount_approved <= 10,00,000)
        Constraint Amount_Approved_Interval
        Check (Mod (Amount_Approved, 1000) = 0));
```

**Establishing a Default value for a column :**

By using DEFAULT clause when defining a column, you can establish a default value for that column. This default value is used for a column, whenever, row is inserted into the table without specifying the column in the INSERT statement.

**Example :**

```
Create table_student

    (Student_id   Varchar2 (20),
    Last_name     Varchar2 (20)               NOT NULL,
    First_name    Varchar2 (20)               NOT NULL,
    B_Date              Date,
    State               Varchar2 (20),
    City                Varchar2 (20),        DEFAULT
                                              'PUNE'.
    Constraint          PK_student
    Primary key    (Student_id);
```

**2. ALTER TABLE Command :**

You can modify a table's definition using ALTER TABLE command. This statementchanges the structure of a table, not its contents. Using ALTER TABLE command, you can make following changes to the table.

(1) Adding a new column to an existing

```
table.ALTER  TABLE  table_name
ADD (Column_name                    datatype
   :
   :
 Column_name n                      datatype);
```

**Example :**

SQL> Describe Department;

| Name | NULL? | Type |
|---|---|---|
| Department_id | | Varachar2 (20) |
| Department_name | | Varachar2 (20) |

SQL> Alter table Department ADD (University Varchar2 (20),

No_of_student Number (3));

SQL> Describe Department;

| Name | Null | Type |
|---|---|---|
| Department_id | | Varachar2 (20) |
| Department_Name | | Varachar2 (20) |
| University | | Varachar2 (20) |
| No_of_student | | Varachar2 (20) |

(2) Modify an existing column in the existing table.ALTER TABLE table_name
MODIFY (Column_name datatype : constraint,
… Column_name datatype : constraint,);

A column in the table can be modified in following ways -

(i) Changing a column definition from NOT NULL to NULL i.e. from mandatory to optional

Consider a table ex_table.

SQL> describe ex_table;

| Name | NULL? | Type |
|---|---|---|
| Record_no | NOTNULL | Numbers (38) |
| Description | Varchar2 (40) | |
| Current_value | NOT NULL | |
| | NumberSQL> Alter Table | |

ex_table;
modify (current_value number
Null)
;Table altered

SQL> Describe ex_table;

| Name | NULL? | Type |
|---|---|---|
| Record_No | NOT NULL | Number (38) |
| Description | Varchar2 (40) | |
| Current_value | | Number |

(ii) Changing a column definition from NULL to NOT NULL.

If a table is empty, you can define a column to be NOT NULL. However, if table is not empty, you cannot change a column to NOT NULL unless every row in the table has a value for that particular column.

(iii) Increasing and Decreasing a Column's Width :

You can increase a character column's width and can increase the number of digits in a number column at any time.

**Example :**

SQL> Describe ex_table;

| | Name | NULL ? | Type |
|---|---|---|---|
| | Record_No NOT NULL | Number (38) | |
| SQL> | Description Varchar2 (40)Current_value Alter table ex_table | NOT NULL | Number |
| | modify | Varchar2 (50)); | |
| SQL> | (DescriptionTable altered Describe ex_table; Name | NULL ? | Type |
| | Record_No | NOT NULL | Number (38) |
| | Description | Varchar2 (50) | |
| | Current_value | NOT NULL | Number |

You can decrease a column's width only if the table is empty or if that column

is NULL for every row of table.

(3) Adding a constraint to an existing table :

Any constraint i.e. a primary key, foreign key, unique key or check constraint can be added to an existing table using ALTER TABLE command.

ALTER TABLE table_name

ADD (constraint)

**Example :**

SQL> Create Table ex_table
(Record_No Number
(38),
Description Varchar2
(40),Current_value
Number); Table created

SQL> Alter Table ex_table add
(Constraint PK_ex_table primary key (Record-No));
Table Altered.

(4) Dropping the constraints
ALTER TABLE table_name
DROP Primary key
Using this you can drop primary key of
table.ALTER TABLE Table_name
DROP constraint constraint_name
Using this you can drop any constraint of the table.

**Rules for adding or modifying a column :**

Following are the rules for adding column to a table :

(1) You may add a column at any time if NOT NULL is not specified.
(2) You may add a NOT NULL column in three steps :
   (i) Add a column without NOT NULL specified,
   (ii) Fill every row in that column with data,
   (iii) Modify the column to be NOT

NULL.Following are the rules to modify a
column.

(1) You can increase a character column's width at any time.
(2) You can increase the number of digits in a NUMBER column at any time.
(3) You can increase or decrease the number of places in a NUMBER column at any time.

If a column is NULL for every row of the table, you can make following changes.

(i) You can change its data type
(ii) You can decrease a character column's width
(iii) You can decrease the number of digits in a NUMBER column.

**3. DROP TABLE Command :**

Dropping a table means to remove the table's definition from the database. DROP TABLE command is used to drop the table as follows :

DROP TABLE table_name;

**Example :**

(1) SQL > Drop table_student;
Table dropped

(2) SQL > Drop table
instructor;Table dropped.

You drop a table only when you no longer need it.

**Note :** The truncate command in ORACLE can also be used to remove only the rows or data in the table and not the table definition.

**Example :**

Truncate student
Table truncated

Truncating cannot be rolled back.

## DATA MANIPULATION LANGUAGE COMMANDS

The SQL DML includes commands to insert tuples into database, to delete tuples from database and to modify tuples in the database.

It includes a query language based on both relational algebra and tuple relational calculus.

In this section we'll study following SQL DML commands.

INSERT
DELET
E
UPDAT
E
SELECT

**1. INSERT Command :**

The syntax of insert statement is :

INSERT INTO table_name
[(column_name [ , column_name] …… [ , column_name])]
VALUES
(column_value [ , column_value] …… [ ,
column_value]);The variables are defined as follows :

Table_name - is the table in which to insert the row.

column_name - is a column belonging to table.

column_value - is a literal value or an expression whose type matches the corresponding column_name.

The number of columns in the list of column_names must match the number of literal values or expressions that appear in parenthesis after the keyword *values*.

**Example :**

SQL> Insert into Employee
(E_name, B_Date, Salary, Address)
Values
('Sachin', '21-MAR-73', 50000.00, 'Mumbai');
row created

SQL> Insert into student
(Student_id, Last_name, First_name)
Values
('SE201', 'Tendulkar', 'Sachin');
row created

If the column names specified in Insert statement are more than values, then it returns an error.

Column and value datatype must match.

According to the syntax of INSERT statement, column list is an optional element. Therefore, if you do not specify the column names to be assigned values, it (DBMS) bydefault uses all the columns. The column order that DBMS uses is the order in which the columns were specified, when the table was created. However, use of Insert statement without column list is dangerous.

For example,

SQL> Describe ex_class;

| Name | NULL ? | Type |
|---|---|---|
| Class_building | NOT NULL | Varchar2 |
| (25)Class_room | NOT NULL | Varchar2 |
| (25) | | |
| Seating_capacity | Number | |

(38)SQL> Insert into ex_class
Values

('250', 'Kothrud Pune', 500);

1 row created.

The row is successfully inserted into the table, because, value and column data types were matching.

But the value 250 is not a correct value for column class_building.

The use of insert without column list may cause following problems.

1. The table definition might change, the number of columns might decrease or increase, and the INSERT fails as a result.
2. The INSERT statement might succeed but the wrong data could be entered in the table.

**2. DELETE Command :**

The syntax of delete statement is :

DELETE FROM table_name
[WHERE condition]

The variables are defined as follows :

table_name - is the table to be updated.

condition - is a valid SQL condition.

DELETE Command without WHERE clause will empty the table completely.

**Example :**

SQL> Delete from Student
Where Student_id = 'SE 201';
1 row deleted.

SQL> Detete from student
Where first_name = 'Sachin' and
Student_id ='SE 202';
1 row deleted.

**3. UPDATE Command :**

If you want to modify existing data in the database, UPDATE command can be used to do that. With this statement you can update zero or more rows in a table.

The syntax of UPDATE command is

:UPDATE table_name
SET column_name : : expression
[, column_name : : expression]
[, column_name : : expression]
[where condition]

The variables are defined as follows :

table_name is the table to be updated

column_name is a column in the table being updated.

expression is a valid SQL expression.

condition is a valid SQL condition.

The UPDATE statement references a single table and assigns an expression to at least one column. The WHERE clause is optional; if an UPDATE statement does not contain a WHERE clause, the assignment of a value to a column will be applied to all rows in the table.

**Example :**

SQL> Update Student
Set
City = 'Pune',
State = 'Maharashtra';

SQL> Update Instructor
Set
Position = 'Professor'
where

Instructor_id = 'P3021';

**SQL Grammar :**

Here, are some grammatical requirements to keep in mind when you are working with SQL.

1. Every SQL statement is terminated by a semicolon.
2. An SQL statement can be entered on one line or split across several lines for clarity.
3. SQL isn't case sensitive. You can mix uppercase and lowercase when referencing SQL keywords (Such as SELECT and INSERT), table names, and column names.

However, case does matter when referencing to the contents of a column.

For Example : If you ask for all customers whose last names begin with 'a' and all customer names are stored in uppercase, you won't receive any rows at all.

**4. SELECT Command :**

The basic structure of an SQL expression consists of three clauses :

select, from and where

- The select clause corresponds to the projection operation of the relational algebra.
  It is used to list the attributes desired in the result of a query.
- The from clause corresponds to the cartesian product operation of the relational algebra. It lists the relations to be scanned in the elevation of the expression.
- The where clause corresponds to the selection predicate of the relational algebra.
  It consists of predicate involving attributes of the relations that appear in thefrom clause.

Simple SQL query i.e. select statement has the form :

select $A_1, A_2, \ldots\ldots, A_n$

from $r_1, r_2, \ldots\ldots, r_m$

where P.

The variables are defined as follows :

$A_1, A_2, \ldots, A_n$ represent the attributes.

$r_1, r_2, \ldots, r_m$ represent the relations from which the attributes are selected.

P - is the predicate.

This query is equivalent to the relational algebra expression

$$\Box_{A_1\ A_2 \ldots\ A_n} (sp\ (r_1 \times r_2 \times r_3 \ldots \times r_m))$$

where clause is optional. If the where clause is omitted, the predicate P is true.

*Select* clause forms the cartesian product of relations named in the *from* clause, performs a relational algebra selection using the *where* clause and then projects the results onto the attributes of the select clause.

**A simple select statement :**

At a minimum, select statement contains the following two elements.

- The select list, the list of columns to be retrieved.
- The from clause, the tables from which to retrieve the rows.

**Example :** Consider the student database table.

(1) A simple select statement - a query that retrieves only student_id from the student table is given

```
SQL>  select student_id
      from student;
      student_id
      ..........................
      S 10231
      S 10232
```

S 10233

S 10234

S 10235

S 10236

6 rows selected.

(2) To select student_id and students Last name, the select statement is :

```
SQL> select student_id, First_name
     from student;
```

| student_id | First_name |
|---|---|
| S 10231 | Sachin |
| S 10232 | Rahul |
| S 10233 | Ajay |
| S 10234 | Sunil |
| S 10235 | Kapil |
| S 10236 | Anil |

6 rows selected.

To select all columns in the table you can use

```
select *
from table_name;
```

**Example :**

```
SQL> select *
     from student;
```

| student_id | Last_name | First_name | B Date | State | City |
|---|---|---|---|---|---|
| S 10231 | Deshpande | Sachin | 12/3/78 | Maharashtra | Pune |
| S 10232 | Gandhi | Rahul | 9/2/58 | Delhi | Delhi |
| S 10233 | Kapur | Ajay | 7/12/62 | Maharashtra | Bombay |
| S 10234 | Kulkarni | Sunil | 6/9/75 | Maharashtra | Pune |
| S 10235 | Dev | Kapil | 2/3/71 | Tamilnadu | Madras |
| S 10236 | Kumar | Anil | 5/9/80 | Maharashtra | Bombay |

The results returned by every SELECT statement constitutes a temporary table. Each received record is a row in this temporary table, and each element of the select list is a column. If a query does not return any record, the temporary - table can be thought of as empty.

**Expressions in the select list :**

In addition to specifying columns, you also can specify expressions in the select list.

Following arithmetic operators can be used in select list :

| Description | Operator |
|---|---|
| Addition | + |
| Subtraction | – |
| Multiplication | * |
| Division | / |

For example, consider the following queries using operators in select list :

```
SQL>Select E_name, Salary * 1000
    from Employee;
```

| E_name | Salary * 1000 |
|---|---|
| Sachin | 1,00,00,000 |
| Rahul | 2,00,00,000 |

| | |
|---|---|
| Ajay | 1,00,00,000 |
| Anil | 1,00,00,000 |

4 rows selected.

SQL>Select Ename, Salary + 10000from Employee;

| E_name | Salary + 10000 |
|---|---|
| Sachin | 20,000 |
| Rahul | 30,000 |
| Ajay | 20,000 |
| Anil | 30,000 |

4 rows selected.

**Select statement using where clause :**

*select* and *from* clauses provide you with either some columns and all rows or all columns and all rows. But if you want only certain rows, you need to add another clause, the *where* clause.

*where* clause consists of one or more conditions that must be satisfied before a row is retrieved by the query.

It searches for a condition and narrows your selection of data.

For example, consider select statement with where clause given below :

```
SQL>Select Student_id, First_name
    from Student
    where Student_id = 'S10234';
```

| Student_id | First_name |
|---|---|
| S10234 | Sunil |

1 row selected

```
SQL>Select E_name Salary
    from Employee
    where Salary  > 10000;
```

| E_name | Salary |
|---|---|
| Rahul | 20000 |
| Anil | 20000 |

2 row selected

where uses the logical connectives : **and, or** and **not**.

where clause uses the comparison operators

| Description | Operator |
|---|---|
| Less than | < |
| Less than or equal to | <= |
| Greater than | > |
| Greater than or equal to | >= |
| Equal to | = |
| Not equal to | != or <> |

```
SQL>Select E_name, Salary
    from Employee
    where Salary >  10000 and E_name  =  Anil
```

| Ename | Salary |
|---|---|
| Anil | 20000 |

1 row selected.

**5. Views in SQL :**

A view in SQL terminology is a single table that is derived from other tables. Theseother tables could be base tables or previously defined views. A view does not necessarily exist in physical form; it is considered a virtual table in contrast to base tables whose tuples are actually stored in the database. This limits the possible update operations that can be applied to views but does not provide any limitations on

querying a view. We can think of view as a way specifying a table that we need not exist physically.

**Specification of Views in SQL :**

The command to specify a view is CREATE VIEW. 'We give the view a table name, a list of attribute names, and a query to specify the contents of view. If none of the view attributes result from applying functions or arithmetic operations, we do not have to specify attribute names for the view as they will be the same as the names of the attributes of the defining tables.

**Example :**

Consider the following relation scheme and corresponding

relation.employee_schema (emp_name, street, city)

works_schema (emp_name, comp_name, salary

company_schema (comp_name, city)

| emp_name | street | city |
|---|---|---|
| Sachin | XYZ | Pune |
| Rahul | ABC | Bombay |
| Raj | ABC | Pune |
| Ajay | XYZ | Bombay |
| Anil | XYZ | Delhi |
| Sunil | ABC | Bombay |

| emp_name | Comp_name | salary |
|---|---|---|
| Sachin | TCS | 10000 |
| Rahul | MBT | 12000 |
| Raj | PCS | 13000 |
| Ajay | MBT | 14000 |
| Anil | PCS | 15000 |
| Sunil | TCS | 11000 |

| Comp_name | city |
|---|---|
| TCS | Delhi |
| MBT | Bomba |
| PCS | yPune |

```
Create view emp_detail (emp, comp, street, city)
As select C.emp_name, C.comp_name, E.street, E.city
from Employee E.company C
where E.emp_name = C.emp_name;
```

A view is always up date; if we modify the base tables on which the view is defined, the view automatically reflects these changes. Hence, the view is not realized at the time of view definition but rather at the time we specify a query onthe view. It is the responsibility of the DBMS and not the user to make sure that the view is up to date.

If we do not need a view any more, we can use the DROP VIEW command to dispose of it.

```
Drop View emp_detail;
```

**Updating of views :**

(1) A view with a single defining table is up datable if the view attributes contain the primary key or some other candidate key of the base relation, because this maps each view tuple to a single base tuple.

(2) Views defined on multiple tables using joins are generally not updatable.

(3) Views defined using grouping and aggregate functions are not updatable.

**Example :**

Consider the view consisting of branch names and names of customers who have either an account or a loan at that branch.

```
SQL>Create view all_customer as
    (select branch_name, customer_name
        from depositor, account
        where depositor·account_number =
    account·account·account_no)
    Union
    (select branch_name, customer_name
        from borrower
where borrower·loan_number = loan·loan_number);
```

The attribute names of a view can be specified explicitly as follows :

```
SQL> Create view branch_total_loan (branch_name,
        total_loan) as
    select branch_name, sum (amount)
    from loan
        group by branch_name;
```

6. **Indexes in SQL :** SQL has statements to create and drop indexes on attributes of base relation. These commands are generally considered to be part of the SQL data definition language (DDL).

An index is a physical access structure that is specified on one or more attributes of the relation. The attributes on which an index is created are termed indexing attributes. An index makes accusing tuples based on conditions that involve its indexing attributes more efficient. This means that in general executing a query will take less time if some attributes involved in the query conditions were indexed than if they were not. This improvement can be dramatic for queries where large relations are involved. In general, if attributes used in selection conditions and in join conditions of a query are indexed, the execution time of the query is greatly improved.

In SQL indexed can be created and dropped dynamically. The create Index command is used to specify an index. Each index is given a name, which is used to drop the index when we do not need it any more.

**Example :**

```
Create Index  Emp_Index
    ON Employee (Emp_name);
```

In general, the index is arranged in ascending order of the indexing attribute values. If we want the values in descending order we can add the keyword DESC after the attribute name. The default in ASC for ascending. We can also create an index on a combination of attributes.

**Example :**

```
Create  Index  Emp_Index1
    ON Employee (Emp_name ASC,
            Comp_name DESC);
```

There are two additional options on indexes in SQL. The first is to specify the key constraint on the indexing attribute or combination of attributes.

The keyword unique following the CREATE command is used to specify a key. The second option on index creation is to specify whether on index is clustering index. The keyword cluster is used in this case of the end of the create Index command. A base relation can have atmost one clustering index but any number of non_clustering indexes.

To drop an index, we issue the Drop Index command. The reason for dropping

indexes is that they are expensive to maintain whenever the base relation is updated and they require additional storage. However, the indexes that specify a key constraint should not be dropped as long as we want the system to continue enforcing that constraint.

**Example :**

Drop Index Emp_Index;

**7. Sequences**

The quickest way to retrieve data from a table is to have a column in the table whose data uniquely identifies a row.By using this column and a specific value in the WHEREcondition of a SELECT sentence the oracle engine will be able to identify and retrieve the row the fastest.
To achieve this , a constraint is attached to a specific column in the table that ensures that the column is never left empty and that the data values in the column are unique.Since human beings do data entry,it is quite likely that a duplicate value could be entered ,which violets this constraint and the entire row is rejected.
If the value entered into this column is computer generated it will always fulfill the unique constraint and the row will always be accepted for storage.
Oracle provides an object called a sequence that can generate numeric values. The value generated can have a maximum of 38 digits. A sequence can be defined to:
-Generate numbers in ascending or descending order
-Provide intervals between numbers
-Caching of sequence numbers in memory to speed up their availability
A sequence is an independent object and can be used with any table that requires its output.

**Creating Sequences**

Always give sequence a name so that it can be referenced later when required.
The minimum information required for generating numbers using a sequence is :
-The starting number
-The maximum number that can be generated by a sequence
- The increment value for generating the next number
This information is provided to oracle at the time of sequence creation

**Syntax:**

```
CREATE SEQUENCE <SequenceName>
[INCREMENT BY  <IntegerValue>
[START WITH <IntegerValue>
MAXVALUE <IntegerValue> /
NOMAXVALUEMINVALUE
<IntegerValue> /NOMINVALUE
CYCLE/NOCYLCLE
CACHE
<IntegerValue>/NOCACHE
ORDER/NOORDER ]
```

**Keywords and Parameters**

**INCREMENT BY : -**Specifies the interval between sequence numbers. It can be any positive or negative value but not zero.If this clause is omitted ,the default value is 1 .

**MINVALUE :-** Specifies the sequence minimum value.

**NOMINVALUE :**Specifies a minimum value of 1 for an ascending sequence and –(10)^26 for a descending sequence.

**MAXVALUE:** Specifies the maximum value that a sequence can generate.

**NOMAXVALUE :** Specifies a maximum of 10^27 for an ascending sequence or -1 fora descending sequence. This is the default clause.

**START WITH :**Speciifes the first sequence number to be generated. The default for anascending sequence is the sequence minimum value(1) and for a descending sequence, it is the maximum value(-1)

**CYCLE:** Specifies that the sequence continues to generate repeat values after reaching either its maximum value.

**NOCYCLE:** Specifies that a sequence cannot generate more values after reaching the maximum value.

**CACHE :**Specifies how many values of a sequence oracle pre-allocates and keeps in memory for faster access.The minimum value for this parameter is two.

**NOCHACHE :**Specifies that values of a sequence are not pre-allocated.

**ORDER :**This guarantees that sequence numbers are generated in the order of request.This is only necessary if using parallel server in parallel mode option .Inexclusive mode option ,a sequence always generates numbers in order.

**NOORDER :**This does not guarantee sequence numbers are generated in order of request.This is only necessary if you are using parallel server in parallel mode option. If the ORDER/NOORDER clause is omitted , a sequence takes the NOORDER clause by default.

**Example**

Create a sequence by the name ADDR_SEQ ,which will generate numbers from 1 uptp 9999 in ascending order with an interval of 1.The sequence must restart from the number 1 after generating number 999.

**CREATE SEQUENCE** ADDR_SEQ **INCREMENT BY 1 START WITH 1 MINVALUE 1MAXVALUE 999 CYCLE ;**

**Referencing a sequence**

Once a sequence is created SQL can be used to view the values held in its cache.To simply view sequence value use a SELECT sentence as described below.
SELECT <SequenceName>.Nextval from DUAL ;
This will display the next value held in the cache on the VDU screen. Everytime nextval references a sequence its output is automatically incremented from the old value to the new value ready for use.
To reference the current value of a sequence:
SELECT <SequenceName>.CurrVal FROM DUAL;

**Dropping a Sequence**

The DROP SEQUENCE command is used to remove the sequence from the database.
Syntax:
DROP SEQUENCE <SequenceName> ;

## DATA CONTROL LANGUAGE

The data control language commands are related to the security of database. They perform tasks of assigning privilages, so users can access certain objects in the database. This section deals with DCL commands.

1. **GRANT Command :**

The objects created by one user are not accessible by another user unless the owner of those objects gives such permissions to other users. These permissions can be given by using the **GRANT** statement. One user can grant permission to another user if he is the owner of the object or has the permission to grant access to other users.

The grant statement provides various types of access to database objects such as tables, views and sequences.

**Syntax :**

GRANT {object
privilages}ON object
name
To user name
[with GRANT OPTION]

**Object privilages :**

Each object privilage that is granted authorizes the grantee to perform some

operations on the object. The user can grant all the privilages or grant only specific object privilages.

The list of object privilages is as follows :

*Alter* - allows the grantee to change the table definition with the ALTER TABLE command.

*Delete* - allows the grantee to remove the records from the table with the DELETE command.

*Index* - allows the grantee to create an index on table with the CREATE INDEX command.

*Insert* - allows the grantee to add records to the table with the INSERT command.

*Select* - allows the grantee to query the tables with SELECT command.

*Update* - allows the grantee to modify the records in tables with UPDATE command.

*With grant option :* It allows the grantee to grant object privilages to other users.

**Example 1 :** Grant all privilages on student table to user Pradeep.

```
SQL > GRANT ALL
      ON student
      To Pradeep;
```

**Example 2 :** Grant select and update privilages on student table to mita

```
SQL> GRANT SELECT, UPDATE
      ON student
   To Mita;
```

**Example 3 :** Grant all privilages on student table to user Sachin with grant option.

```
SQL> GRANT ALL
      ON student
   To Sachin
      WITH GRANT OPTION;
```

**2. REVOKE Command :**

The REVOKE statement is used to deny the grant given on an object.

**Syntax :**

```
REVOKE {object
     privilages}ON
     object name FROM
     user name;
```

The list of object privilages is :

*Alter* - allows the grantee to change the table definition with the ALTER TABLE command.

*Delete* - allows the grantee to remove the records from the table with the DELETE command.

*Index* - allows the grantee to create an index on table with the CREATE INDEX command.

*Insert* - allows the grantee to add records to the table with the INSERT command.

*Select* - allows the grantee to query the tables with SELECT command.

*Update* - allows the grantee to modify the records in tables with UPDATE

command.You cannot use REVOKE command to perform following operations :

1. Revoke the object privilages that you didn't grant to the revokee.
2. Revoke the object privilages granted through the operating system.

**Example 1 :** Revoke Delete privilege on student table from Pradeep.

```
REVOKE DELETE
ON student
From Pradeep;
```

**Example 2 :** Revoke the remaining privilages on student that were granted to Pradeep.

```
    Revoke
    ALLON
    student
FROM    Pradeep
```

**3. COMMIT Command :**

Commit command is used to permanently record all changes that the user has made to the database since the last commit command was issued or since thebeginning of the database session.

**Syntax :**

COMMIT;

**Implicity COMMIT :**

The actions that will force a commit to occur even without your instructing it to are :

quit, exit,
create table or create view
drop table or drop view
grant or revoke
connect or disconnect
alter
audit and non-audit

Using any of these commands is just like using commit. Until you commit, only youcan see how your work affects the tables. Anyone else with access to these tables will continue to get the old information.

**4. ROLLBACK command :**

The ROLLBACK statement does the exact opposite of the commit statement. It ends the transaction but undoes any changes made during the transaction. Rollback is useful for two reasons :

(1) If you have made a mistake, such as deleting the wrong row for a table, you can use rollback to restore the original data. Rollback will take you back to intermediate statement in the current transaction, which means that you do not have to erase the entire transaction.

(2) ROLLBACK is useful if you have started a transaction that you cannot complete. This might occur if you have a logical problem or if there is an SQL statement that does not execute successfully. In such cases rollback allows you to return to the starting point to allow you to take corrective action and perhaps try again.

Syntax : ROLLBACK [WORK] [TO [SAVEPOINT] save

point]where

WORK - is optional and is provided for ANSI compatibility

SAVEPOINT - is optional and is used to rollback a partial transaction, as far as the specified save point.

Savepoint : is a savepoint created during the current transaction.

**Using rollback without savepoint clause.**

1. Ends the transaction.
2. Undoes all the changes in the current transaction.
3. Erases all savepoints in that transaction
4. Releases the transaction locks.

**Using rollback with the to savepoint clause.**

1. Rolls back just a portion of the transaction.
2. Retains the savepoint rolled back to, but losses those created after the named savepoint.
3. Releases all tables and row locks that were acquired since the savepoint was taken.

**Example :**

To rollback entire transaction : ROLLBACK,

To rollback to savepoint sps : ROLLBACK TO SAVEPOINT sps;

**Savepoints :**

Savepoints mark and save the current point in the current processing of a transaction. Used with the ROLLBACK statement, savepoints can undo part of a transaction.

By default the maximum number of savepoints per transaction is 5. An active savepoint is the one that is specified since the last commit or rollback.

**Syntax : SAVEPOINT savepoint :**

After a savepoint, is created, you can either continue processing, commit your work rollback the entire transaction, or rollback to the savepoint.

## SELECT QUERY AND CLAUSES

The basic structure of an SQL expression consists of three clauses :

select, from and where,

- The select clause corresponds to the projection operation of the relational algebra.
  It is used to list the attributes desired in the result of a query.
- The from clause corresponds to the cartesian product operation of the relational algebra. It lists the relations to be scanned in the elevation of the expression.
- The where clause corresponds to the selection predicate of the relational algebra.
  It consists of predicate involving attributes of the relations that appear in thefrom clause.
  Simple SQL query i.e. select statement has the form :

*select* $A_1, A_2, \ldots\ldots, A_n$

*from* $r_1, r_2, \ldots\ldots, r_m$

*where P.*

The variables are defined as follows :

$A_1, A_2, \ldots, A_n$ represent the attributes.

$r_1, r_2, \ldots, r_m$ represent the relations from which the attributes are selected.P - is the predicate.

This query is equivalent to the relational algebra expression

$\Pi_{A_1\ A_2 \ldots A_n} (s_p (r_1 \times r_2 \times \ldots \times r_m))$

where clause is optional. If the where clause is omitted, the predicate P is true. *Select* clause forms the cartesian product of relations named in the *from* clause, performs a relational algebra selection using the *where* clause and then projects the results onto the attributes of the select clause.

The purpose of select statement is to retrieve and display data from one or more database tables It is an extremely powerful statement capable of performing the equivalent relational algebra’s Selection, Projection, and Join operations in a single statement. Select is the most frequently used SQL command and has the following general form :

```
SELECT          DISTINCT |ALL]
FROM            Table_Name [alias][,…]
[WHERE          condition]
[GROUP BY       column_List] [HAVING
condition][ORDER BY        column_List]
```

The sequence of processing in a select statement is :

FROM
WHERE
GROUP
BY
HAVING

SELECT
ORDER
BY

The order of the clauses in the select command can not be changed. The only two mandatory columns are : SELECT and FROM, the remainder are optional.

**1. Expressions in the select list :**

In addition to specifying columns, you also can specify expressions in the select list.

Following arithmetic operators can be used in select list :

| Description | Operator |
|---|---|
| Addition | + |
| Subtraction | – |
| Multiplication | * |
| Division | / |

For example, consider the following queries using operators in select list :

```
SQL>Select E_name, Salary * 1000

from Employee;
E_name                  Salary * 1000
---------------------------------------
Sachin                  1,00,00,000
Rahul              2,00,00,000
Ajay               1,00,00,000
Anil               1,00,00,000
4 rows selected.
SQL>    Select E_name, Salary + 10000
        from Employee;
E_nam         Salary + 10000
---------------------------------------
Sachin             20,000
Rahul              30,000
Ajay               20,000
Anil               30,000
4 rows selected.
```

**2. Select statement using where clause :**

*select* and *from* clauses provide you with either some columns and all rows or all columns and all rows. But if you want only certain rows, you need to add another clause, the *where* clause.

*where* clause consists of one or more conditions that must be satisfied before a row is retrieved by the query.

It searches for a condition and narrows your selection of data.

For example, consider select statement with where clause given below :

```
SQL>    Select Student_id, First_Name
        from Student
        where Student_id = 'S10234';
        Student_id                         First_name
        -----------------------------------------------
        S10234 Sunil
        1 row selected
SQL>    Select E_name Salary
        from Employee
        where Salary > 10000
```

| E_name | Salary |
| --- | --- |
| Rahul | 20000 |
| Anil | 20000 |

2 row selected

where uses the logical connectives : **and, or** and **not**.

where clause uses the comparison operators

| Description | Operator |
|---|---|
| Less than | < |
| Less than or equal to | <= |
| Greater than | > |
| Greater than or equal to | >= |
| Equal to | = |
| Not equal to | != or < > |

```
SQL>Select E_name, Salary
       from  Employee
       where Salary>10000 and Ename = Anil
    E_name                              Salary
    ......................................................
    Anil                                20000
    1 row selected.
```

**Range Searching**

In order to select data that is within a range of values ,the BETWEEN operator is used. The BETWEEN operator allows the selection of rows that contain values within a specified lower and upper limit. The range coded after the word BETWEEN is inclusive.

The lower value must be coded first.The two values in between the range must be linked with the keyword AND.The BETWEEN operator can be used with both characterand numeric data types.However the datatypes can not be mixed.i.e the lower value ofa range of values from a character column and the other from a numeric column.

**Example 1 : List the transactions performed in months of January to March**

Solution :

SELECT * FROM TRANS_MSTR WHERE TO_CHAR(DT,'MM') BETWEEN 01 AND03 ;

Equivalent to

SELECT * FROM TRANS_MSTR WHERE TO_CHAR (DT,'MM')>=01 ANDTO_CHAR(DT,'MM')<=03;

Explanation

The above select will retrieve all those records from the ACCT_MSTR table where thevalue held in the DT field is between 01 and 03 (both values inclusive).This is done using TO_CHAR() function which extracts the month value from the DT field. This is then compared using the AND operator.

**Example 2 : List all the accounts which have not been accessed in the fourth quarter of the financial year**

Solution

SELECT DISTINCT FROM TRANS_MSTR WHERE TO_CHAR(DT,'MM') NOT BETWEEN 01 AND 04 ;

Explanation

The above select will retrieve all those records from the ACCT_MSTR table where thevalue held in the DT field is not between 01 and 04(both values inclusive).This is doneusing TO_CHAR() function which extracts the month value from the DT field and thencompares them using the not and the between operator.

**SELECT STATEMENT WITH ORDER BY CLAUSE**

ORDER BY clause is similar to the GROUP BY clause. The ORDER BY clauseenables you to sort your data in either ascending or descending order.

The ORDER BY clause consists of a list of column identifiers that the result is to be sorted on, separated by columns. A column identifier may be either a column name or a column number.

It is possible to include more than one element in the ORDER BY clause. The major sort key determines the overall order of the result table

If the values of the major sort key are unique, there is no need for additional keys to control the sort. However, if the values of the major sort key are not unique, there may be multiple rows in the result table with the same value for the major sort key. In this case it may be desirable to order rows with the same value for the major sort key by some additional sort key. If a second element appears in the ORDER BY clause, it is called a ***minor sort key.***

**Example :** Consider the worker database :

```
SQL>select *
        from worker
    order By F_NAME asc 0;
```

| F_NAME | STATUS | GENDER | BIRTHDATE |
|---|---|---|---|
| Ajay | Regular | M | 05 / 03 / 69 |
| Ashwini | Regular | F | 11 / 01 / 70 |
| Rahul | Summer | M | 01 / 12 / 72 |
| Smita | Regular | F | 23 / 09 / 67 |

## GROUP BY CLAUSE

Another helpful clause is the group by clause. A group by clause arranges your data rows into a group according to the columns you specify.

A query that includes group by clause is called a ***grouped query*** because it groups that data from the SELECT tables and generates single summary row for each group.

The columns named in the group by clause are called the ***grouping columns.***

When GROUP BY clause is used, each item in the SELECT list must be single-valued per group.

The select clause may contain only :

- Column names
- Aggregate functions
- Constants
- An expression involving combinations of the above.

All column names in SELECT must appear in GROUP BY clause, unless the name is used only in an aggregate function. The contrary is not true; there may be column names in GROUP BY clause that do not appear in SELECT clause.

When the WHERE clause is used with GROUP BY the WHERE clause is applied first, then groups are formed from the remaining rows that satisfy the search condition.

**Example :**

Consider the worker table given below :

```
SQL>select *
        from worker
STATUS   GENDER   BIRTHDATE   F_NAME
```

| | | | |
|---|---|---|---|
| Ashwini | Regular | F | 11 / 01 / 70 |
| Rahul | Summer | M | 01 / 12 / 72 |
| Ajay | Regular | M | 05 / 03 / 69 |
| Smita | Regular | F | 23 / 09 / 67 |

```
SQL>Select   *
        from worker
        Group By status;
```

| F_NAME | STATUS | GENDER | BIRTHDATE |
|---|---|---|---|
| Ashwini | Regular | F | 11 / 01 / 70 |
| Ajay | Regular | M | 05 / 03 / 69 |
| Smita | Regular | F | 23 / 09 / 67 |
| Rahul | Summer | M | 01 / 12 / 72 |

(2) To group by more than one column,

```
SQL>select   *
        from worker
        Group By status, Gender;
```

| F_NAME | STATUS | GENDER | BIRTHDATE |
|---|---|---|---|
| Ashwini | Regular | F | 11 / 01 / 70 |
| Smita | Regular | F | 23 / 09 / 67 |
| Ajay | Regular | M | 05 / 03 / 69 |
| Rahul | Summer | M | 01 / 12 / 72 |

**2.2, 2.3,2.4,2.5,2.6, 2.7 Check Your Progress**

**Fill in the blanks**

1) DCL contain …………………&........................... commands.
2) Primry Key is the combination of…………………&…………………
3) After table command operates on ............................ ends.
4)cmd is used to save data in database.
5) The condition in group by clause is given by ............................ clause.

## HAVING CLAUSE

The Having clause is similar to the where clause. The Having clause does for aggregate data what where clause does for individual rows. The having clause is another search condition. In this case, however, the search is based on each group of grouped table.

The difference between where clause and having clause is in the way the query is processed.

In a where clause, the search condition on the row is performed before rows are grouped. In having clause, the groups are formed first and the search condition is applied to the group.

Syntax is :

```
select select_list
from table_list
[where condition [AND : OR] …… condition]
```

[group by column 1, column 2, …… column
N][Having condition]

**Example :**

```
SQL>select  *
        from worker
        Group By status, Gender
        Having Gender  = 'F';
```

| F_NAME | STATUS | GENDER | BIRTHDATE |
|---|---|---|---|
| Ashwin | Regular | F | 11 / 01 / 70 |
| i Smita | Regular | F | 23 / 09 / 72 |

```
SQL>select  *
        from worker
        where Birthdate < 11 / 01 / 70
        Group By status, Gender
        Having Gender  = 'M';
```

| F_NAME | STATUS | GENDER | BIRTHDATE |
|---|---|---|---|
| Ajay | Regular | M | 05 / 03 / 69 |

# ACHARYA INSTITUTE OF GRADUATE STUDIES
**(NAAC Re-Accredited and Affiliated to Bangalore City University)**
**Soldevanahalli, Bengaluru-560107**

**Unit 4: RELATIONAL DATABASE LANGUAGE** (Introduction to SQL, Features of SQL, SQL Languages, DDL commands- Create, Add, Drop, Constraints in SQL, DML Commands – Insert, Delete, Update)
**Unit 5: DATA QUERY LANGUAGE** (Where clause, Order by, Group by, DCL commands – Grant, Revoke, TCL Commands – Commit, Roll Back, Savepoint, Aggregate Functions, Relational Algebra)

## INTRODUCTION

Most of the problems faced at the time of implementation of any system are outcome of a poor database design. In many cases it happens that system has to be continuously modified in multiple respects due to changing requirements of users. It is very important that a proper planning has to be done.
A relation in a relational database is based on a relational schema, which consists of number of attributes.
A relational database is made up of a number of relations and corresponding relational database schema.
The goal of a relational database design is to generate a set of relation schema that allows us to store information without unnecessary redundancy and also to retrieve information easily.
One approach to design schemas that are in an appropriate normal form. The normal forms are used to ensure that various types of anomalies and inconsistencies are not introduced into the database.

## WHAT IS RDBMS?

RDBMS stands for Relational Database Management System. RDBMS data is structured in database tables, fields and records. Each RDBMS table consists of database table rows. Each database table row consists of one or more database table fields.
RDBMS store the data into collection of tables, which might be related by common fields (database table columns). RDBMS also provide relational operators to manipulate the data stored into the database tables. Most RDBMS use sql as database querylanguage.The most popular RDBMS are MS SQL Server, DB2, Oracle and MySQL.
The relational model is an example of record-based model. Record based models are so named because the database is structured in fixed format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record types.

The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model.

The relational model was designed by the IBM research scientist and mathematician, Dr.E.F.Cod. The relational model originated from a paper authored by Dr.codd entitled "A Relational Model of Data for Large Shared Data Banks", written in 1970. This paper included the following concepts that apply to database management systems forrelational databases.

The relation is the only data structure used in the relational data model to represent both entities and relationships between them.

Tuples, Attributes and Domain.

Rows of the relation are referred to as **tuples** of the relation and columns are its **attributes**. Each attribute of the column are drawn from the set of values known as **domain**. The domain of an attribute contains the set of values that the attribute may assume.
From the historical perspective, the relational data model is relatively new .The first database systems were based on either network or hierarchical models .The relational data model has established itself as the primary data model for commercial data processing applications. Its success in this domain has led to its applications outside data processing in systems for computer aided design and other environments.

DIFFERENCE BETWEEN DBMS & RDBMS

A DBMS has to be persistent, that is it should be accessible when the programcreated the data ceases to exist or even the application that created the data restarted. A DBMS also has to provide some uniform methods independent of a specific application for accessing the information that is stored. RDBMS is a Relational Data Base Management System Relational DBMS. This adds the additional condition that the system supports a tabular structure for the data, with enforced relationships between the tables. This excludes the databases that don't support a tabular structure or don't enforce relationships between tables. You can say DBMS does not impose any constraints or security with regard to data manipulation it is user or the programmer responsibility to ensure the ACID PROPERTY of the database whereas the RDBMS is more with this regard because RDBMS define the integrity constraint for the purpose of holding ACID PROPERTY.

The DBMS interfaces with application programs so that the data contained in the database can be used by multiple applications and users. The DBMS allows these users to access and manipulate the data contained in the database in a convenient and effective manner. In addition the DBMS exerts centralized control of the database, prevents unauthorized

users from accessing the data and ensures privacy of data In this chapter we study the query language : Structured Query Language (SQL) which uses a combination of Relational algebra and Relational calculus.It is a data sub language used to organize, manage and retrieve data fromrelational database, which is managed by Relational Database Management System (RDBMS).

Vendors of DBMS like Oracle, IBM, DB2, Sybase, and Ingress use SQL asprogramming language for their database.SQL originated with the system R project in 1974 at IBM's San Jose ResearchCentre.Original version of SQL was SEQUEL which was an Application Program Interface(API) to the system R project.

The predecessor of SEQUEL was named SQUARE.SQL-92 is the current standard and is the current version.The SQL language can be used in two ways

- Interactively or
- Embedded inside another program

The SQL is used interactively to directly operate a database and produce the desired results. The user enters SQL command that is immediately executed. Most databases have a tool that allows interactive execution of the SQL language. These include SQL Base's SQL Talk, Oracle's SQL Plus, and Microsoft's SQL server 7 QueryAnalyzer.

The second way to execute a SQL command is by embedding it in another language such as Cobol, Pascal, BASIC, C, Visual Basic, Java, etc. The result of embedded SQL command is passed to the variables in the host program, which in turnwill deal with them. The combination of SQL with a fourth-generation language brings together the best of two worlds and allows creation of user interfaces and database access in one application.

What is SQL?

- SQL stands for Structured Query Language. It is used for storing and managing data in RelationalDatabase Management System (RDBMS).
- It is a standard language for Relational Database System. It enables a user to **create, read, update and delete** relational databases and tables.
- All the RDBMS like **MySQL, Informix, Oracle, MS Access and SQL Server** use SQL as their standard database language.
- SQL allows users to query the database in a number of ways, using English-like statements.

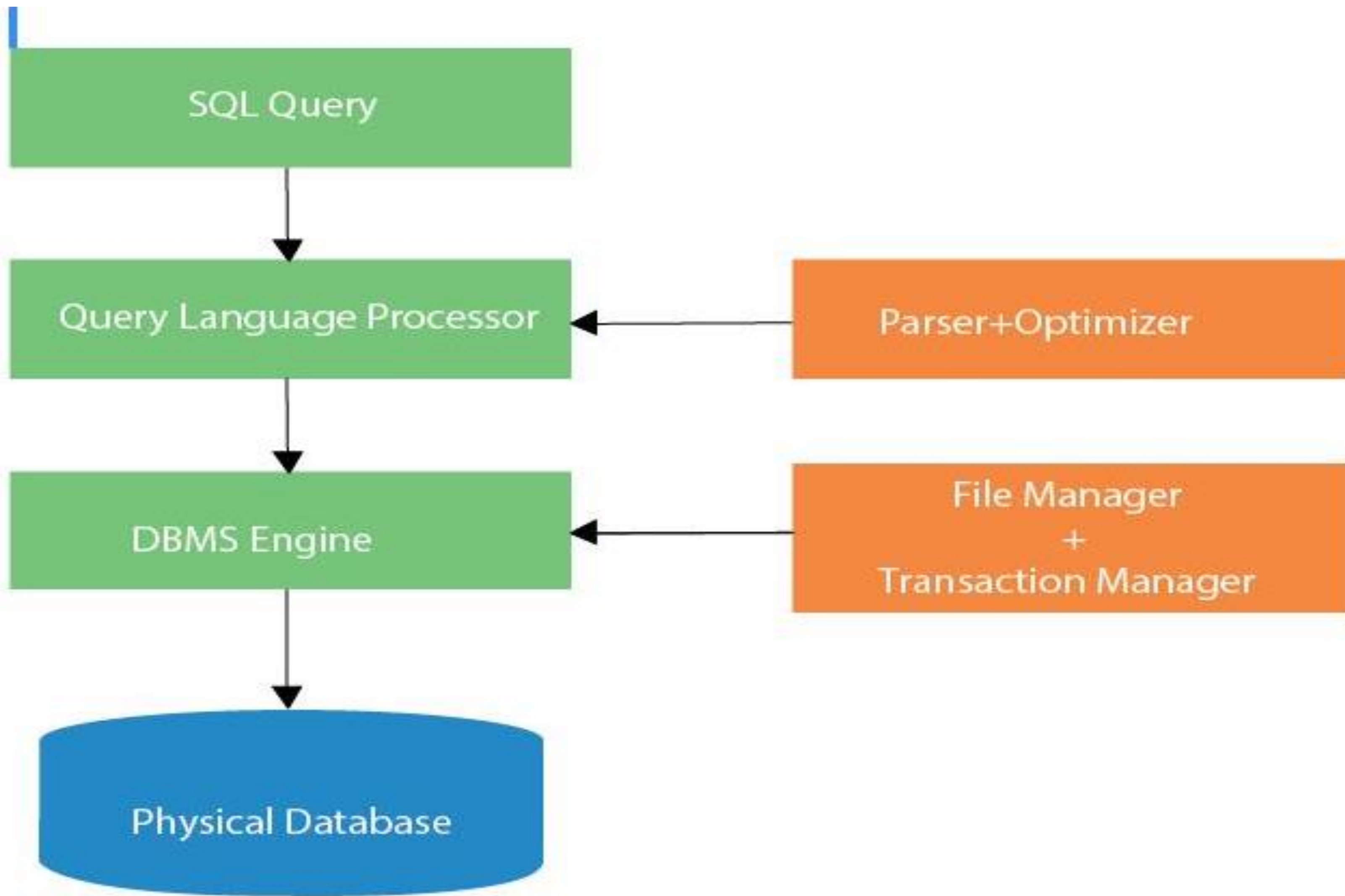
SQL follows the following rules:

- Structure query language is not case sensitive. Generally,keywords of SQL are written in uppercase.
- Statements of SQL are dependent on text lines. We canuse a single SQL statement on one or multiple text line.
- Using the SQL statements, you can perform most of theactions in a database.

# What is SQL Process?

- When an SQL command is executing for any RDBMS, then the system figure out the best way to carry out the request and the SQL engine determines that howto interpret the task.
- In the process, various components are included. These components can be optimization Engine, Queryengine, Query dispatcher, classic, etc.
- All the non-SQL queries are handled by the classicquery engine, but SQL query engine won't handlelogical files.

# What is SQL Process?

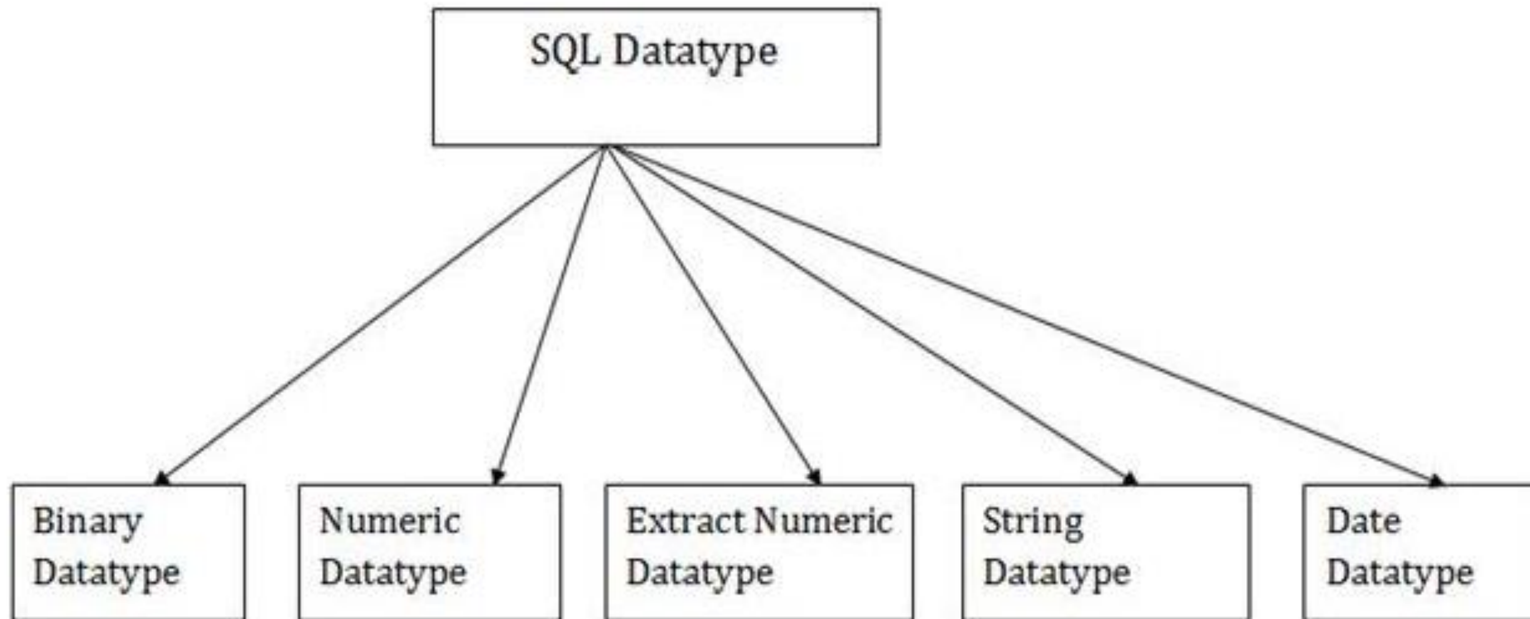


# What is Advantages of SQL?

- High speed
- No coding needed
- Well defined standards
- Portability
- Interactive language
- Multiple data view

What is SQL Datatype?

- SQL Datatype is used to define the values that acolumn can contain.
- Every column is required to have a name and datatype in the database table.

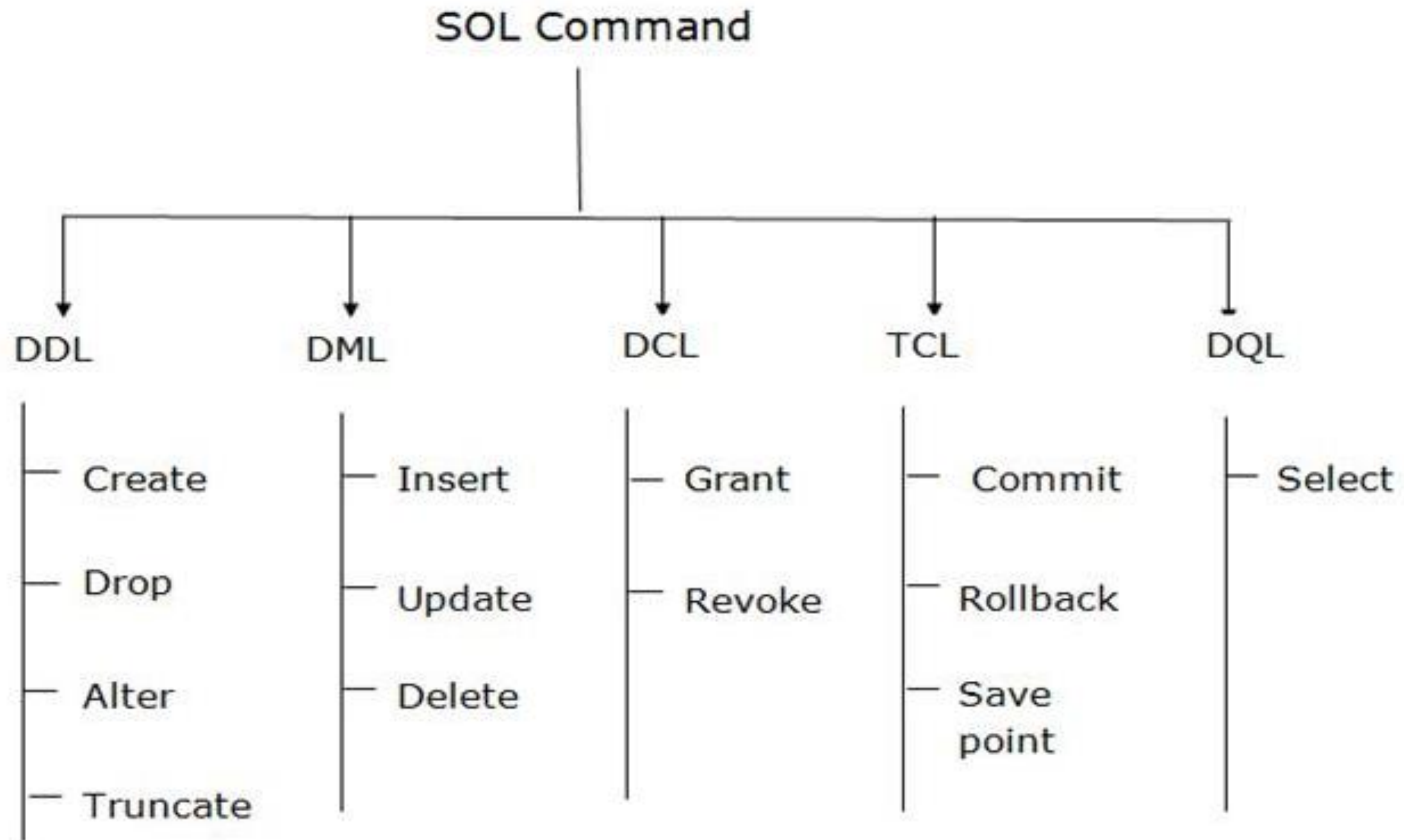


# SQL Commands

- SQL commands are instructions. It is used to communicate with the database. It is also used toperform specific tasks, functions, and queries of data.
- SQL can perform various tasks like create a table, add data to tables, drop the table, modify the table, set permission for users.

# Types of SQL Commands

- There are five types of SQL commands: DDL, DML,DCL, TCL, and DQL.



# Data Definition Language (DDL)

- DDL changes the structure of the table like creating atable, deleting a table, altering a table, etc.

- All the command of DDL are auto-committed thatmeans it permanently save all the changes in the database.
- Here are some commands that come under DDL:
  - CREATE
  - ALTER
  - DROP
  - TRUNCATE

# Data Definition Language (DDL)- CREATE

**CREATE** It is used to create a new table in the database.

**Syntax:**

REATE TABLE TABLE_NAME (COLUMN_NAME DATATYPES[,. ........................................................................ ]);

**Example:**

CREATE TABLE EMPLOYEE(Name VARCHAR2(20), Email VARCHAR2(100), DOB DATE);

# Data Definition Language (DDL)- Drop

Drop: It is used to delete both the structure and recordstored in the table.

**Syntax:**

DROP TABLE ;

**Example:**

DROP TABLE EMPLOYEE;

# Data Definition Language (DDL)- **ALTER**

**ALTER:** It is used to alter the structure of the database. This changecould be either to modify the characteristics of an existing attribute or probably to add a new attribute.

**Syntax:**

ALTER TABLE table_name ADD column_name COLUMN-definition;ALTER TABLE MODIFY(COLUMN DEFINITION........................................................................................... );

**Example:**

ALTER TABLE STU_DETAILS ADD(ADDRESS VARCHAR2(20)); ALTER TABLE STU_DETAILS MODIFY (NAME VARCHAR2(20));

# Data Definition Language (DDL)- TRUNCATE

**TRUNCATE:** It is used to delete all the rows from the table and freethe space containing the table.

**Syntax:**

```
TRUNCATE TABLE table_name;
```

**Example:**

```
TRUNCATE TABLE EMPLOYEE;
```

# Data Manipulation Language

- DML commands are used to modify the database. It isresponsible for all form of CHANGES in the database.
- The command of **DML is not auto-committed** that means it can't permanently save all the changes in thedatabase. They can be rollback.

Here are some commands that come under DML:

- **INSERT**
- **UPDATE**
- **DELETE**

# Data Manipulation Language - INSERT

**INSERT:** The INSERT statement is a SQL query. It is used to insertdata into the row of a table.

**Syntax:**

INSERT INTO TABLE_NAME (col1, col2, col3,.... col N)
VALUES (value1, value2, value3,.................................................... valueN);

OR

INSERT INTO TABLE_NAME VALUES (value1, value2, value3, ..................................... valueN);

**Example:**

INSERT INTO XYZ (Author, Subject) VALUES ("Sonoo", "DBMS");

# Data Manipulation Language - **UPDATE**

Update: This command is used to **update or modify** the value of acolumn in the table.

**Syntax:**

```
UPDATE table_name SET [column_name1= value1,...column_nameN = valueN] [WHERE CONDITION]
```

**Example:**

```
UPDATE students
SET User_Name = 'Sonoo'WHERE Student_Id = '3'
```

# Data Control Language

DCL commands are used to GRANT and TAKE BACKauthority from any database user.

Here are some commands that come under DCL:

- Grant
- Revoke

# Data Control Language - **Grant**

**GRANT:** It is used to give user access privileges to a database.

**Example:**

```
GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER, ANOTHER_USER;
```

**REVOKE:** It is used to take back permissions from the user.

**Example:**

```
REVOKE SELECT, UPDATE ON MY_TABLE FROM USER1, USER2;
```

# Transaction Control Language

TCL commands can only use with DML commands likeINSERT, DELETE and UPDATE only.

These operations are automatically committed in the database that's why they cannot be used while creatingtables or dropping them.

Here are some commands that come under TCL:

- COMMIT
- ROLLBACK
- SAVEPOINT

# Transaction Control Language - COMMIT

**Commit:** Commit command is used to save all the transactions tothe database.

**Syntex:**

```
COMMIT;
```

**Example:**

```
DELETE FROM CUSTOMERS WHERE AGE = 25;
COMMIT;
```

# Transaction Control Language - **Rollback**

**Rollback:** Rollback command isused to undo transactions that have not already been saved to the database.

**Syntex:**

```
ROLLBACK;
```

**Example:**

```
DELETE FROM CUSTOMERSWHERE AGE = 25;
ROLLBACK;
```

**SAVEPOINT:** It is used to roll thetransaction back to a certain point without rolling back the entire transaction.

**Syntex:**

```
SAVEPOINT SAVEPOINT_NAME;
```

# Data Query Language

DQL is used to fetch the data from the database.It uses only one command:

SELECT

**a.** **SELECT:** This is the same as the projection operation of relational algebra. It is used to select the attribute based onthe condition described by WHERE clause.

**Syntax:**

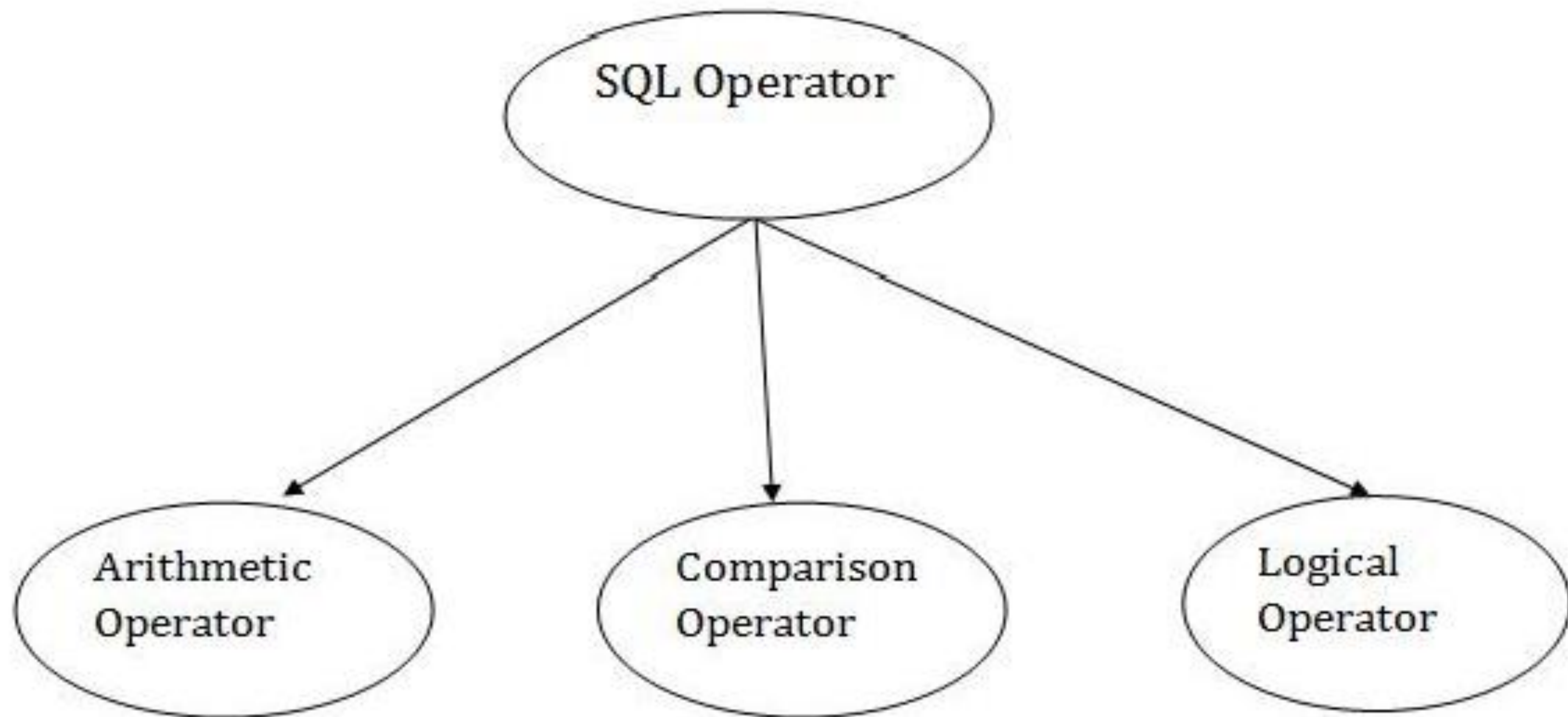
SELECT expressions FROM TABLES WHERE conditions;

**Example:**

SELECT emp_name FROM employee WHERE age > 20;

# SQL Operator

There are various types of SQL operator:



# SQL Comparison Operators:

| Operator | Description |
|---|---|
| + | It adds the value of both operands. |
| - | It is used to subtract the right-hand operand from the left-handoperand. |
| * | It is used to multiply the value of both operands. |
| / | It is used to divide the left-hand operand by the right-handoperand. |
| % | It is used to divide the left-hand operand by the right-handoperand and returns reminder. |

# SQL Arithmetic Operators

| Operator | Description |
|---|---|
| = | It checks if two operands values are equal or not, if the values arequeal then condition becomes true. |
| != | It checks if two operands values are equal or not, if values are notequal, then condition becomes true. |
| <> | It checks if two operands values are equal or not, if values are notequal then condition becomes true. |
| > | It checks if the left operand value is greater than right operand value,if yes then condition becomes true. |
| < | It checks if the left operand value is less than right operand value, ifyes then condition becomes true. |
| >= | It checks if the left operand value is greater than or equal to the rightoperand value, if yes then condition becomes true. |

# SQL Arithmetic Operators

| Operator | Description |
|---|---|
| <= | It checks if the left operand value is less than or equal to the rightoperand value, if yes then condition becomes true. |
| !< | It checks if the left operand value is not less than the right operandvalue, if yes then condition becomes true. |
| !> | It checks if the left operand value is not greater than the right operandvalue, if yes then condition becomes true. |

# SQL Logical Operators

| Operator | Description |
|---|---|
| All | It compares a value to all values in another value set. |
| AND | It allows the existence of multiple conditions in an SQL statement. |
| ANY | It compares the values in the list according to the condition. |
| Between | It is used to search for values that are within a set of values. |
| IN | It compares a value to that specified list value. |
| NOT | It reverses the meaning of any logical operator. |
| OR | It combines multiple conditions in SQL statements. |
| EXIST | It is used to search for the presence of a row in a specified table. |
| LIKE | It compares a value to similar values using wildcard operator. |

# Example:

SQL> CREATE TABLE EMPLOYEE ( EMP_ID INT
NOT NULL,
EMP_NAME VARCHAR (25) NOT NULL, PHONE_NO
INT NOT NULL,
ADDRESS CHAR (30),

PRIMARY KEY (ID)

);

- **DESC EMPLOYEE;**
- DELETE FROM table_name WHERE condition
- DROP TABLE "table_name";
- SELECT * FROM table_name;
- INSERT INTO TABLE_NAME VALUES (value1, value2, value 3, .... Value N);
- INSERT INTO TABLE_NAME[(col1, col2, col3,.... col N)] VALUES (value1, value2, value 3, .... Value N);
- UPDATE table_name SET column_name = value WHERE condition;

# Example:

- UPDATE table_name SET column_name = value1, column_name2 = valueWHERE condition;
- DELETE FROM table_name WHERE some_condition;

# Views in SQL

- Views in SQL are considered as a virtual table. A viewalso contains rows and columns.
- To create the view, we can select the fields from oneor more tables present in the database.
- A view can either have specific rows based on certaincondition or all the rows of a table.

# Creating view

A view can be created using the **CREATE VIEW** statement. We cancreate a view from a single table or multiple tables.

Syntax

```
CREATE VIEW view_name ASSELECT
column1, column2.....
FROM table_nameWHERE
condition;
```

Creating View from a single table

```
CREATE VIEW DetailsView ASSELECT
NAME, ADDRESS
FROM Student_DetailsWHERE
STU_ID < 4;
```

# Creating View from multiple tables

View from multiple tables can be created by simply includemultiple tables in the SELECT statement.

In the given example, a view is created named MarksView fromtwo tables Student_Detail and Student_Marks.

```
CREATE VIEW MarksView AS
SELECT Student_Detail.NAME, Student_Detail.ADDRESS, Student_Marks.MARKSFROM Student_Detail, Student_Mark
WHERE Student_Detail.NAME = Student_Marks.NAME;

SELECT * FROM MarksView;DROP VIEW
view_name;
```

# SQL Index

- Indexes are special lookup tables. It is used to retrieve data from the database very fast.
- An Index is used to speed up select queries and where clauses. But it shows down the data input with insert and update statements. Indexes can be created or dropped without affecting the data.
- An index in a database is just like an index in the back of a book.

## Create Index statement

```
CREATE INDEX index_name
ON table_name (column1, column2, ...);
```

# Unique Index statement

**Syntax**

CREATE UNIQUE INDEX index_name ON table_name (column1, colum
n2, ...);

**Example**

CREATE UNIQUE INDEX websites_idx ON websites (site_name);

Drop Index Statement

**Syntax**

DROP INDEX index_name;DROP INDEX

**Example**

websites_idx;

# SQL Sub Query

A Subquery is a query within another SQL query and embeddedwithin the WHERE clause.

**Important Rule:**

- A subquery can be placed in a number of SQL clauses like WHERE clause,FROM clause, HAVING clause.
- You can use Subquery with SELECT, UPDATE, INSERT, DELETE statementsalong with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.
- A subquery is a query within another query. The outer query is known asthe main query, and the inner query is known as a subquery.
- Subqueries are on the right side of the comparison operator.
- A subquery is enclosed in parentheses.
- In the Subquery, ORDER BY command cannot be used. But **GROUP BYcommand can be used** to perform the same function as ORDER BY command.

# Subqueries with the Select Statement

SQL subqueries are most frequently used with the Select statement.

**Syntax:**

```
SELECT column_nameFROM
table_name

WHERE column_name expression operator
( SELECT column_name   from table_name WHERE ... );
```

**Example:**

```
SELECT *
    FROM EMPLOYEE WHERE ID IN
    (SELECT IDFROM EMPLOYEE
    WHERE SALARY > 4500);
```

# Subqueries with the INSERT Statement

- SQL subquery can also be used with the Insert statement. In the insert statement, data returned from the subquery is used to insert into anothertable.
- In the subquery, the selected data can be modified with any of thecharacter, date functions.

**Syntax:**

INSERT INTO table_name (column1, column2, column3 .......................................................... )
**SELECT** * **FROM** table_name **WHERE** VALUE OPERATOR

**Example:**

```
INSERT INTO EMPLOYEE_BKP SELECT *
    FROM EMPLOYEEWHERE ID IN
    (SELECT ID FROM EMPLOYEE);
```

# Subqueries with the UPDATE Statement

The subquery of SQL can be used in conjunction with the Updatestatement. When a subquery is used with the Update statement, then either single or multiple columns in a table can be updated.

**Syntax:**

```
UPDATE table        SET column_name = new_value WHERE VALUE OPERATOR(SELECT COLUMN_NAME
                    FROM TABLE_NAME   WHERE condition);
```

**Example:**

Let's assume we have an EMPLOYEE_BKP table available which is backup ofEMPLOYEE table. The given example updates the SALARY by .25 times in theEMPLOYEE table for all employee whose AGE is greater than or equal to 29.

```
UPDATE EMPLOYEE
    SET SALARY = SALARY * 0.25
        WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKPWHERE AGE >= 29);
```

# Subqueries with the DELETE Statement

The subquery of SQL can be used in conjunction with the Deletestatement just like any other statements mentioned above.

**Syntax:**

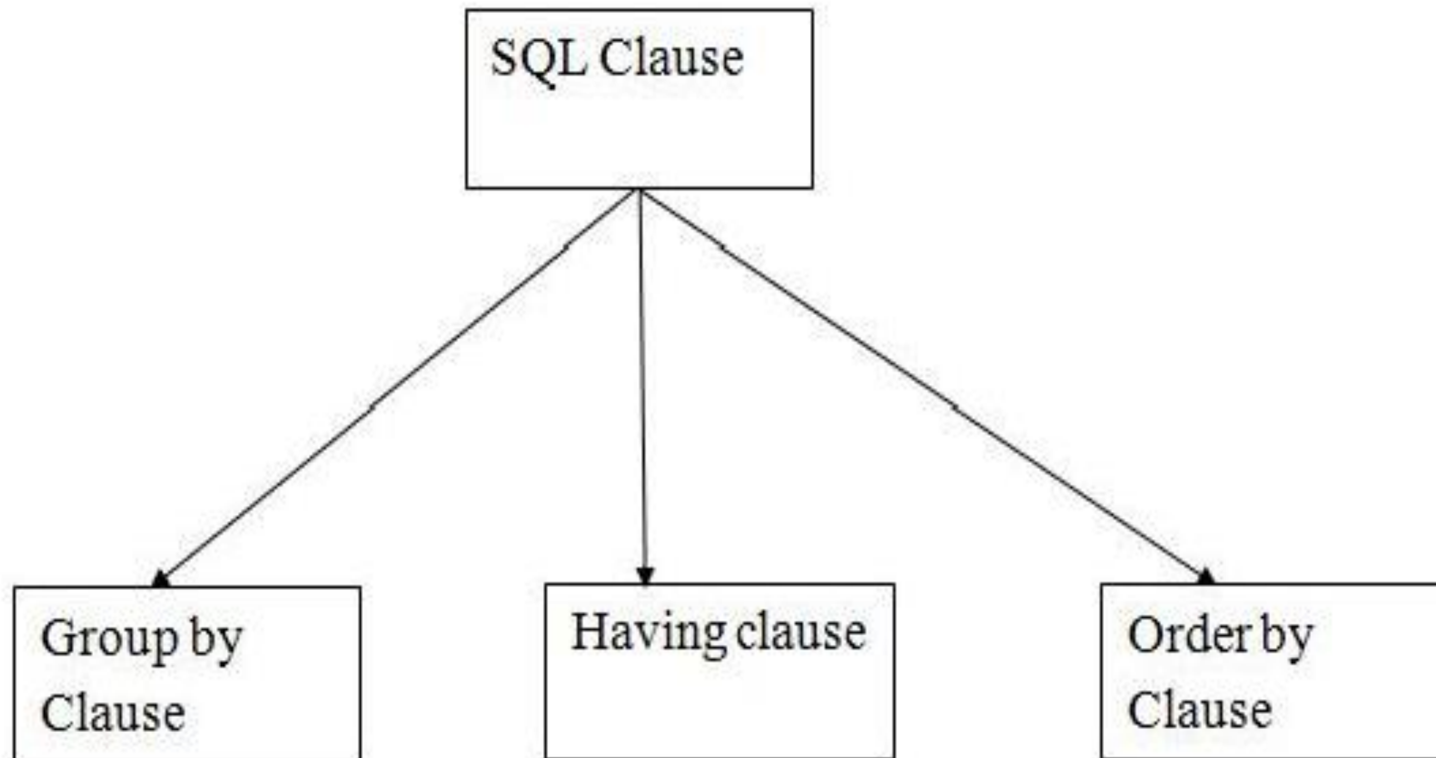
**DELETE FROM TABLE_NAME WHERE VALUE OPERATOR**
**(SELECT COLUMN_NAME FROM TABLE_NAME WHERE condition);**

**Example:**

Let's assume we have an EMPLOYEE_BKP table available which is backup of EMPLOYEE table. The given example deletes the records from the EMPLOYEEtable for all EMPLOYEE whose AGE is greater than or equal to 29.

**DELETE FROM EMPLOYEE**
**WHERE AGE IN (SELECT AGE FROM EMPLOYEE_BKPWHERE AGE >= 29 );**

# SQL Clauses



# GROUP BY

- SQL GROUP BY statement is used to arrange identical data intogroups.
- The GROUP BY statement is used with the SQL SELECTstatement.
- The GROUP BY statement follows the WHERE clause in aSELECT statement and precedes the ORDER BY clause.
- The GROUP BY statement is used with aggregation function.

**Syntax**

SELECT column FROM table_name
WHERE conditionsGROUP BY
columnORDER BY column

**Example**

SELECT COMPANY, COUNT(*)FROM
PRODUCT_MAST GROUP BY COMPANY;

# HAVING

- HAVING clause is used to specify a search condition for a groupor an aggregate.
- Having is used in a GROUP BY clause. If you are not using GROUP BY clause then you can use HAVING function like aWHERE clause

**Syntax**

SELECT column1, column2   FROM table_name

WHERE conditions

GROUP BY column1, column2HAVING conditions

ORDER BY column1, column2;

**Example**

SELECT COMPANY, COUNT(*)FROM PRODUCT_MAST GROUP BY COMPANY HAVING COUNT(*)>2;

# ORDER BY

- The ORDER BY clause sorts the result-set in ascending ordescending order.
- It sorts the records in ascending order by default. DESC keyword is used to sort the records in descending order.

**Syntax**

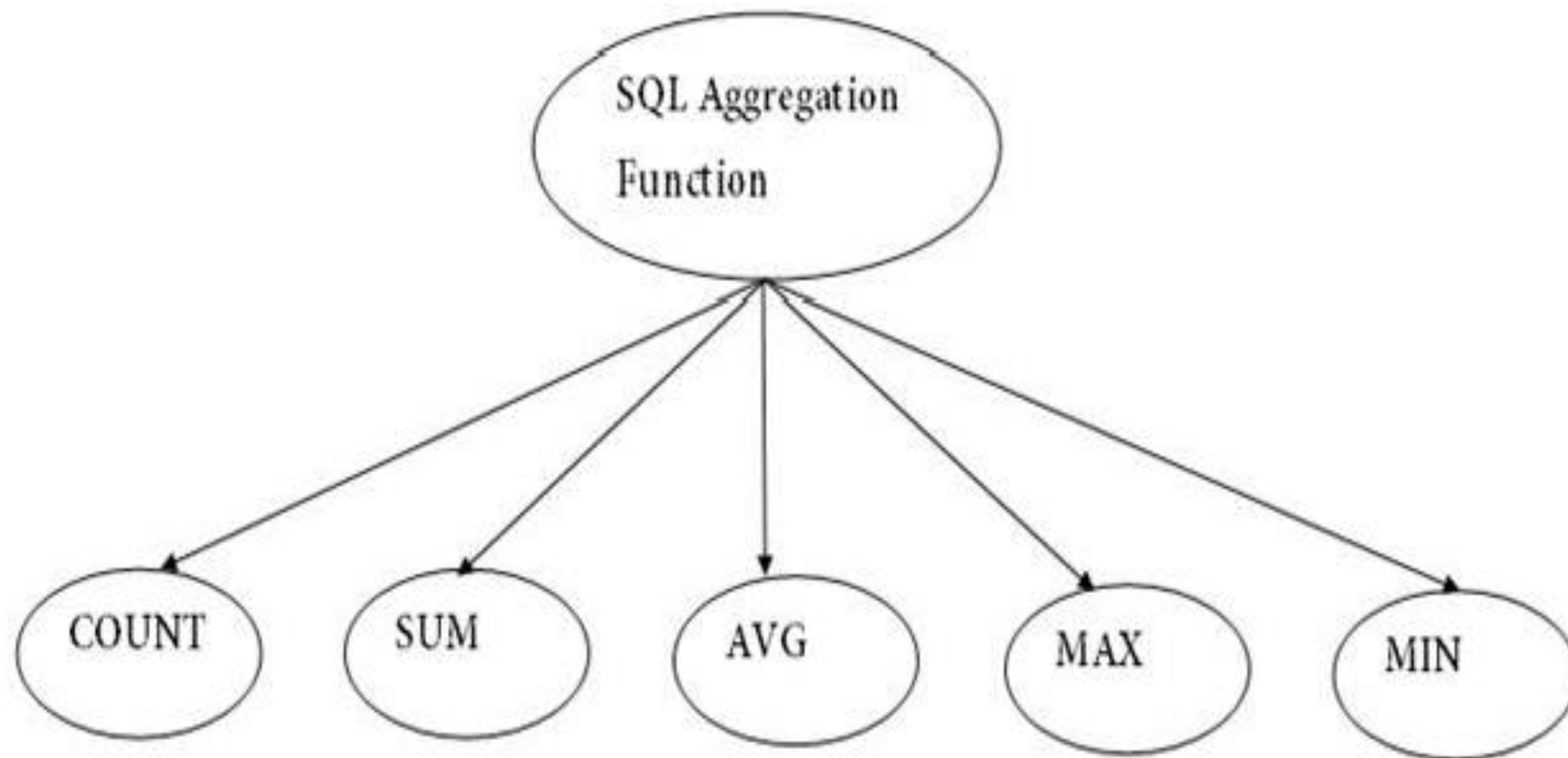
SELECT column1, column2FROM
table_name WHERE condition
ORDER BY column1, column2... ASC|DESC;

**Example**

SELECT *
FROM CUSTOMER ORDER BY
NAME;OR

SELECT *
FROM CUSTOMER ORDER BY NAME
DESC;

# SQL Aggregate Functions



# COUNT FUNCTION

- COUNT function is used to Count the number of rows in a databasetable. It can work on both numeric and non-numeric data types.
- COUNT function uses the COUNT(*) that returns the count of all therows in a specified table. COUNT(*) considers duplicate and Null.

**Syntax**

COUNT(*) or COUNT( [ALL|DISTINCT] expression )

**Example**

- SELECT COUNT(*) FROM PRODUCT_MAST;
- SELECT COUNT(*) FROM PRODUCT_MAST; WHERE RATE>=20;
- SELECT COUNT(DISTINCT COMPANY) FROM PRODUCT_MAST;
- SELECT COMPANY, COUNT(*) FROM PRODUCT_MAST GROUP BY COMPANY;
- SELECT COMPANY, COUNT(*) FROM PRODUCT_MAST GROUP BY COMPANYHAVING COUNT(*)>2;

# SUM FUNCTION

- Sum function is used to calculate the sum of all selectedcolumns. It works on numeric fields only.

**Syntax**

SUM() or SUM( [ALL|DISTINCT] expression )

**Example**

```
SELECT SUM(COST)   FROM PRODUCT_MAST;
```

**SUM() with WHERE**

```
SELECT SUM(COST)   FROM PRODUCT_MAST   WHERE QTY>3;
```

**SUM() with GROUP BY**

```
SELECT SUM(COST)   FROM PRODUCT_MAST   WHERE QTY>3
GROUP BY COMPANY;
```

**SUM() with HAVING**

```
SELECT COMPANY, SUM(COST)   FROM PRODUCT_MAST   GROUP BY COMPANY   HAVING SUM(COST)>=170;
```

# AVG FUNCTION

- The AVG function is used to calculate the average value of thenumeric type. AVG function returns the average of all non-Null values.

**Syntax**

AVG() or AVG( [ALL|DISTINCT] expression )

**Example**

SELECT AVG(COST) FROM PRODUCT_MAST;

# MAX FUNCTION

- MAX function is used to find the maximum value of a certain column. Thisfunction determines the largest value of all selected values of a column.

**Syntax**

MAX() or MAX( [ALL|DISTINCT] expression )

**Example**

SELECT MAX(RATE) FROM PRODUCT_MAST;

## MIN FUNCTION

- MIN function is used to find the minimum value of a certain column. Thisfunction determines the smallest value of all selected values of a column

**Syntax**

MIN() or MIN( [ALL|DISTINCT] expression )

**Example**

SELECT MIN(RATE) FROM PRODUCT_MAST;

## SQL JOIN

SQL, JOIN means "to combine two or more tables". In SQL, JOIN clause is used tocombine the records from two or more tables in a database.

## Types of SQL JOIN

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN

## INNER JOIN

In SQL, INNER JOIN selects records that have matching values in both tables as long as the condition is satisfied. It returns the combination of all rows from boththe tables where the condition satisfies.

**Syntax**

```
SELECT table1.column1, table1.column2, table2.column1,....
FROM table1 INNER JOIN
table2
ON table1.matching_column = table2.matching_column;
```

**Example**

```
SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENTFROM EMPLOYEE
INNER JOIN PROJECT
ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;
```

## LEFT JOIN

The SQL left join returns all the values from left table and the matching valuesfrom the right table. If there is no matching join value, it will return NULL. **Syntax**

```
SELECT table1.column1, table1.column2, table2.column1,....
FROM table1 LEFT JOIN
table2
ON table1.matching_column = table2.matching_column;
```

**Example**

```
SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENTFROM EMPLOYEE
LEFT JOIN PROJECT
ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;
```

## RIGHT JOIN

In SQL, RIGHT JOIN returns all the values from the values from the rows of righttable and the matched values from the left table. If there is no matching in bothtables, it will return NULL.

**Syntax**

```
SELECT table1.column1, table1.column2, table2.column1,....
FROM table1 RIGHT JOIN
table2
ON table1.matching_column = table2.matching_column;
```

**Example**

```
SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENTFROM EMPLOYEE
RIGHT JOIN PROJECT
ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;
```

# FULL JOIN

In SQL, FULL JOIN is the result of a combination of both left and right outer join.Join tables have all the records from both tables. It puts NULL on the place of matches not found.

**Syntax**

```
SELECT table1.column1, table1.column2, table2.column1,....
FROM table1 FULL JOIN
table2

ON table1.matching_column = table2.matching_column;
```

**Example**

```
SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENTFROM EMPLOYEE

FULL JOIN PROJECT
ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;
```

# SQL Set Operation

The SQL Set operation is used to combine the two ormore SQL SELECT statements

**Types of Set Operation**

Union UnionAll
IntersectMinus

# Union Operation

- The SQL Union operation is used to combine the result of two or more SQLSELECT queries.
- In the union operation, all the number of datatype and columns must besame in both the tables on which UNION operation is being applied.
- The union operation eliminates the duplicate rows from its resultset.

**Syntax**

```
SELECT column_name FROM table1UNION

SELECT column_name FROM table2;
```

**Example**

```
SELECT * FROM FirstUNION

SELECT * FROM Second;
```

## Intersect Operation

- It is used to combine two SELECT statements. The Intersect operation returnsthe common rows from both the SELECT statements.
- In the Intersect operation, the number of datatype and columns must be thesame.
- It has no duplicates and it arranges the data in ascending order by default.

**Syntax**

```
SELECT column_name FROM table1INTERSECT
SELECT column_name FROM table2;
```

**Example**

```
SELECT * FROM FirstINTERSECT
SELECT * FROM Second;
```

## MINUSOperation

- It combines the result of two SELECT statements. Minus operator is used to display the rows which are present in the first query but absent in the secondquery.
- It has no duplicates and data arranged in ascending order by default.

**Syntax**

```
SELECT column_name FROM table1MINUS
SELECT column_name FROM table2;
```

**Example**

```
SELECT * FROM FirstMINUS
SELECT * FROM Second;
```