

S.Y.B.Sc. (CS)
PAPER - III, Section - I
Data Base Management Systems

UNIT I: Relational Model (15 Lectures)

(a) Overview: Overview of database management system , limitations of data processing environment, database approach, data independence, three level of abstraction, DBMS structure.

(b) Entity Relation Model: Entity, attributes, keys, relations, cardinality, participation, weak entities, ER diagram, Generalization, Specialization and aggregation, conceptual design with ER model, entity versus attribute, entity versus relationship, binary versus ternary relationship, aggregate versus ternary relationship.

(c) Relational Structure: Introduction to relational model, integrity constraints over relations.

(d) Schema refinement and Normal forms: Functional dependencies, first, second, third, and BCNF normal forms based on primary keys, lossless join decomposition.

UNIT II: Query Languages (15 Lectures)

(a) Relational Algebra: select and projection, Set operations like union, intersection, difference, cross product, Joins – conditional, equi join and natural joins, division, examples. Overview of relational Calculus.

(b) Creating and altering tables: Conversion of ER to relations with and without constraints; CREATE statement with constraints like KEY, CHECK, DEFAULT, ALTER and DROP statement.

(c) Handling data using SQL: selecting data using SELECT statement, FROM clause, WHERE clause, HAVING clause, ORDER BY, GROUP BY, DISTINCT and ALL predicates, Adding data with INSERT statement, changing data with UPDATE statement, removing data with DELETE statement

(d) Functions: Aggregate functions-AVG, SUM, MIN, MAX and COUNT, Date functions-DATEADD(), DATEDIFF(), GETDATE(), DATENAME()YEAR, MONTH, WEEK, DAY, String functions-LOWER(), UPPER(), TRIM(), RTRIM(), PATINDEX(), REPLICATE(), REVERSE(), RIGHT(), LEFT()

(e) Joining tables: Inner, outer and cross joins, union.

(f) Sub queries: sub queries with IN, EXISTS, sub queries restrictions, Nested sub queries, correlated sub queries, queries with modified comparison operations, SELECT INTO operation, UNION operation. Sub queries in the HAVING clause.

Unit III: Implementing Indexes, Views and procedures (15 Lectures)

(a) File Organization and Indexing: Cluster, Primary and secondary indexing, Index data structure: hash and Tree based indexing, Comparison of file organization: cost model, Heap files, sorted files, clustered files. Creating, dropping and maintaining indexes using SQL.

(b) Views: Meaning of view, Data independence provided by views, creating, altering dropping, renaming and manipulating views using SQL.

(c) Stored Procedures: Types and benefits of stored procedures, creating stored procedures using SQL, executing stored procedures: Automatically executing stored procedures, altering stored procedures, viewing stored procedures.

(d) Triggers: Concept of triggers, Implementing triggers in SQL: creating triggers, Insert, delete, and update triggers, nested triggers, viewing, deleting and modifying triggers, and enforcing data integrity through triggers.



INTRODUCTION TO DATABASE MANAGEMENT SYSTEM

Unit Structure

- 1.0 Objectives
- 1.1 What is a Database?
- 1.2 What is a DBMS?
- 1.3 Why a DBMS?
- 1.4 Files vs. DBMS
- 1.5 Advantages of using a DBMS
- 1.6 Database Approach
- 1.7 Data Independence
- 1.8 DBMS Structure
- 1.9 Levels of Data Abstraction
- 1.10 Unit End Exercise
- 1.11 Further Reading and References

1.1 OBJECTIVES

In this chapter we will look at the definition of a database, a database management system and its need. We will further look at how a database management system differs from a normal file processing system. Further we will study the advantages and structure of a DBMS. We will also study what is does one mean by the term data independence and the different levels of data abstraction

1.1 WHAT IS A DATABASE?

A database is a collection of information or data organized and presented to serve a specific purpose. (You could take the

example of a telephone book as a common database.) A computerized database is an updated, organized file of machine readable information that is rapidly searched and retrieved by computer. Data are binary computer representations of stored entities. Data can exist in many forms. Data can be defined in many ways. Information science defines data as unprocessed information. Data is converted into information, and information is converted into knowledge. For the purposes of Enterprise, data is a small unit of information, i.e. a learner's name or an exam mark.

1.2 WHAT IS A DBMS?

Database Management System is a collection of computer programs that allow storage, modification, and extraction of information from a database. There are many different types of DBMSs, ranging from small systems that run on personal computers to huge systems that run on mainframes. The following are examples of database applications: computerized library systems, automated teller machines, banking systems, flight reservation systems, computerized parts inventory systems MGI. The examples of DBMS are MS-ACCESS, ORACLE, SQL SERVER etc.

1.3 WHY A DBMS?

1. Providing data independence and efficient access of data.
2. Reducing the application development time.
3. Providing Data integrity and security.
4. Uniform data administration

Check Your Progress!!!

1. List the different DBMS applications,
2. Name any four DBMS.
3. State the need of DBMS.

1.4 FILES VS. DBMS

Database management systems were developed to handle the following difficulties of typical file-processing systems supported by conventional operating systems.

1. Data redundancy and inconsistency
2. Difficulty in accessing data
3. Data isolation - multiple files and formats
4. Integrity problems
5. Atomicity of updates
6. Concurrent access by multiple users
7. Security problems

1.5 ADVANTAGES OF USING A DBMS

1. **Data independence:** Database application programs are independent of the details of data representation and storage. The conceptual and external schemas provide independence from physical storage decisions and logical design decisions respectively.
2. **Efficient access:** A DBMS provides efficient storage and retrieval mechanisms, including support for very large files, index structures and query optimization.
3. **Reduced application development time:** Since the DBMS provides several important functions required by applications, such as concurrency control and crash recovery, high level query facilities, etc., only application-specific code needs to be written. Even this is facilitated by suites of application development tools available from vendors for many database management systems.
4. **Data integrity and security:** The view mechanism and the authorization facilities of a DBMS provide a powerful access control mechanism. Further, updates to the data that violate the semantics of the data can be detected and rejected by the DBMS if users specify the appropriate integrity constraints.
5. **Data administration:** By providing a common umbrella for a large collection of data that is shared by several users, a DBMS facilitates maintenance and data administration tasks. A good DBA can effectively shield end-users from the chores of fine-tuning the data representation, periodic back-ups etc.
6. **Concurrent access:** A DBMS supports the notion of a transaction, which is conceptually a single user's sequential

program. Users can write transactions as if their programs were running in isolation against the database. The DBMS executes the actions of transactions in an interleaved fashion to obtain good performance, but schedules them in such a way as to ensure that conflicting operations are not permitted to proceed concurrently.

- 7. Crash recovery:** The DBMS maintains a continuous log of the changes to the data, and if there is a system crash, it can restore the database to a transaction-consistent state. That is, the actions of incomplete transactions are undone, so that the database state reflects only the actions of completed transactions.

1.6 DATABASE APPROACH

A database is more than a file – it contains information about more than one entity and information about relationships among the entities.

Data about a single entity (e.g., Product, Customer, Customer Order, Department) are each stored to a “table” in the database.

Databases are designed to meet the needs of multiple users and to be used in multiple applications.

One significant development in the more user-friendly relational DBMS products is that users can sometimes get their own answers from the stored data by learning to use data querying methods.

Check Your Progress!!!

1. What shortcomings of a file processing system are overcome by a DBMS?
2. How does a DBMS overcome from a system crash?

1.7 DATA INDEPENDENCE

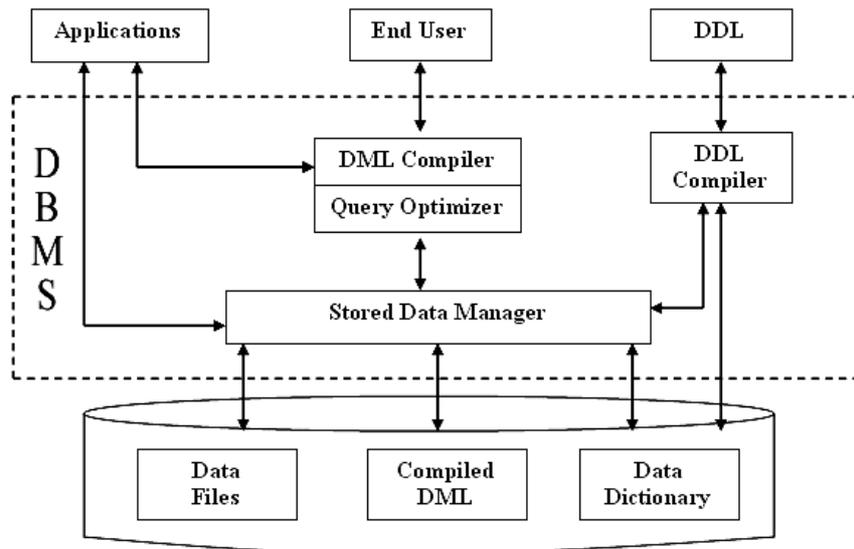
It is the ability to modify a schema definition in one level without affecting a schema definition in the next higher level. The interfaces between the various levels and components should be

well defined so that changes in some parts do not seriously influence others.

Two levels of data independence:

- Physical data independence
- Logical data independence

1.8 DBMS STRUCTURE



DBMS (Database Management System) acts as an interface between the user and the database. The user requests the DBMS to perform various operations (insert, delete, update and retrieval) on the database. The components of DBMS perform these requested operations on the database and provide necessary data to the users. The various components of DBMS are:

1. **DDL Compiler** – Data Description Language compiler processes schema definitions specified in the DDL. It includes metadata information such as the name of the files, data items, storage details of each file, mapping information and constraints etc.
2. **DML Compiler and Query optimizer** – The DML commands such as insert, update, delete, retrieve from the application program are sent to the DML compiler for compilation into object code for database access.
3. **Data Manager** – The Data Manager is the central software component of the DBMS also known as Database Control System. The Main Functions Of Data Manager Are:

- a. Convert operations in user's Queries coming from the application programs or combination of DML Compiler and Query optimizer which is known as Query Processor from user's logical view to physical file system.
- b. Controls DBMS information access that is stored on disk.
- c. It also controls handling buffers in main memory.
- d. It also enforces constraints to maintain consistency and integrity of the data.
- e. It also synchronizes the simultaneous operations performed by the concurrent users.
- f. It also controls the backup and recovery operations.

4. Data Dictionary – Data Dictionary is a repository of description of data in the database. It contains information about

- a. Data - names of the tables, names of attributes of each table, length of attributes, and number of rows in each table.
- b. Relationships between database transactions and data items referenced by them which is useful in determining which transactions are affected when certain data definitions are changed.
- c. Constraints on data i.e. range of values permitted.
- d. Detailed information on physical database design such as storage structure, access paths, files and record sizes.
- e. Access Authorization - is the Description of database users their responsibilities and their access rights.
- f. Usage statistics such as frequency of query and transactions.
- g. Data dictionary is used to actually control the data integrity, database operation and accuracy. It may be used as a important part of the DBMS.

5. Data Files – It contains the data portion of the database.

6. Compiled DML – The DML compiler converts the high level Queries into low level file access commands known as compiled DML.

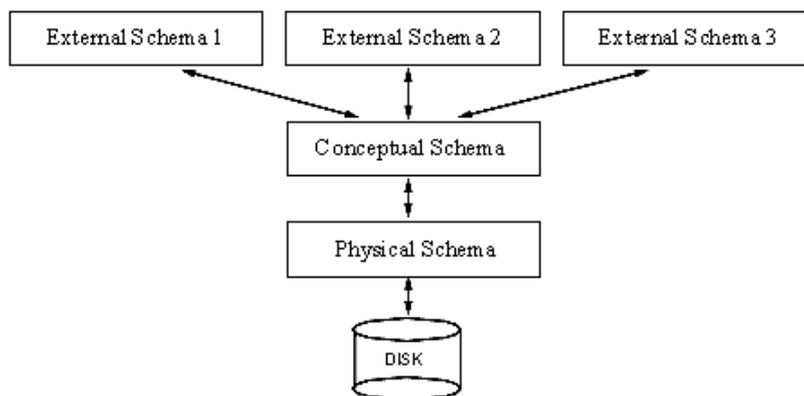
7. End Users – They are users who use the database, directly or indirectly.

1.9 LEVELS OF DATA ABSTRACTION

A database is considered as containing data about an enterprise. The three levels of the architecture are three different views of the data:

1. **External** – individual user view
2. **Conceptual** – community user view
3. **Physical** – physical or storage view

The three level database architecture allows a clear separation of the information meaning (conceptual view) from the external data representation and from the physical data structure layout. A database system that is able to separate the three different views of data is likely to be flexible and adaptable. This flexibility and adaptability is data independence that we have discussed earlier.



External Schema:

The external level is the view that the individual user of the database has. This view is often a restricted view of the database and the same database may provide a number of different views for different classes of users. In general, the end users and even the applications programmers are only interested in a subset of the database. For example, a department head may only be interested in the departmental finances and student enrolments but not the library information. The librarian would not be expected to have any interest in the information about academic staff. The payroll office would have no interest in student enrolments.

Conceptual Schema:

The conceptual view is the information model of the enterprise and contains the view of the whole enterprise without any concern for the physical implementation. This view is normally

more stable than the other two views. In a database, it may be desirable to change the internal view to improve performance while there has been no change in the conceptual view of the database. The conceptual view is the overall community view of the database and it includes all the information that is going to be represented in the database. The conceptual view is defined by the conceptual schema which includes definitions of each of the various types of data.

The internal view is the view about the actual physical storage of data. It tells us what data is stored in the database and how. At least the following aspects are considered at this level:

1. Storage allocation e.g. B-trees, hashing etc.
2. Access paths e.g. specification of primary and secondary keys, indexes and pointers and sequencing.
3. Miscellaneous e.g. data compression and encryption techniques, optimization of the internal structures.

Physical Schema:

Efficiency considerations are the most important at this level and the data structures are chosen to provide an efficient database. The internal view does not deal with the physical devices directly. Instead it views a physical device as a collection of physical pages and allocates space in terms of logical pages.

The three levels together:

The separation of the conceptual view from the physical view enables us to provide a logical description of the database without the need to specify physical structures. This is often called **physical data independence**. Separating the external views from the conceptual view enables us to change the conceptual view without affecting the external views. This separation is sometimes called **logical data independence**.

Assuming the three level view of the database, a number of mappings are needed to enable the users working with one of the external views. For example, the payroll office may have an external view of the database that consists of the following information only:

1. Staff number, name and address.
2. Staff tax information e.g. number of dependents.
3. Staff bank information where salary is deposited.
4. Staff employment status, salary level, leaves information etc.

The conceptual view of the database may contain academic staff, general staff, casual staff etc. A mapping will need to be created where all the staff in the different categories is combined into one category for the payroll office. The conceptual view would include information about each staff's position, the date employment started, full-time or part-time, etc. This will need to be mapped to the salary level for the salary office. Also, if there is some change in the conceptual view, the external view can stay the same if the mapping is changed.

Check Your Progress!!!

1. State the two levels of data independence.
2. State and explain the main functions of data manager.
3. What information is stored in a data dictionary?

1.10 UNIT END EXERCISE

1. Define: data, DBMS
2. What are the characteristics of data in a database?
3. What are the advantages of using a DBMS?
4. What are the different types of Data Models?
5. Explain the 3 different views or architecture of the data.
6. With the help of a neat labeled diagram explain the structure of a DBMS.

1.11 FURTHER READING AND REFERENCES

Database Management Systems – Ramakrishnam, Gehrke , McGraw- Hill.

Fundamentals of Database Systems – Elmasri and Navathe, Pearson Education

Database Systems, Design, Implementation and Management – Peter Rob and Coronel, Thomson Learning

A First Course in Database Systems, Jeffrey D. Ullman, Jennifer Widom, Pearson Education



THE ENTITY RELATIONSHIP MODEL

Unit Structure

2.0 Objectives

2.1 Introduction

2.2 Entities and Entity Sets

2.3 Relationships, Roles and Relationship Sets

2.4 Keys

2.5 Degree of a Relationship

2.6 Cardinality Ratio

2.7 Weak Entity type

2.8 The ER Diagrams

2.9 Specialization

2.10 Generalization

2.11 Aggregation

2.12 Entity vs. Attribute

2.13 Entity vs. Relationship

2.14 Unit End Exercise

2.15 Further Reading and References

2.0 OBJECTIVES

In this chapter we will study the entity relationship model of a database in detail. We will initially look at the concept of entities, relationships and the different types of keys. We will further look at the cardinality ratio of relationship types. We will further introduce the concept of ER – Diagrams and how to draw them. By the end of this chapter, we would study the concepts of Specialization, Generalization and Aggregation.

2.1 INTRODUCTION

The **E-R** (entity-relationship) data model views the real world as a set of basic **objects** (entities) and **relationships** among these objects. It is intended primarily for the database design process by allowing the specification of an **enterprise scheme**. This represents the overall logical structure of the database.

The entity-relationship model is useful because it facilitates communication between the database designer and the end user during the requirements analysis. To facilitate such communication the model has to be simple and readable for the database designer as well as the end user. It enables an abstract global view (that is, the enterprise conceptual schema) of the enterprise to be developed without any consideration of efficiency of the ultimate database.

The entity-relationship model views an enterprise as being composed of entities that have relationships between them. To develop a database model based on the E-R technique involves identifying the entities and relationships of interest. These are often displayed in an E-R diagram.

2.2 ENTITIES AND ENTITY SETS

An entity is a person, place, or a thing or an object or an event which can be distinctly identified and is of interest. A specific student, for example, John Smith with student number 84002345 or a subject Database Management Systems with subject number CP3010 or an institution called James Cook University are examples of entities. Although entities are often concrete like a company, an entity may be abstract like an idea, concept or convenience, for example, a research project P1234 - Evaluation of Database Systems or a Sydney to Melbourne flight number TN123. Entities are classified into different entity sets (or entity types). An entity set is a set of entity instances or entity occurrences of the same type. For example, all employees of a company may constitute an entity set employee and all departments may belong to an entity set Dept. An entity may belong to one or more entity sets. For example, some company employees may belong to employee as well as other entity sets, for example, managers. Entity sets therefore are not always disjoint.

An entity set or entity type is a collection of entity instances of the same type and must be distinguished from an entity instance.

The real world does not just consist of entities. Recall that a database is a collection of interrelated information about an enterprise. A database therefore consists of a collection of entity sets; it also includes information about relationships between the entity sets.

2.3 RELATIONSHIPS, ROLES AND RELATIONSHIP SETS

Relationships are associations or connections that exist between entities and may be looked at as mappings between two or more entity sets. A relation therefore is a subset of the cross products of the entity sets involved in the relationship. The associated entities may be of the same type or of different types. For example, working - for is a relationship between an employee and a company. Supervising is a relationship between two entities belonging to the same entity set (employee).

A relationship set R_i is a set of relations of the same type. It may be looked at as a mathematical relation among n entities each taken from an entity set, not necessarily distinct:

If E_1, E_2, \dots, E_n are entity sets, then a relationship set R is a **subset** of $\{(c_1, c_2, \dots, c_n) \mid c_1 \in E_1, c_2 \in E_2, \dots, c_n \in E_n\}$ where (c_1, c_2, \dots, c_n) is a relationship.

For example, a person entity in a relationship works-for has a role of an employee while the entity set company has a role of an employer.

Often role names are verbs and entity names are nouns which enables one to read the entity and relationships for example as "employee works-for employer".

Check Your Progress!!!

Define

- | | |
|-----------------|---------------------|
| 1. Entity | 2. Entity Set |
| 3. Relationship | 4. Relationship Set |

2.4 KEYS

Differences between two or more entities must be expressed in terms of attributes.

A **superkey** is a set of one or more attributes which; taken collectively, allow us to identify uniquely an entity in the entity set. For example, in the entity set *customer*, *customer-name* and *S.I.N.* is a superkey.

Note that *customer-name* alone is not, as two customers could have the same name.

A superkey may contain extraneous attributes, and we are often interested in the smallest superkey. A superkey for which no subset is a superkey is called a **candidate key**.

In the example above, *S.I.N.* is a candidate key, as it is minimal, and uniquely identifies a customer entity.

A **primary key** is a candidate key (there may be more than one) chosen by the database designer to identify entities in an entity set.

2.5 DEGREE OF A RELATIONSHIP

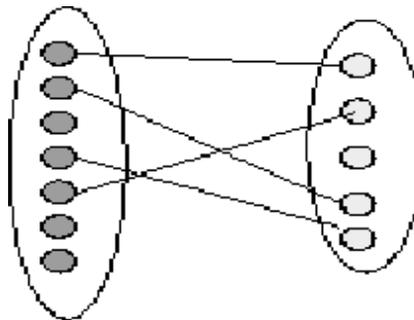
The degree of a relationship is the number of entities participating in the relationship. When we speak of a student registered for a course, we are discussing a relationship, register, where two entity sets (Student and Course) are involved; the relationship is of degree 2 because each instance of register will always involve one student entity and one course entity.

Relationships of degree 2 are called binary relationships; relationships of degree 3 are called ternary relationships. In general we speak of n-ary relationships where n entities participate in a relationship.

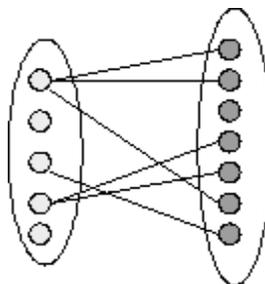
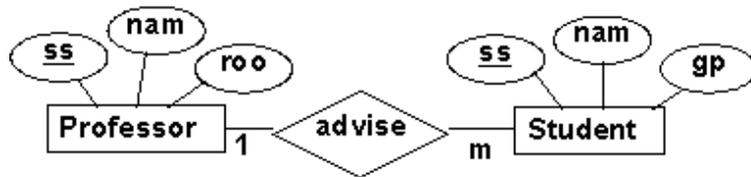
2.6 CARDINALITY RATIO

We will often be interested in the **cardinality** ratio of a relationship type, that is, how many of each entity type participate in the relationship. Possible cardinality ratios are the following.

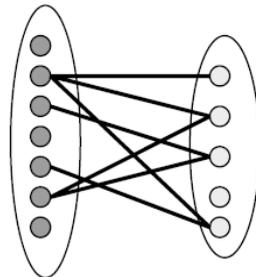
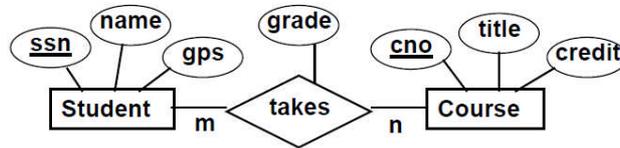
One-to-one (1-to-1): Each entity in E1 is associated with 0 or one entity in E2, and vice versa.



one-to-many :- A one-to-many relationship type (1-N or 1:N) is one in which a single entity of one entity type can be related to several entities of another type, but each entity of the other type is related to at most one entity of the first type (wow, that's a mouthful!).



many-to-many :- A many-to-many relationship type (N-M or N:M) is one in which a single entity of one entity type is related to at most N entities of another type, and vice-versa.



Participation Constraints and Existence Dependencies

The participation constraint specifies whether the existence of an entity depends on its being related to another entity via the relationship type.

The constraint specifies the minimum number of relationship instances that each entity can participate in.

There are two types of participation constraints:

- **Total**

If an entity can exist, only if it participates in at least one relationship instance, then that is called total participation, meaning that every entity in one set, must be related to at least one entity in a designated entity set.

An example would be the Employee and Department relationship. If company policy states that every employee must work for a department, then an employee can exist only if it participates in at least one relationship instance (i.e. an employee can't exist without a department). It is also sometimes called an existence dependency. Total participation is represented by a double line, going from the relationship to the dependent entity.

- **Partial:**

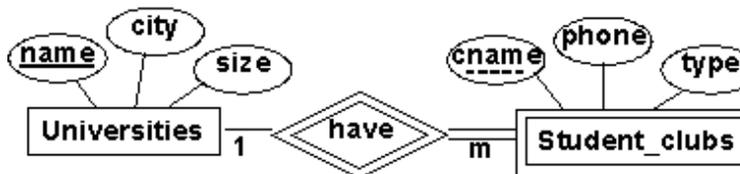
If only a part of the set of entities participate in a relationship, then it is called partial participation. Using the Company example, every employee will not be a manager of a department, so the participation of an employee in the “Manages” relationship is partial. Partial participation is represented by a single line.

Check Your Progress!!!

1. Define the terms, super key, primary key and candidate key.
2. State the different possible cardinality ratios.
3. Explain the two types of Participation Constraints.

2.7 WEAK ENTITY TYPE

A weak entity type is an entity that needs the key attributes from another entity to uniquely identify tuples. Weak entities lack keys. In an E/R diagram a weak entity type is represented by a nested pair of rectangles as shown below. The weak entity is connected by an identifying or owning relationship to the entity type that supplies the key attributes, which in turn is called the owning entity type. An owning relationship is depicted as a nested pair of diamonds.

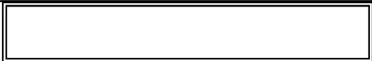
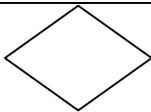
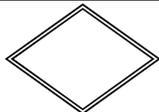
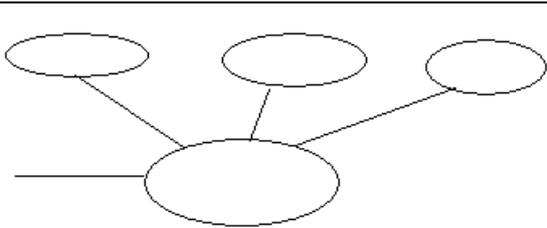


2.8 THE ER DIAGRAMS

As noted earlier, one of the aims of building an entity-relationship diagram is to facilitate communication between the database designer and the end user during the requirements analysis. To facilitate such communication, the designer needs adequate communication tools. Entity-Relationship diagrams are such tools that enable the database designer to display the overall database view of the enterprise (the enterprise conceptual schema).

An E-R diagram naturally consists of a collection of entity sets and relationship sets and their associations. A diagram may also show the attributes and value sets that are needed to describe the entity sets and the relationship sets in the ERD.

Symbols used in an E-R Diagrams

Symbol	Meaning
	ENTITY
	WEAK ENTITY
	RELATIONSHIP
	IDENTIFYING RELATIONSHIP
	ATTRIBUTE
	KEY ATTRIBUTE
	MULTIVALUED ATTRIBUTE
	COMPOSITE ATTRIBUTE

Steps in ER Modelling:

1. Identify the Entities
2. Find relationships
3. Identify the key attributes for every Entity
4. Identify other relevant attributes
5. Draw complete E-R diagram with all attributes including Primary Key

Check Your Progress!!!

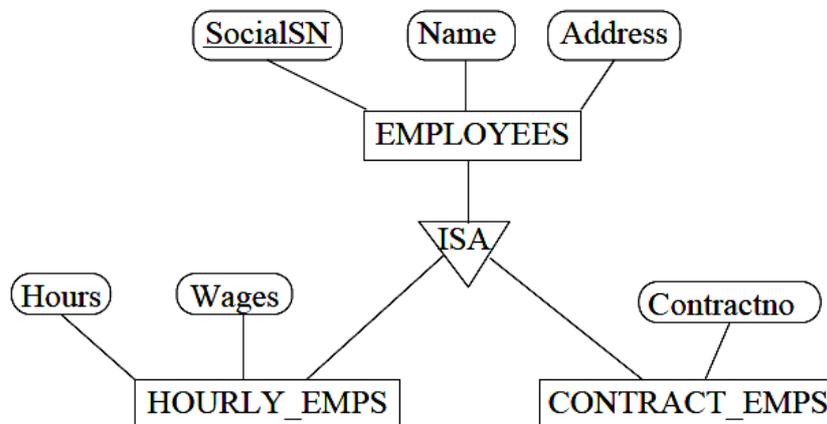
1. State the different symbols used to draw an ER Diagram.
2. State the different steps for drawing an ER Diagram.

2.9 SPECIALIZATION

An entity set may include subgroupings of entities that are distinct in some way from other entities in the set. For instance, a subset of entities within an entity set may have attributes that are not shared by all the entities in the entity set. The E-R model provides a means for representing these distinctive entity groupings. Consider an entity set employee, with attributes ssn, name, and address. An employee may be further classified as one of the following:

- Hourly Employee
- Contract Employee

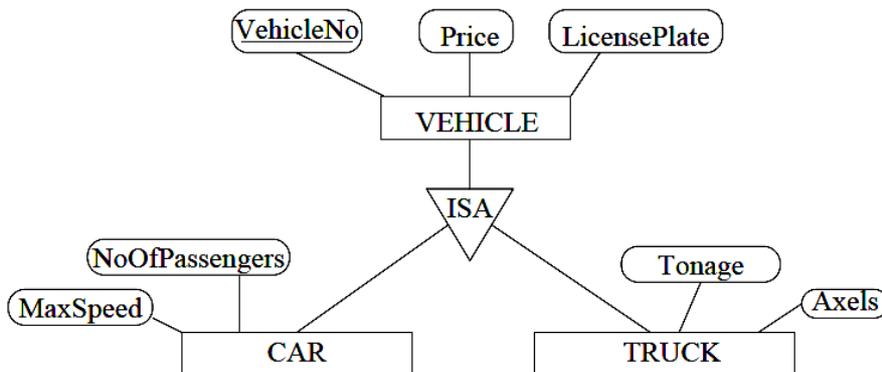
Each of these employee types is described by a set of attributes that includes all the attributes of entity set person plus possibly additional attributes. For example, hourly customer entities may be described further by the attribute hours and wages, whereas contract employee entities may be described further by the attribute contract number. The process of designating subgroupings within an entity set is called specialization. The specialization of person allows us to distinguish among employees according to whether they are hourly employees or contract employees.



2.10 GENERALIZATION

The design process may also proceed in a bottom-up manner, in which multiple entity sets are synthesized into a higher-level entity set on the basis of common features. The database designer may have first identified a car entity set with the attributes vehicle no, price, license plate, max speed and no of passengers, and a truck entity set with the attributes vehicle no, price, license plate, tonage and axles. There are similarities between the car entity set and the truck entity set in the sense that they have several attributes in common. This commonality can be expressed by generalization, which is a containment relationship that exists between a higher-level entity set and one or more lower-level entity sets. In our example, person is the higher-level entity set and customer and employee are lower-level entity sets.

Higher- and lower-level entity sets also may be designated by the terms superclass and subclass, respectively. The person entity set is the superclass of the customer and employee subclasses. For all practical purposes, generalization is a simple inversion of specialization.

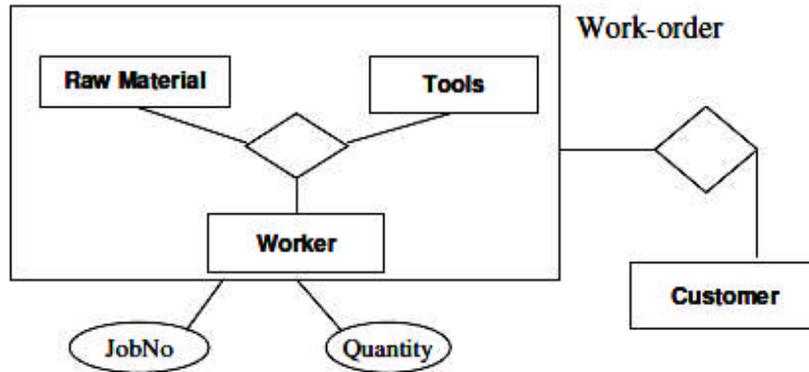


2.11 AGGREGATION:

Aggregation is an abstraction in which relationship sets (along with their associated entity sets) are treated as higher-level entity sets, and can participate in relationships. It treats relationship as an entity. It can be used to build complex entities from existing entities. There are two ways of defining complex entities:

1. create an attribute whose value is another entity
2. define an entity as containing a group of related entities

For example, in the ER diagram below, the work order entity, in turn consists of the raw material, tools and workers entities. The work order entity is itself related to the customer entity.



Check Your Progress!!!

State an example for each of the following:

1. Specialization
2. Generalization
3. Aggregation

2.12 ENTITY VS. ATTRIBUTE:

While identifying the attributes of an entity set, it is sometimes not clear whether a property should be modeled as an attribute or as an entity set. For example, the phone number of an employee.

One option is to use the attribute *phone_no*. This is suitable if we only have to store one phone number per employee.

Another option is to create an entity set called *phone_nos* and have a relationship with employees. This is useful in two cases

1. We have to record more than one phone number.
2. We have to capture the structure of the phone number like country code, city code, number etc. This can be useful to conduct searches like all the phone numbers of a particular city.

2.13 ENTITY VS. RELATIONSHIP

Consider a relationship manager between two entity sets, employees and departments. (Some of the employees manage some departments). Suppose that each manager is given some

budget to manage a department. In this case, given a department we know the manager (Assuming a key constraint) and also the budget given. But what if the budget is the sum that covers all the departments of the manager? In this case budget is the attribute of a specific manager and hence we must create a new entity set called managers which is a subclass of the employee entity and then associate the budget with the new entity set.

2.14 UNIT END EXERCISE

1. Define with examples, entity, entity set, relationship, relationship set and key.
2. What are the different possible cardinality ratios?
3. Explain the symbols used in an ER Diagram.
4. Explain Generalization, Specialization and Aggregation.
5. Explain the steps involved in ER Modelling.

2.15 FURTHER READING AND REFERENCES

Database Management Systems – Ramakrishnam, Gehrke , McGraw- Hill.

Fundamentals of Database Systems – Elsmasri and Navathe, Pearson Education

Database Systems, Design, Implementation and Management – Peter Rob and Coronel, Thomson Learning

A First Course in Database Systems, Jeffrey D. Ullman, Jennifer Widom, Pearson Education



RELATIONAL MODEL

Unit Structure

3.0 Introduction

3.1 Relation

3.2 The Relational data structure

3.3 Relational data Integrity

3.4 Integrity Constraints

3.5 Unit End Exercise

3.6 Further Reading and References

3.0 INTRODUCTION

In this chapter, we will study the concepts of relation, tuples and attributes. We will further look at the meaning of the term integrity and the various integrity constraints.

3.1 RELATION

A relation is a set of tuples. A database is a collection of relations. A relation is a Mathematical entity corresponding to a table. Each row in a table represents a fact that corresponds to an entity or a relationship that exists. Each row is called a tuple. Formally, the column headings of the table are the attributes of a relation. Each attribute must be atomic. Each attribute has a domain. The domain must be a simple data type, including for convenience strings, enumeration, dates, and sub range types. All tuples in a relation have the same structure; constructed from the same set of attributes.

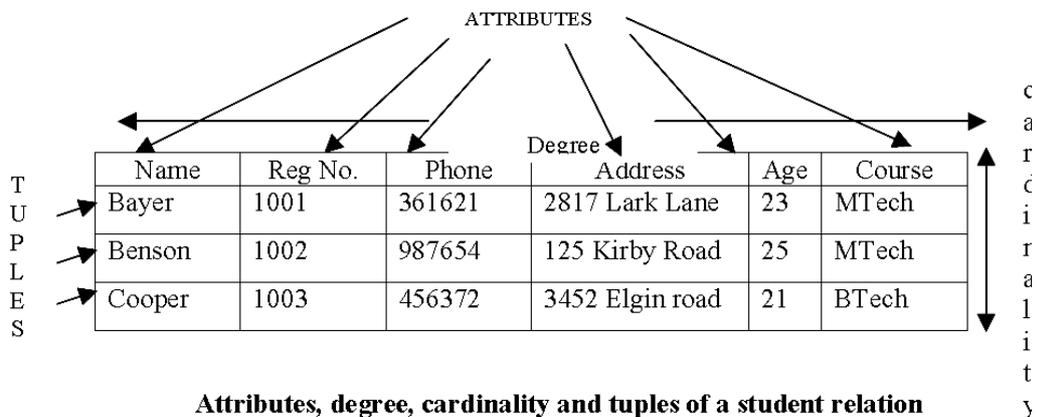
- row ~ tuple
- column ~ attribute

Values in a tuple are related to each other. Relation R can be thought of as a predicate R
 $R(x, y, z)$ is true if tuple (x, y, z) is in R.

3.2 THE RELATIONAL DATA STRUCTURE

The smallest unit of data in the relational model is the individual data value. Such values are assumed to be atomic, which means that they have no internal structure as far as the model is concerned. A domain is a set of all possible data values. For example in supplier parts example, the domain of supplier numbers is the set of all valid supplier numbers. Thus domains are pools of values, from which the actual values appearing the attributes are drawn. The domain concept is a very important and integral part of relational model.

A relation schema R , is denoted by $R(A_1, A_2, \dots, A_n)$, is made up of a relation name R and a list of attributes A_1, A_2, \dots, A_n . Each attribute A_i is the name of a role played by some domain D in the relation schema R . D is called the domain of A_i and is denoted by $\text{dom}(A_i)$. A relation schema is used to describe a relation; R is called the name of this relation. The degree of a relation is the number of attributes n of its relation schema. The figure shown below is an example of a STUDENT relation.



The earlier definition of a relation can be restated more formally as follows. A relation (or relation state) $r(R)$ is a mathematical relation of degree n on the domain $\text{dom}(A_1), \text{dom}(A_2), \dots, \text{dom}(A_n)$, which is a subset of Cartesian product of the domains that define R :

$$r(R) \subseteq (\text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_n))$$

The Cartesian product specifies all combinations of values from the underlying domains. Hence, if we denote the total number of values, or cardinality, in domain D by $|D|$ (assuming that all

domains are finite), the total number of tuples in the Cartesian product is

$$|dom(A_1)| \times |dom(A_2)| \times \dots \times |dom(A_n)|$$

So we can think relation as a table, then a **tuple** corresponds to a row of the table; the number tuples is called the **cardinality**; the number of attributes is called the **degree**; and a domain is a pool of values, from which the values of specific attributes of specific relation are taken.

Check Your Progress!!!

1. Define the terms, relation, tuple and attribute.
2. Give the mathematical definition of a relation.

3.3 RELATIONAL DATA INTEGRITY

As you know most of the relations have an attribute, which can uniquely identify each tuple in the relation. In some cases there can be more than one attribute, which can uniquely identify each tuple in the relation. This attribute is called as a **candidate key**. If there are more than one attribute both of the attributes are eligible to be identified as a candidate key. One of the candidate keys is arbitrarily designated to be the primary key and others are called as secondary or alternate keys. A key is minimal set of attributes guaranteeing separation for the members of the relation. When more than one key exists, a **primary key** is selected.

Element_Table				
Symbol	Name	Atomic_Number	Melting_Point	Boiling_Point
Al	Aluminum	13	933	2792
Fe	Iron	26	1811	3134
Ag	Silver	47	1235	2435

In the above table symbol, name and atomic number can uniquely identify each row, so any one can be a candidate key, or the Element_Table has three candidate keys. Let R be the relation with attributes A_1, A_2, \dots, A_n . The set of attributes $K=(A_i, A_j, \dots, A_n)$ of R is said to be a candidate key of R if and only if the following two properties are satisfied:

- **Uniqueness** – At any given point of time, no two distinct tuples of R have the same value of A_i , the same value for A_j, \dots and the same value for A_n .

- **Minimality** – No proper subset of the set (A_i, A_j, \dots, A_n) has the uniqueness property.

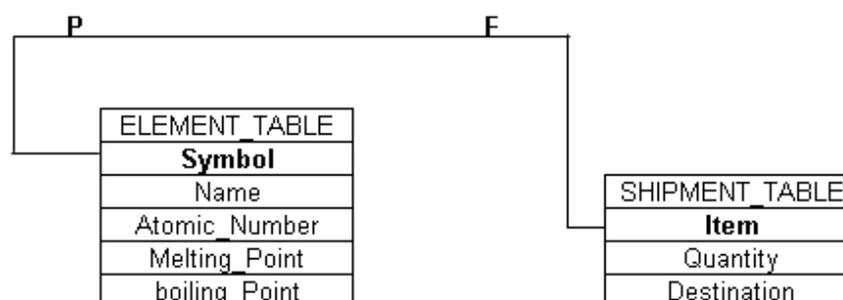
In the Element_Table relation there are three candidate keys, so we can choose any one of them as the primary key. There are no hard and fast rules on how to choose the primary key from the list of candidate keys. It is a matter of preference and convenience of database designer.

Let us take a look at another relation, SHIPMENT_TABLE.

Shipment_Table		
Item	Quantity	Destination
Al	100	Delhi
Fe	400	Kochi
Ag	300	Calcutta

In the ELEMENT_TABLE, the attribute Symbol and in the SHIPMENT_TABLE the attribute Item has same data values. And it is clear that a given value for that attribute, say Item 'Ag' should be permitted to appear in the database only if the same value appears as a value of the Primary Key 'Symbol' in the relation ELEMENT_TABLE..

Such an attribute is a **foreign key**. A foreign key is an attribute or attribute combination of one relation whose values are required to match those out of the primary key of some other relation. Also the foreign key and the primary key should be defined on the same underlying domain.



3.4 INTEGRITY CONSTRAINTS

Relational model includes several types of constraints whose purpose is to maintain the accuracy and integrity of the data in the database. The major types of integrity constraints are:

- Domain Constraints
- Entity Integrity
- Referential Integrity
- Operational Constraints

Domain Constraints

All the values that appear in a column of a relation must be taken from the same domain. A domain usually consists of the following components.

1. Domain Name
2. Meaning
3. Data Type
4. Size or length
5. Allowable values or Allowable range(if applicable)

Entity Integrity

The Entity Integrity rule is so designed to assure that every relation has a primary key and that the data values for the primary key are all valid. Entity integrity guarantees that every primary key attribute is non null. No attribute participating in the primary key of a base relation is allowed to contain nulls. Primary key performs unique identification function in a relational model. Thus a null primary key performs the unique identification function in a relation would be like saying that there are some entity that had no known identity. An entity that cannot be identified is a contradiction in terms, hence the name entity integrity.

Referential Integrity

In the relational model the association between the tables is defined using foreign keys. The association between the SHIPMENT and ELEMENT tables is defined by including the Symbol attribute as a foreign key in the SHIPMENT table. This implies that before we insert a row in the SHIPMENT table, the element for that order must already exist in the ELEMENT table. A referential integrity constraint is a rule that maintains consistency among the rows of two tables or relations. The rule states that if there is a foreign key in one relation, either each of the foreign key

value must match a primary key value in the other table or else the foreign key value must be null.

Operational Constraints

These are the constraints enforced in the database by the business rules or real world limitations. For example if the retirement age of the employees in a organization is 60, then the age column of the employee table can have a constraint “Age should be less than or equal to 60”. These kinds of constraints enforced by the business and the environment are called operational constraints.

Check Your Progress!!!

1. When can a set of attributes be said a candidate key of a relation?
2. Give some examples of operational constrains on the student relation.

3.5 UNIT END EXERCISE

1. Define the term Relation.
2. Explain the Relational Data Structure.
3. Explain the different Integrity Constraints.

3.6 FURTHER READING AND REFERENCES

Database Management Systems – Ramakrishnam, Gehrke , McGraw- Hill.

Fundamentals of Database Systems – Elsmasri and Navathe, Pearson Education

Database Systems, Design, Implementation and Management – Peter Rob and Coronel, Thomson Learning

A First Course in Database Systems, Jeffrey D. Ullman, Jennifer Widom, Pearson Education



SCHEMA REFINEMENT AND NORMAL FORMS

Unit Structure

- 4.0 Objectives
- 4.1 Functional Dependencies
- 4.2 Identifying Functional Dependencies
- 4.3 Inference Rules for Functional Dependencies
- 4.4 Analysis of Redundancies
- 4.5 What is Normalization?
- 4.6 Normal Forms
- 4.7 Lossless-Join Decomposition
- 4.8 Unit End Exercise
- 4.9 Further Reading and References

4.0 OBJECTIVES

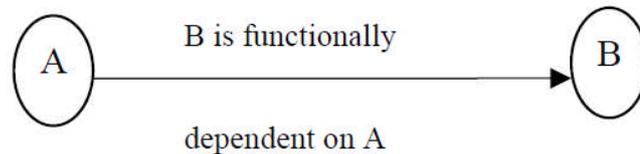
In this chapter we will study the meaning of the term functional dependencies, how to identify them and the inference rules for functional dependencies. We would further look at data normalization and the different normal forms – 1NF, 2NF, 3NF and BCNF. By the end of this chapter we will study how larger tables can be broken into smaller ones without loss of data.

4.1 FUNCTIONAL DEPENDENCIES

For our discussion on functional dependencies assume that a relational schema has attributes (A, B, C... Z) and that the whole database is described by a single universal relation called $R = (A, B, C, \dots, Z)$. This assumption means that every attribute in the database has a unique name.

What is functional dependency in a relation?

A functional dependency is a property of the semantics of the attributes in a relation. The semantics indicate how attributes relate to one another, and specify the functional dependencies between attributes. When a functional dependency is present, the dependency is specified as a constraint between the attributes. Consider a relation with attributes A and B, where attribute B is functionally dependent on attribute A. If we know the value of A and we examine the relation that holds this dependency, we will find only one value of B in all of the tuples that have a given value of A, at any moment in time. Note however, that for a given value of B there may be several different values of A.



In the figure above, A is the determinant of B and B is the consequent of A.

The determinant of a functional dependency is the attribute or group of attributes on the left-hand side of the arrow in the functional dependency. The consequent of a functional dependency is the attribute or group of attributes on the right-hand side of the arrow.

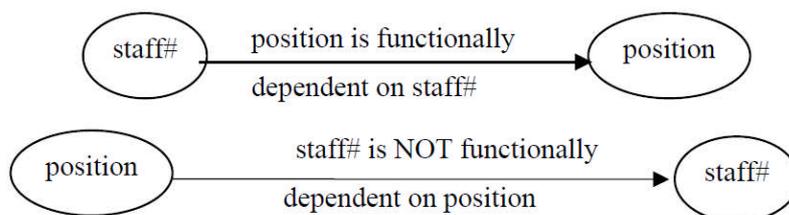
4.2 IDENTIFYING FUNCTIONAL DEPENDENCIES

Now let us consider the following Relational schema
STAFFBRANCH

staff#	sname	position	salary	branch#	baddress
SL21	Kristy	manager	30000	B005	22Deer Road
SG37	Debris	assistant	12000	B003	162Main Street
SG14	Alan	supervisor	18000	B003	163Main Street
SA9	Traci	assistant	12000	B007	375Fox Avenue
SG5	David	manager	24000	B003	163Main Street

The functional dependency $\text{staff\#} \rightarrow \text{position}$ clearly holds on this relation instance. However, the reverse functional dependency $\text{position} \rightarrow \text{staff\#}$ clearly does not hold.

The relationship between staff\# and position is 1:1 – for each staff member there is only one position. On the other hand, the relationship between position and staff\# is 1:M – there are several staff numbers associated with a given position.



For the purposes of normalization we are interested in identifying functional dependencies between attributes of a relation that have a 1:1 relationship.

When identifying functional dependencies between attributes in a relation it is important to distinguish clearly between the values held by an attribute at a given point in time and the set of all possible values that an attributes may hold at different times.

In other words, a functional dependency is a property of a relational schema (its intension) and not a property of a particular instance of the schema (extension).

The reason that we need to identify functional dependencies that hold for all possible values for attributes of a relation is that these represent the types of integrity constraints that we need to identify. Such constraints indicate the limitations on the values that a relation can legitimately assume. In other words, they identify the legal instances which are possible.

Let's identify the functional dependencies that hold using the relation schema STAFFBRANCH

In order to identify the time invariant functional dependencies, we need to understand the semantics of the various attributes in each of the relation schemas in question.

For example, if we know that a staff member's position and the branch at which they are located determines their salary. There is no way of knowing this constraint unless you are familiar with the enterprise, but this is what the requirements analysis phase and the conceptual design phase are all about!

staff# \rightarrow sname, position, salary, branch#, baddress

branch# \rightarrow baddress

baddress \rightarrow branch#

branch#, position \rightarrow salary

baddress, position \rightarrow salary

4.3 INFERENCE RULES FOR FUNCTIONAL DEPENDENCIES

We'll denote as F , the set of functional dependencies that are specified on a relational 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 that satisfy the dependencies in F .

These additional functional dependencies that hold are those functional dependencies which can be inferred or deduced from the functional dependencies in F .

The set of all functional dependencies implied by a set of functional dependencies F is called the closure of F and is denoted F^+ .

The notation: $F \ X \rightarrow Y$ denotes that the functional dependency $X \rightarrow Y$ is implied by the set of functional dependencies F .

Formally, $F^+ \equiv \{X \rightarrow Y \mid F \ X \rightarrow Y\}$

A set of inference rules is required to infer the set of functional dependencies in F^+ .

For example, if I tell you that Ajay is older than Sanjay and that Sanjay is older than Vijay, you are able to infer that Ajay is older than Vijay. How did you make this inference? Without thinking about it or maybe knowing about it, you utilized a transitivity rule to allow you to make this inference. The set of all functional

dependencies that are implied by a given set S of functional dependencies is called the closure of S , written S^+ . Clearly we need an algorithm that will allow us to compute S^+ from S . The following are the six well-known inference rules that apply to functional dependencies.

IR1: reflexive rule – if $X \supseteq Y$, then $X \rightarrow Y$

IR2: augmentation rule – if $X \rightarrow Y$, then $XZ \rightarrow YZ$

IR3: transitive rule – if $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

IR4: projection rule – if $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$

IR5: additive rule – if $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$

IR6: pseudo transitive rule – if $X \rightarrow Y$ and $YZ \rightarrow W$, then $XZ \rightarrow W$

The first three of these rules (IR1-IR3) are known as Armstrong's Axioms and constitute a necessary and sufficient set of inference rules for generating the closure of a set of functional dependencies. These rules can be stated in a variety of equivalent ways. Each of these rules can be directly proved from the definition of functional dependency. Moreover the rules are complete, in the sense that, given a set S of functional dependencies, all functional dependencies implied by S can be derived from S using the rules. The other rules are derived from these three rules.

Given $R = (A, B, C, D, E, F, G, H, I, J)$ and

$F = \{AB \rightarrow E, AG \rightarrow J, BE \rightarrow I, E \rightarrow G, GI \rightarrow H\}$

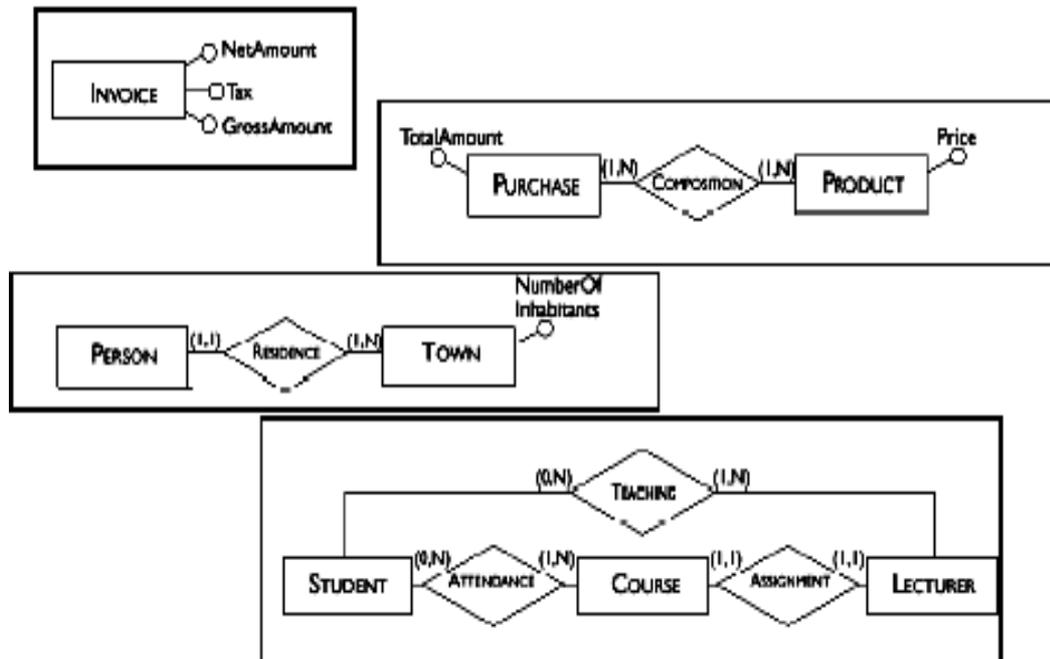
Does $F \models AB \rightarrow GH$?

Proof

1. $AB \rightarrow E$, given in F
2. $AB \rightarrow AB$, reflexive rule IR1
3. $AB \rightarrow B$, projective rule IR4 from step 2
4. $AB \rightarrow BE$, additive rule IR5 from steps 1 and 3
5. $BE \rightarrow I$, given in F
6. $AB \rightarrow I$, transitive rule IR3 from steps 4 and 5
7. $E \rightarrow G$, given in F
8. $AB \rightarrow G$, transitive rule IR3 from steps 1 and 7
9. $AB \rightarrow GI$, additive rule IR5 from steps 6 and 8
10. $GI \rightarrow H$, given in F
11. $AB \rightarrow H$, transitive rule IR3 from steps 9 and 10
12. $AB \rightarrow GH$, additive rule IR5 from steps 8 and 11 - proven

4.4 ANALYSIS OF REDUNDANCIES

A redundancy in a conceptual schema corresponds to a piece of information that can be derived (that is, obtained through a series of retrieval operations) from other data in the database.



Deciding About Redundancies

The presence of a redundancy in a database may be decided upon the following factors:

- **An advantage:** a reduction in the number of accesses necessary to obtain the derived information;
- **A disadvantage:** because of larger storage requirements, (but, usually at negligible cost) and the necessity to carry out additional operations in order to keep the derived data consistent.

The decision to maintain or delete a redundancy is made by comparing the cost of operations that involve the redundant information and the storage needed, in the case of presence or absence of redundancy.

Check Your Progress!!!

1. Define the term functional dependency.
2. State the six inference rules that apply to functional dependencies.
3. State an advantage and disadvantage of redundancy.

4.5 WHAT IS NORMALIZATION?

Normalization is a formal process for determining which fields belong in which tables in a relational database. Normalization follows a set of rules worked out at the time relational databases were born. A normalized relational database provides several benefits:

- Elimination of redundant data storage.
- Close modeling of real world entities, processes, and their relationships.
- Structuring of data so that the model is flexible.
- Normalization ensures that you get the benefits relational databases offer. Time spent learning about normalization will begin paying for itself immediately.

Terminology

There are a couple terms that are central to a discussion of normalization: “key” and “dependency”. These are probably familiar concepts to anyone who has built relational database systems, though they may not be using these words. We define and discuss them here as necessary background for the discussion of normal forms that follows.

4.6 NORMAL FORMS

4.6.1 1ST NORMAL FORM (1NF)

A table (relation) is in 1NF if

1. There are no duplicated rows in the table.
2. Each cell is single-valued (i.e., there are no repeating groups or arrays).
3. Entries in a column (attribute, field) are of the same kind.

Note: The order of the rows is immaterial; the order of the columns is immaterial. The requirement that there be no duplicated rows in the table means that the table has a key (although the key might be made up of more than one column—even, possibly, of all the columns).

So we come to the conclusion,

- A relation is in 1NF if and only if all underlying domains contain atomic values only.
- The first normal form deals only with the basic structure of the relation and does not resolve the problems of redundant information or the anomalies discussed earlier.

For example consider the following example relation is in 1NF:
student(sno, sname, dob).

A relation is in first normal form if and only if, in every legal value of that relation every tuple contains one value for each attribute.

The above definition merely states that the relations are always in first normal form which is always correct. However the relation that is only in first normal form has a structure those undesirable for a number of reasons.

First normal form (1NF) sets the very basic rules for an organized database:

- Eliminate duplicative columns from the same table.
- Create separate tables for each group of related data and identify each row with a unique column or set of columns (the primary key).

4.6.2 2ND NORMAL FORM (2NF)

A table is in 2NF if it is in 1NF and if all non-key attributes are dependent on the entire key.

The second normal form attempts to deal with the problems that are identified with the relation above that is in 1NF. The aim of second normal form is to ensure that all information in one relation is only about one thing.

A relation is in 2NF if it is in 1NF and every non-key attribute is fully dependent on each candidate key of the relation.

Note: Since a partial dependency occurs when a non-key attribute is dependent on only a part of the (composite) key, the definition of 2NF is sometimes phrased as, "A table is in 2NF if it is in 1NF and if it has no partial dependencies."

Recall the general requirements of 2NF:

- Remove subsets of data that apply to multiple rows of a table and place them in separate rows.
- Create relationships between these new tables and their predecessors through the use of foreign keys.

These rules can be summarized in a simple statement: 2NF attempts to reduce the amount of redundant data in a table by extracting it, placing it in new table(s) and creating relationships between those tables.

Let's look at an example. Imagine an online store that maintains customer information in a database. Their Customers table might look something like this:

CustNum	FirstName	LastName	Address	City	State	ZIP
1	John	Doe	12 Main Street	Sea Cliff	NY	11579
2	Alan	Johnson	82 Evergreen	Sea Cliff	NY	11579
3	Beth	Thompson	1912 NE 1st St	Miami	FL	33157
4	Jacob	Smith	142 Irish Way	South Bend	IN	46637
5	Sue	Ryan	412 NE 1st St	Miami	FL	33157

A brief look at this table reveals a small amount of redundant data. We're storing the "Sea Cliff, NY 11579" and "Miami, FL 33157" entries twice each. Now, that might not seem like too much added storage in our simple example, but imagine the wasted space if we had thousands of rows in our table. Additionally, if the ZIP code for Sea Cliff were to change, we'd need to make that change in many places throughout the database.

In a 2NF-compliant database structure, this redundant information is extracted and stored in a separate table. Our new table (let's call it ZIPs) might look like this:

ZIP	City	State
11579	Sea Cliff	NY
33157	Miami	FL
46637	South Bend	IN

We can even fill this table in advance -- the post office provides a directory of all valid ZIP codes and their city/state relationships. Surely, you've encountered a situation where this type of database was utilized. Someone taking an order might have asked you for your ZIP code first and then knew the city and state you were calling from. This type of arrangement reduces operator error and increases efficiency.

4.6.3 3RD NORMAL FORM (3NF)

A table is in 3NF if it is in 2NF and if it has no transitive dependencies.

The basic requirements of 3NF are as follows

- Meet the requirements of 1NF and 2NF
- Remove columns that are not fully dependent upon the primary key.

Imagine that we have a table of widget orders:

Order Number	Customer Number	Unit Price	Quantity	Total
1	241	\$10	2	\$20
2	842	\$9	20	\$180
3	919	\$19	1	\$19
4	919	\$12	10	\$120

Remember, our first requirement is that the table must satisfy the requirements of 1NF and 2NF. Are there any duplicative columns? No. Do we have a primary key? Yes, the order number. Therefore, we satisfy the requirements of 1NF. Are there any subsets of data that apply to multiple rows? No, so we also satisfy the requirements of 2NF.

Now, are all of the columns fully dependent upon the primary key? The customer number varies with the order number and it doesn't appear to depend upon any of the other fields. What about the unit price? This field could be dependent upon the customer number in a situation where we charged each customer a set price. However, looking at the data above, it appears we sometimes charge the same customer different prices. Therefore, the unit price is fully dependent upon the order number. The quantity of items also varies from order to order, so we're OK there.

What about the total? The total can be derived by multiplying the unit price by the quantity; therefore it's not fully dependent upon the primary key. We must remove it from the table to comply with the third normal form:

Order Number	Customer Number	Unit Price	Quantity
1	241	\$10	2
2	842	\$9	20
3	919	\$19	1
4	919	\$12	10

4.6.4 BOYCE-CODD NORMAL FORM (BCNF)

A relation is said to be in the BCNF if and only if it is in the 3NF and every non-trivial, left-irreducible functional dependency has a candidate key as its determinant. In more informal terms, a relation is in BCNF if it is in 3NF and the only determinants are the candidate keys.

Check Your Progress!!!

When is a relation said to be in a

1. 1NF
2. 2NF
3. 3NF
4. BCNF

4.7 LOSSLESS-JOIN DECOMPOSITION

So far we have normalized a number of relations by decomposing them. We decomposed a relation intuitively. We need a better basis for deciding decompositions since intuition may not always be correct. We illustrate how a careless decomposition may lead to problems including loss of information.

Consider the following relation

enrol (sno, cno, date-enrolled, room-No., instructor)

Suppose we decompose the above relation into two relations enrol1 and enrol2 as follows

enrol1 (sno, cno, date-enrolled) enrol2 (date-enrolled, room-No., instructor)

There are problems with this decomposition but we wish to focus on one aspect at the moment. Let an instance of the relation enroll be

sno	Cno	date-enrolled	room-No	instructor
830057	CP302	1 FEB 2004	MP006	Gupta
830057	CP303	1 FEB2004	MP006	Sharma
820159	CP302	10 JAN 2004	MP006	Gupta
825678	CP304	1 FEB 2004	CE122	Chopra
826789	CP305	15 JAN 2004	EA123	Shukla

and let the decomposed relations enrol1 and enrol2 be:

Sno	Cno	date-enrolled
830057	CP302	1 FEB 2004
830057	CP303	1 FEB2004
820159	CP302	10 JAN 2004
825678	CP304	1 FEB 2004
826789	CP305	15 JAN 2004

date-enrolled	room-No	instructor
1 FEB 2004	MP006	Gupta
1 FEB2004	MP006	Sharma
10 JAN 2004	MP006	Gupta
1 FEB 2004	CE122	Chopra
15 JAN 2004	EA123	Shukla

All the information that was in the relation enrol appears to be still available in enrol1 and enrol2 but this is not so. Suppose, we wanted to retrieve the student numbers of all students taking a course from Chopra, we would need to join enrol1 and enrol2. The join would have tuples as follows:

sno	Cno	date-enrolled	room-No	instructor
830057	CP302	1 FEB 2004	MP006	Gupta
830057	CP302	1 FEB2004	MP006	Sharma
830057	CP303	1 FEB 2004	MP006	Gupta
830057	CP303	1 FEB2004	MP006	Sharma
830057	CP302	1 FEB2004	CE122	Chopra
830057	CP303	1 FEB2004	EA123	Shukla
826789	CP305	15 JAN 2004	EA123	Shukla
.
.

The join contains a number of false tuples that were not in the original relation Enrol. Because of these additional tuples, we have lost the information about which students take courses from Chopra. (Yes, we have more tuples but less information because we are unable to say with certainty who is taking courses from Chopra). Such decompositions are called lossy decompositions. A nonloss or lossless decomposition is that which guarantees that the join will result in exactly the same relation as was decomposed. One might think that there might be other ways of recovering the original relation from the decomposed relations but, sadly, no other operators can recover the original relation if the join does not (why?).

We need to analyse why some decompositions are lossy. The common attribute in above decompositions was Date-enrolled. The common attribute is the clue that gives us the ability to find the relationships between different relations by joining the relations together. If the common attribute is not unique, the relationship information is not preserved. If each tuple had a unique value of Date-enrolled, the problem of losing information would not have existed. The problem arises because several enrolments may take place on the same date.

A decomposition of a relation R into relations R_1, R_2, \dots, R_n is called a lossless-join decomposition (with respect to FDs F) if the relation R is always the natural join of the relations R_1, R_2, \dots, R_n . It should be noted that natural join is the only way to recover the relation from the decomposed relations. There is no other set of operators that can recover the relation if the join cannot.

It is not difficult to test whether a given decomposition is lossless-join given a set of functional dependencies F . We consider the simple case of a relation R being decomposed into R_1 and R_2 . If the decomposition is lossless-join, then one of the following two conditions must hold:

- **Dependency Preservation**

It is clear that decomposition must be lossless so that we do not lose any information from the relation that is decomposed. Dependency preservation is another important requirement since a dependency is a constraint on the database and if holds then we know that the two (sets) attributes are closely related and it would be useful if both

attributes appeared in the same relation so that the dependency can be checked easily.

- **Lack of Redundancy**

We have discussed the problems of repetition of information in a database. Such repetition should be avoided as much as possible.

Lossless-join, dependency preservation and lack of redundancy is not always possible with BCNF. Lossless-join, dependency preservation and lack of redundancy is always possible with 3NF.

4.8 UNIT END EXERCISE

1. Explain the term Functional Dependency.
2. What is normalization?
3. Explain the different normal forms.
4. Compare Lossy and Lossless Decomposition.

4.9 FURTHER READING AND REFERENCES

Database Management Systems – Ramakrishnam, Gehrke , McGraw- Hill.

Fundamentals of Database Systems – Elmasri and Navathe, Pearson Education

Database Systems, Design, Implementation and Management – Peter Rob and Coronel, Thomson Learning

A First Course in Database Systems, Jeffrey D. Ullman, Jennifer Widom, Pearson Education



RELATIONAL ALGEBRA

Unit Structure

- 5.0 Introduction
- 5.1 Introduction
- 5.2 Operations in Relational Algebra
- 5.3 Selection Operation σ
- 5.4 PROJECT Operation π
- 5.5 RENAME Operation ρ
- 5.6 The Union Operation \cup
- 5.7 The Intersection Operation \cap
- 5.8 The Set Difference $-$
- 5.9 The Cartesian Product \times
- 5.10 Joins
- 5.11 Relational Calculus
- 5.12 Unit End Exercise
- 5.13 Further Reading and References

5.0 OBJECTIVES

In this chapter we will study the different relational algebra operators and their usage. We would further introduce relational calculus as an alternative to relational algebra.

5.1 INTRODUCTION

Relational algebra and relational calculus are formal languages associated with the relational model. Informally, relational algebra is a (high-level) procedural language and relational calculus a non-procedural language. However, formally both are equivalent to one another. A language that produces a relation that can be derived using relational calculus is relationally complete. Relational algebra operations work on one or more

relations to define another relation without changing the original relations. In relational algebra both operands and results are relations, so output from one operation can become input to another operation.

5.2 OPERATIONS IN RELATIONAL ALGEBRA

Basic Operations:

- Selection (σ): choose a subset of rows.
- Projection (π): choose a subset of columns.
- Renaming (ρ): change names of tables & columns

- Union (\cup): unique tuples from either table.

- Set difference ($-$): tuples in R1 not in R2.
- Cartesian Product (\times): Combine two tables.

Additional Operations:

- Intersection (\cap)
- Joins ()

5.3 SELECTION OPERATION σ

The select command gives a programmer the ability to choose tuples from a relation (rows from a table). The idea of the selection operation is to choose tuples of a relation (rows of a table).

Format:

σ selection . condition (R). Choose tuples that satisfy the selection condition.

E.g.: $\sigma_{\text{Major} = \text{'CS'}}$ (Students)

Student Table				Result			
SID	Name	GPA	Major	SID	Name	GPA	Major
456	Jay	3.4	CS	456	Jay	3.4	CS
457	Ajay	3.2	CS	457	Ajay	3.2	CS

Note: All the Relational Algebra select commands does choose tuples from a relation.

This means that, the desired output is to display the name of students who has taken CS as Major. The Selection condition is a Boolean expression including =, ≠, <, ≤, >, ≥, and, or, not.

5.4 PROJECT OPERATION Π

The Relational Algebra project command allows the programmer to choose attributes (columns) of a given relation and delete information in the other attributes. The idea of the projection operation is to choose certain attributes of a relation (columns of a table)

Format: $\Pi_{\text{Attribute List}}(\text{Relation})$

It returns a relation with the same tuples as (Relation) but limited to those attributes of interest (in the attribute list).selects some of the columns of a table; it constructs a vertical subset of a relation; implicitly removes any duplicate tuples (so that the result will be a relation).

E.g.: $\Pi_{\text{Major}}(\text{Students})$

Student Table				Result
SID	Name	GPA	Major	Major
456	Jay	3.4	CS	CS
457	Ajay	3.2	CS	CS

5.5 RENAMING OPERATION ρ

The rename operation is used to change the name of relation R, and names of attributes of R

Format: $\rho_S(R)$ or $\rho_S(A1, A2, \dots)(R)$:

E.g.: $\rho_{\text{CS_Students}}(\sigma_{\text{Major} = \text{'CS'}} \text{Students})$

Student Table				CS_Students
SID	Name	GPA	Major	Major
456	Jay	3.4	CS	CS
457	Ajay	3.2	CS	CS

Check Your Progress!!!

Explain the usage of the following relational algebra operations:

1. Selection
2. Projection
3. Rename

5.6 THE UNION OPERATION \cup

UNION of R and S the union of two relations is a relation that includes all the tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.

The union operation is denoted as \cup in set theory. It returns the union (set union) of two compatible relations.

For a union operation $R \cup S$ to be legal, we require that

- R and S must have the same number of attributes.
- The domains of the corresponding attributes must be the same.

E.g.:

R		S		R \cup S	
A	1	A	1	A	1
B	2	C	2	B	2
D	3	D	3	C	2
F	4	E	4	D	3
E	5			E	5
				F	4
				E	4

5.7 THE INTERSECTION OPERATION \cap

The intersection of R and S includes all tuples that are both in R and S. The set intersection is a binary operation that is written as $R \cap S$ and is defined as the usual set intersection in set theory:

$$R \cap S = \{t : t \in R, t \in S\}$$

The result of the set intersection is only defined when the two relations have the same headers. This operation can be simulated in the basic operations as follows:

$$R \cap S = R - (R - S)$$

E.g.:

R		S		R ∩ S	
A	1	A	1	A	1
B	2	C	2		
D	3	D	3		
F	4	E	4		
E	5				

5.8 THE SET DIFFERENCE

DIFFERENCE of R and S is the relation that contains all the tuples that are in R but that are not in S. For set operations to function correctly the relations R and S must be union compatible. Two relations are set difference compatible if

They have the same number of attributes

The domain of each attribute in column order is the same in both R and S.

The set difference is a binary operation that is written as $R - S$ and is defined as the usual set difference in set theory:

$$R - S = \{t: t \in R, \neg t \in S\}$$

The result of the set difference is only defined when the two relations have the same headers.

E.g.:

R		S		R - S	
A	1	A	1	B	2
B	2	C	2	F	4
D	3	D	3		
F	4	E	4		
E	5				

Check Your Progress!!!

Explain the usage of the following relational algebra operations:

1. Union
2. Intersect
3. Set Difference

5.9 THE CARTESIAN PRODUCT \times

The Cartesian product of two tables combines each row in one table with each row in the other table.

The Cartesian product of n domains, written $\text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_n)$, is defined as follows.

$$(A_1 \times A_2 \times \dots \times A_n = \{(a_1, a_2, \dots, a_n) \mid a_1 \in A_1 \text{ AND } a_2 \in A_2 \text{ AND } \dots \text{ AND } a_n \in A_n\})$$

We will call each element in the Cartesian product a tuple. So each (a_1, a_2, \dots, a_n) is known as a tuple. Note that in this formulation, the order of values in a tuple matters.

Example: The table E (for EMPLOYEE)

EID	EName	Dept
1	Mahesh	A
2	Suresh	B
3	Ganesh	C

Example: The table D (for DEPARTMENT)

DID	DName
A	Marketing
B	Sales
C	Legal

The operation $\sigma (E \times D)$ will result as:

EID	EName	Dept	DID	DName
1	Mahesh	A	A	Marketing
1	Mahesh	A	B	Sales
1	Mahesh	A	C	Legal
2	Suresh	B	A	Marketing
2	Suresh	B	B	Sales
2	Suresh	B	C	Legal
3	Ganesh	C	A	Marketing
3	Ganesh	C	B	Sales
3	Ganesh	C	C	Legal

5.10 JOINS

Theta-join

The theta-join operation is the most general join operation. We can define theta-join in terms of the operations that we are familiar with already.

$$R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$$

So the join of two relations results in a subset of the Cartesian product of those relations. Which subset is determined by the join condition θ . Let's look at an example. The result of $\text{EMPLOYEE} \bowtie_{\text{Dept} = \text{DID}} \text{DEPARTMENT}$ is shown below

EID	EName	Dept	DID	DName
1	Mahesh	A	A	Marketing
2	Suresh	B	B	Sales
3	Ganesh	C	C	Legal

Equi-join

The join condition can be any well-formed logical expression, but usually it is just the conjunction of equality comparisons between pairs of attributes, one from each of the joined relations. This common case is called an equi-join. The example given above is an example of an equi-join.

Outer Joins:

- A join operation is complete, if all the tuples of the operands contribute to the result.
- Tuples not participating in the result are said to be dangling
- Outer join operations are variants of the join operations in which the dangling tuples are appended with NULL fields. They can be categorized into left, right, and full outer joins.

Consider the following relations:

Instructor

ID	Name	Dept_name
10101	Suresh	CS
12121	Mahesh	IT
15151	Ganesh	Chem

Teaches

ID	Course_ID
10101	CS-101
12121	IT-202
76766	CH-301

Instructor \bowtie_{right} Teaches (RIGHT OUTER JOIN)

ID	Name	Dept_name	Course_ID
10101	Suresh	CS	CS-101
12121	Mahesh	IT	IT-202
76766	Null	Null	CH-301

Instructor \bowtie_{left} Teaches (LEFT OUTER JOIN)

ID	Name	Dept_name	Course_ID
10101	Suresh	CS	CS-101
12121	Mahesh	IT	IT-202
15151	Ganesh	Chem	null

Instructor $\bowtie_{\text{left}} \bowtie_{\text{right}}$ Teaches (FULL OUTER JOIN)

ID	Name	Dept_name	Course_ID
10101	Suresh	CS	CS-101
12121	Mahesh	IT	IT-202
76766	Null	Null	CH-301
15151	Ganesh	Chem	null

Check Your Progress!!!

1. Explain the usage of the following relational algebra operations:
 - a. Cartesian Product
 - b. Joins
2. Explain the different types of Joins.

5.11 RELATIONAL CALCULUS

An operational methodology, founded on predicate calculus, dealing with descriptive expressions that are equivalent to the operations of relational algebra. Codd's reduction algorithm can convert from relational calculus to relational algebra. Two forms of the relational calculus exist: the tuple calculus and the domain calculus. Codd proposed the concept of a relational calculus (applied predicate calculus tailored to relational databases).

Why it is called relational calculus?

It is founded on a branch of mathematical logic called the predicate calculus. Relational calculus is a formal query language where we write one declarative expression to specify a retrieval request and hence there is no description of how to evaluate a query; a calculus expression specifies what is to be retrieved rather than how to retrieve it. Therefore, the relational calculus is considered to be a nonprocedural language. This differs from relational algebra, where we must write a sequence of operations to specify a retrieval request; hence it can be considered as a procedural way of stating a query. It is possible to nest algebra operations to form a single expression; however, a certain order among the operations is always explicitly specified in a relational algebra expression. This order also influences the strategy for evaluating the query.

It has been shown that any retrieval that can be specified in the relational algebra can also be specified in the relational calculus, and vice versa; in other words, the expressive power of the two languages is identical. This has led to the definition of the concept of a relationally complete language. A relational query language L is considered relationally complete if we can express in L any query that can be expressed in relational calculus. Relational completeness has become an important basis for comparing the expressive power of high-level query languages. However certain frequently required queries in database applications cannot be expressed in relational algebra or calculus. Most relational query languages are relationally complete but have more expressive power than relational algebra or relational calculus because of additional operations such as aggregate functions, grouping, and ordering.

5.12 UNIT END EXERCISE

Consider the following relations

works(person name, company name, salary);
lives(person name, street, city);
located in(company name, city);
managers(person name, manager name);
where manager name refers to person name.

Write Relational Algebra Queries for the following:

1. Find the names of the persons who work for company 'FBC' (company name='FBC').
2. List the names of the persons who work for company 'FBC' along with the cities they live in.
3. Find the persons who work for company 'FBC' with a
4. Find the names of the persons who live and work in the same city.
5. Find the names of the persons who live in the same city and on the same street as their managers.
6. Find the names of the persons who do not work for company 'FBC'.

5.13 FURTHER READING AND REFERENCES

Database Management Systems – Ramakrishnam, Gehrke , McGraw- Hill.

Fundamentals of Database Systems – Elmasri and Navathe, Pearson Education

Database Systems, Design, Implementation and Management – Peter Rob and Coronel, Thomson Learning

A First Course in Database Systems, Jeffrey D. Ullman, Jennifer Widom, Pearson Education



SQL

Unit Structure

- 6.0 Introduction
- 6.1 Introduction
- 6.2 CREATE TABLE Statement
- 6.3 Integrity Constraints
- 6.4 ALTER TABLE Statement
- 6.5 RENAME Statement
- 6.6 DROP Statement
- 6.7 INSERT Statement
- 6.8 UPDATE Statement
- 6.9 DELETE Statement
- 6.10 TRUNCATE Statement
- 6.11 SELECT Statement
- 6.12 GROUP Functions
- 6.13 GROUP BY Clause
- 6.14 HAVING Clause
- 6.15 Unit End Exercise
- 6.16 Further Reading and References

6.0 OBJECTIVES

In this chapter we will study the different DDL commands of SQL – CREATE TABLE, ALTER TABLE and DROP TABLE, and the DML commands of SQL – INSERT, UPDATE, DELETE and SELECT – simple queries.

6.1 INTRODUCTION

There are three groups of commands in SQL:

1. Data Definition
2. Data Manipulation and
3. Transaction Control

SQL Data Definition Language (DDL)

The Data Definition Language (DDL) part of SQL permits database tables to be created or deleted. We can also define indexes (keys), specify links between tables, and impose constraints between database tables.

The most important DDL statements in SQL are:

- CREATE TABLE - creates a new database table
- ALTER TABLE - alters (changes) a database table
- DROP TABLE - deletes a database table

SQL Data Manipulation Language (DML)

SQL language also includes syntax to update, insert, and delete records. These query and update commands together form the Data Manipulation Language (DML) part of SQL:

- INSERT INTO - inserts new data into a database table
- UPDATE - updates data in a database table
- DELETE - deletes data from a database table
- SELECT - extracts data from a database table

Transaction Control Language(TCL)

The SQL Data Control Language (DCL) provides security for your database. The DCL consists of the GRANT, REVOKE, COMMIT, and ROLLBACK statements. GRANT and REVOKE statements enable you to determine whether a user can view, modify, add, or delete database information.

6.2 CREATE TABLE STATEMENT

The CREATE TABLE Statement is used to create tables to store data. One can define Integrity Constraints like primary key, unique key, foreign key for the columns while creating the table. These constraints can be defined either at the column level or at the table level.

SYNTAX

```
CREATE TABLE table_name
(column_name1 datatype [constraint],
column_name2 datatype [constraint],
... column_nameN datatype [constraint]
) [constraints];
```

Following is an example to create the Customer (CustNum,FirstName, LastName, Address, City, State, ZIP) table,

```
CREATE TABLE Customer
(CustNum number(4),
FirstName varchar(20),
LastName varchar(20),
Address varchar(40),
City varchar(20),
State varchar(20)
ZIP number(6));
```

6.3 INTEGRITY CONSTRAINTS

A relational DBMS imposes one or more data integrity constraints to preserve the consistency and correctness of its stored data. These constraints restrict the data values that can be inserted or updated into the table. Several different types of data integrity constraints are commonly found in relational databases. The constraints available in SQL are Primary Key, Foreign Key, Not Null, Unique, Check.

Constraints can be defined in two ways

1. The constraints can be specified immediately after the column definition. This is called column-level definition.
2. The constraints can be specified after all the columns are defined. This is called table-level definition.

ENTITY INTEGRITY CONSTRAINT – PRIMARY KEY

The Primary Key constraint defines a column or combination of columns which uniquely identifies each row in the table. When a primary key is specified for a table, the DBMS automatically checks the uniqueness of the primary key value for every INSERT and UPDATE statement performed on the table. An attempt to insert a row with a duplicate primary key value or to update a row so that its primary key would be a duplicate, will fail with an error message.

Syntax to define a Primary key at column level:

```
column name datatype [CONSTRAINT constraint_name]
PRIMARY KEY
```

Syntax to define a Primary key at table level:

```
[CONSTRAINT constraint_name] PRIMARY KEY
(column_name1,column_name2,..)
```

For Example: The following query sets CustNum column of Customer table as Primary Key (at Column Level)

```
CREATE TABLE Customer
(CustNum number(4) custnum_pk PRIMARY KEY,
FirstName varchar(20),
LastName varchar(20),
Address varchar(40),
City varchar(20),
State varchar(20)
ZIP number(6));
```

For Example: The following query sets CustNum column of Customer table as Primary Key (at Table Level)

```
CREATE TABLE Customer
(CustNum number(4),
FirstName varchar(20),
LastName varchar(20),
Address varchar(40),
City varchar(20),
State varchar(20),
ZIP number(6),
CONSTRAINT custnum_pk PRIMARY KEY(CustNum));
```

REFERENTIAL INTEGRITY - FOREIGN KEY

This constraint identifies any column referencing the PRIMARY KEY of another table. It establishes a relationship between two columns in the same table or between different tables. For a column to be defined as a Foreign Key, it should be defined as a Primary Key in the table in which it is referring. One or more columns can be defined as foreign key.

Syntax to define a Foreign key at column level:

```
[CONSTRAINT constraint_name] REFERENCES
Referenced_Table_name(column_name)
```

Syntax to define a Foreign key at table level:

```
[CONSTRAINT constraint_name] FOREIGN
KEY(column_name) REFERENCES
referenced_table_name(column_name);
```

For Example consider the Order (O_Id*, C_Id#, O_Date, O_Amount, O_Status) table where C_Id is referencing the CustNum column of Customer table.

The following query sets CustId column of Order table as Foreign Key (at Column Level)

```
CREATE TABLE
(O_Id number(5) CONSTRAINT o_id_pk PRIMARY KEY,
C_Id number(5) CONSTRAINT o_id_fk REFERENCES,
Customer(CustNum),
O_Date date,
O_Amount number(5,2),
O_Status varchar(10),
);
```

The following query sets CustId column of Order table as Foreign Key (at Table Level)

```
CREATE TABLE
(O_Id number(5),
C_Id number(5),
O_Date date,
O_Amount number(5,2),
O_Status varchar(10),
CONSTRAINT o_id_pk PRIMARY KEY(O_Id)
CONSTRAINT o_id_fk FOREIGN KEY (C_Id)
REFERENCES, Customer(CustNum)
);
```

NOT NULL CONSTRAINT

The NOT NULL constraint ensures that all rows in the table contain a definite value for the specified column.

SYNTAX

```
[CONSTRAINT constraint name] NOT NULL
```

For example, the following query sets ZIP column of Customer table to be NOT NULL

```
CREATE TABLE Customer
(CustNum number(4) PRIMARY KEY,
FirstName varchar(20),
LastName varchar(20),
Address varchar(40),
City varchar(20),
State varchar(20)
ZIP number(6) NOT NULL,
);
```

UNIQUE KEY

It is sometimes appropriate to require a column or a group of columns that is not the PRIMARY KEY of a table a unique value in every row. This goal can be achieved by imposing a uniqueness constraint on the named columns. This constraint ensures that a column or a group of columns in each row have a distinct value. The column(s) can have a null value but the values cannot be duplicated.

Syntax to define a Unique key at column level

```
[CONSTRAINT constraint_name] UNIQUE
```

Syntax to define a Unique key at table level

```
[CONSTRAINT constraint_name] UNIQUE(column_name)
```

For Example: The following query sets FirstName column of Customer table as Unique (at Column Level)

```
CREATE TABLE Customer
(CustNum number(4) PRIMARY KEY,
FirstName varchar(20) UNIQUE,
LastName varchar(20),
Address varchar(40),
City varchar(20),
State varchar(20),
ZIP number(6),
);
```

CHECK CONSTRAINT

The CHECK constraint defines a business rule on a column. When a check constraint is specified for a column, the DBMS automatically checks the value of that column each time a new row is inserted or a row is updated to insure that the search condition is true. If not, the INSERT or UPDATE statement fails. The constraint can be applied for a single column or a group of columns.

Syntax to define a Check constraint:

```
[CONSTRAINT constraint_name] CHECK (condition)
```

For Example: The following query would restrict to enter Mumbai or Pune as values for City column of Customer table (at Column Level)

```
CREATE TABLE Customer
(CustNum number(4) PRIMARY KEY,
FirstName varchar(20) UNIQUE,
LastName varchar(20),
Address varchar(40),
City varchar(20) CHECK CITY in ('Mumbai', 'Pune'),
State varchar(20),
ZIP number(6),
);
```

For Example: The following query would restrict to enter Mumbai or Pune as values for City column of Customer table (at Table Level)

```
CREATE TABLE Customer
(CustNum number(4) PRIMARY KEY,
FirstName varchar(20) UNIQUE,
LastName varchar(20),
Address varchar(40),
City varchar(20) CHECK CITY in ('Mumbai', 'Pune'),
State varchar(20),
ZIP number(6),
CONSTRAINT city_ck CHECK (CITY in ('Mumbai', 'Pune'))
);
```

Check Your Progress!!!

1. State the different SQL – DDL and DML statements.
2. Explain with examples, the syntax of Create Statement.
3. What are Integrity Constraints? Explain the syntax to add integrity constraints in SQL.

6.4 ALTER TABLE STATEMENT

The SQL ALTER TABLE command is used to modify the definition (structure) of a table. One can add new columns to a table, remove existing columns from a table, change datatype of a column of a table and add or remove constraints from a table by using the ALTER command.

SYNTAX TO ADD A COLUMN

```
ALTER TABLE table_name ADD column_name datatype;
```

For Example: The following query would add a column “mobile_no” to the cusyomer table,

```
ALTER TABLE Customer ADD mobile_no number(10);
```

SYNTAX TO DROP A COLUMN

```
ALTER TABLE table_name DROP column_name;
```

For Example: The following query would drop the column "City" from the Customer table,

```
ALTER TABLE employee DROP City;
```

SYNTAX TO MODIFY A COLUMN

```
ALTER TABLE table_name MODIFY column_name
datatype;
```

For Example: The following query would modify the column Address in the Customer table to store longer addresses,

```
ALTER TABLE Customer MODIFY Address varchar(50);
```

SYNTAX TO ADD A CONSTRAINT

```
ALTER TABLE table_name ADD CONSTRAINT
[constraint_name] the_constraint;
```

For Example: The following query would add a primary key constraint to CustNum column of Customer table

```
ALTER TABLE Customer ADD CONSTRAINT cust_pk
PRIMARY KEY (CustNum)
```

SYNTAX TO DROP A CONSTRAINT

```
ALTER TABLE table_name DROP constraint_name;
```

For Example: The following query would drop the primary key constraint on CustNum column of Customer table

```
ALTER TABLE Customer cust_pk;
```

6.5 RENAME STATEMENT

One can change the name of a table or a database object using the SQL RENAME command.

SYNTAX

```
RENAME old_table_name TO new_table_name;
```

For Example: To change the name of the table Customer to The_Customers, the query would be like

```
RENAME Customer TO The_Customers;
```

6.6 DROP STATEMENT

The SQL DROP command is used to remove an existing object from the database. If a table is dropped, all the rows in the

table would be deleted and the table structure is removed from the database.

SYNTAX

DROP TABLE table_name;

For Example: The following query would drop the table Customer,
DROP TABLE Customer;

Check Your Progress!!!

Explain with examples, the syntax of:

1. Alter Table Statement.
2. Rename Statement
3. Drop Table Statement

6.7 INSERT STATEMENT

A new row of data is added to a table by using the INSERT Statement. There are two ways of adding new rows of data to a table:

1. **Single – row INSERT:** It adds a single new row of data to a table.
2. **Multi – row INSERT:** It extracts rows of data from another table and adds them to the table.

SINGLE ROW INSERT

The single – row INSERT statement, adds a new row of data to a table. The INTO clause specifies the table that receives the new row, and the VALUES clause specifies the data values that the new row will contain. The column list indicates which data value goes into which column of the new row. While inserting a row, if you are adding value for all the columns of the table then the column list can be omitted.

SYNTAX

```
INSERT INTO TABLE_NAME
[ (col1, col2, col3,...colN)]
VALUES (value1, value2, value3,...valueN);
OR
INSERT INTO TABLE_NAME
VALUES (value1, value2, value3,...valueN);
```

Note:

1. When SQL inserts a new row of data to a table, it automatically assigns a NULL value to any column whose name is missing from the column list in the INSERT statement.
2. The assignments of a NULL value can be made explicit by including these columns in the column list and specifying the keyword NULL in the VALUES list.

For Example: If you want to insert a row to the Customer table, the query would be like,

```
INSERT INTO Customer (CustNum, FirstName,
LastName, Address, City, State, ZIP) VALUES (101,
'Vipul', 'Saluja', 'Khar', 'Mumbai', 'Maharashtra', NULL);
OR
INSERT INTO Customer VALUES (101, 'Vipul', 'Saluja',
'Khar', 'Mumbai', 'Maharashtra', 400052);
```

MULTI ROW INSERT

The multi – row INSERT statement, adds a multiple rows of data to a table. The INTO clause specifies the table that receives the new rows. The data values for the new row are not explicitly specified within the statement text. The source of new rows is the specified select query.

SYNTAX

```
INSERT INTO table_name
[(column1, column2, ... columnN)]
SELECT column1, column2, ...columnN
FROM table_name [WHERE condition];
```

For Example: The following query would insert all rows of the Customer table to the New_Customer table,

```
INSERT INTO New_Customer (CustNum, FirstName,
LastName, Address, City, State, ZIP)
SELECT CustNum, FirstName, LastName, Address, City,
State, ZIP FROM Customer;
OR
INSERT INTO New_Customer
SELECT * FROM Customer;
```

6.8 UPDATE STATEMENT

Any changes are to be made to the existing data i.e. rows of a table, can be done with the UPDATE Statement.

SYNTAX

```
UPDATE table_name
SET column_name1 = value1,
column_name2 = value2,
...
...
column_nameN = valueN,
[WHERE condition]
```

One should note that the WHERE clause identifies the rows that get affected in the UPDATE statement. If the WHERE clause is omitted, all the rows would be affected.

For Example: The following query would update the City of a Customer Number 101 to Mumbai,

```
UPDATE Customer
SET City ='Mumbai'
WHERE CustNum = 101;
```

6.9 DELETE STATEMENT

One can delete the existing row(s) from a table using the DELETE Statement.

SYNTAX

```
DELETE FROM table_name [WHERE condition];
```

One should note that the WHERE clause identifies the rows in the column that gets deleted. If the WHERE clause is omitted all the rows in the table would be deleted.

For Example: The following query would delete Customer Number 101 from the Customer table,

```
DELETE FROM Customer
WHERE CustNum = 101;
```

The following query would delete all rows from the Customer table

```
DELETE FROM Customer;
```

6.10 TRUNCATE STATEMENT

One can delete all the rows from the table and free the space containing the table using the TRUNCATE statement

SYNTAX

TRUNCATE TABLE table_name;

For Example: The following query would delete all the rows from the Customer Table

TRUNCATE TABLE Customer;

Difference between DELETE and TRUNCATE Statements:

- The DELETE Statement deletes only the rows from the table based on the condition given in the where clause or deletes all the rows from the table if no condition is specified, but does not free the space containing the table, whereas the TRUNCATE statement deletes all the rows of the table and frees up the space containing the table.
- TRUNCATE requires exclusive access to the table whereas DELETE doesn't.
- TRUNCATE as compared to DELETE is faster and uses fewer system and transaction log resources.

Check Your Progress!!!

1. Explain with examples, the syntax of the two forms of Insert statement.
2. Explain with examples, the syntax of update and delete statement of SQL
3. State the difference between delete and truncate statement of SQL.

6.11 SELECT STATEMENT

The SQL SELECT statement is used to query or retrieve data from a table in the database. A query may retrieve information from specified columns or from all of the columns in the table. A full formed SELECT statement contains six clauses. The SELECT and FROM clauses are compulsory whereas the remaining four – WHERE, GROUP BY, HAVING and ORDER BY are optional.

- **SELECT** – It lists the column names that are to be retrieved by the query.
- **FROM** – It list the table name(s) that contains the data to be retrieved by the query.
- **WHERE** – It tells SQL to include/reject rows from the result depending on the condition specified.
- **GROUP BY** – It specifies a summary query. Instead of producing one row of query results for each row of data, the summary query groups together similar rows and then produces the result for each froup.
- **HAVING** – It tells SQL to include only certain groups produced by the GROUP BY clause in the query results. It is used like a WHERE clause as a search condition to specify the desired groups.
- **ORDER BY** – It sorts the query results based on the data in one or more columns.

SYNTAX

```
SELECT column_list FROM table-name
[WHERE Clause]
[GROUP BY clause]
[HAVING clause]
[ORDER BY clause];
```

For example, the following query would retrieve first name and last name of all the Customers.

```
SELECT firstname, lastname FROM Customers;
```

Row Selection – The WHERE Clause

At times, we may not want all the rows of the table as a query result. We may want to include only some of the rows of the query result. The WHERE clause is used to specify the rows you want to retrieve.

The WHERE clause consists of the keyword WHERE followed by the search condition. SQL goes through each row in the table and applies the search condition to the row. For each row, the search condition can produce on the following three results:

- Row is included in query result if search condition is TRUE
- Row is excluded from the query result if search condition is FALSE
- Row is excluded from the query result if search condition is NULL

For Example, The following query lists the name of customers located at Mumbai

```
SELECT firstname, lastname FROM Customer
WHERE City= 'Mumbai';
```

Search Conditions

Following are the search conditions in SQL:

- **Comparison Test (=, < >, <, <=, >, >=)** – compares the value of one expression with another expression. The expressions can be as column names, constants, or complex arithmetic expressions.

For example, the following query returns the details of customer number 101

```
SELECT * FROM Customer
WHERE CustNum = 101;
```

- **Range Test (BETWEEN)** – tests whether the value of an expression falls within a specified range of values.

For example, the following query returns the details of all customers with customer numbers between 110 and 120

```
SELECT * FROM Customer
WHERE CustNum between 110 and 120
```

- **Set Membership Test (IN)** – checks whether the value of an expression matches one of the set values.

For example, the following query returns the details of all customers located at Mumbai, Pune or Hyderabad

```
SELECT * FROM Customer
WHERE City IN ('Mumbai', 'Pune', 'Hyderabad')
```

- **Pattern Matching Test (LIKE)** – checks whether the value of a column containing a string data matches a specified pattern. In pattern matching test, the following wildcard

characters are used. These characters are interpreted in a special way.

- % - used to match zero or more characters
- _ - used to match a single character

For example, the following query returns the details of all customers located whose first names start with M

```
SELECT * FROM Customer
WHERE firstname like 'M%'
```

- **NULL Value Test (IS NULL)** – checks whether a column has a NULL value.

For example, the following query returns the details of all customers who have not yet been given a CustNum

```
SELECT * FROM Customer
WHERE CustNum IS NULL'
```

Note:

We can build complex search criteria using the logical AND, OR and NOT keywords.

Sorting the Results – The ORDER BY Clause

The rows of a query result are not arranged in any particular order. We can sort the results of a query by using the ORDER BY clause in the SELECT statement. A list of columns can be mentioned in the ORDER BY clause. By default sorting is done in ascending order. We can also specify ascending order by using the keyword ASC after the column name. For sorting to be done in descending order the keyword DESC must be included after the column name.

For example, the following query returns the details of all customers arranged in the ascending order of cities in which they are located

```
SELECT * FROM Customer
ORDER BY City
```

Eliminating Duplicates – The DISTINCT Clause

We can eliminate duplicate rows of query result by inserting the DISTINCT keyword in the SELECT statement.

For example, the following query returns all the unique cities where the customers are located

```
SELECT DISTINCT CITY FROM Customer
```

6.12 AGGREGATE FUNCTIONS

Aggregate functions that calculate summary values, such as averages and sums, from the values in a particular column and return a single value for each set of rows to which the function applies. The aggregate functions are AVG, COUNT, COUNT(*), MAX, MIN and SUM. Aggregate functions can be applied either to all rows in a table, to a subset of table rows specified by a WHERE clause, or to one or more groups of table rows specified by the GROUP BY clause.

Consider the employee(empno, ename, job, hiredate, sal, comm,deptno) for the example queries below

1. **COUNT ()**: This function returns the number of data values of the specified column, for those rows that satisfy the condition of the WHERE clause. If the WHERE clause is omitted, then the query returns the total number of data values in the specified column.

For Example: The following query would return the number of Employees drawing a salary of more than 10000

```
SELECT COUNT (empno) FROM employee
WHERE salary > 10000;
```

2. **COUNT (*)**: This function returns the numbers of rows in the table rather than the number of data values in a column of the table.

For Example: The following query would return the number of employees

```
SELECT COUNT (*) FROM employees
```

3. **MAX()**: This function returns the maximum value of the specified column, for those rows that satisfy the condition of the WHERE clause. If the WHERE clause is omitted, then the query returns the maximum of all rows of the specified column.

For example: The following query would return the maximum salary drawn by an employee,

```
SELECT MAX (salary) FROM employee;
```

4. **MIN()**: This function returns the minimum value of the specified column, for those rows that satisfy the condition of

the WHERE clause. If the WHERE clause is omitted, then the query returns the maximum of all rows of the specified column.

For example: The following query would return the maximum salary drawn by an employee,

```
SELECT MAX (salary) FROM employee;
```

5. **AVG():** This function computes and returns the average of the data values of the specified numeric column, for those rows that satisfy the condition of the WHERE clause. If the WHERE clause is omitted, then the query returns the average of all data values of the specified column.

For example, the following query returns the average salary of all employees,

```
SELECT AVG (salary) FROM employee;
```

6. **SUM():** This function computes and returns the sum of the data values of specified numeric column, for those rows that satisfy the condition of the WHERE clause. If the WHERE clause is omitted, then the query returns the average of all data values of the specified column.

For example, the following query returns the sum of salaries of all employees,

```
SELECT SUM (salary) FROM employee;
```

Check Your Progress!!!

1. Explain the six clauses of Select statement.
2. Explain the different search conditions that can be applied in Select statement.
3. Write a note on the aggregate functions of SQL.

6.13 GROUPED QUERIES

A query that includes the GROUP BY clause is called a grouped query because it groups the data from the source tables and produces a single summary row for each row group. The columns named in the GROUP BY clause are called the grouping columns of the query as they determine how the rows are divided into groups. There is a link between the SQL aggregate functions and the GROUP BY clause. A column function takes a column of data values and produces a single result. The GROUP BY clause should contain all the columns in the select list except those used along with the group functions.

For Example: The following query would return total amount of salary spent on each department,

```
SELECT dept, SUM (salary)
FROM employee
GROUP BY deptno;
```

For Example: The following query would return total amount of salary location wise and department wise,

```
SELECT location, dept, SUM (salary)
FROM employee
GROUP BY location, dept;
```

6.14 GROUPED SEARCH CONDITIONS – THE HAVING CLAUSE

Just as the WHERE clause can be used to include or reject the rows in query result, the HAVING clause can be used to include or reject row groups i.e. HAVING clause is used to filter data based on the group functions. The format of the HAVING clause is similar to the WHERE clause, consisting of the keyword HAVING followed by a searched condition for groups.

Group functions cannot be used in WHERE Clause but can be used in HAVING clause.

For Example: The following query would return the department that has paid a total salary of more than 50000,

```
SELECT dept, SUM (salary)
FROM employee
GROUP BY dept
HAVING SUM (salary) > 50000
```

When WHERE, GROUP BY and HAVING clauses are used together in a SELECT statement, the WHERE clause is processed first, then the rows that are returned after the WHERE clause is executed, are grouped based on the GROUP BY clause and finally, any conditions on the group functions in the HAVING clause is applied to the grouped rows before the final output is displayed.

6.15 UNIT END EXERCISE

Consider the following relations:

Dept(deptno, dname, location)

Emp(empno, ename, job, mgr, sal, comm., deptno, hiredate)

Write queries for the following:

1. To Create the Dept and Emp Tables.
2. Display all information in the tables EMP and DEPT.
3. Display only the hire date and employee name for each employee.
4. Display the hire date, name and department number for all clerks.
5. Display the names and salaries of all employees with a salary greater than 2000.
6. Display the names of all employees with an 'A' in their name.
7. Display the names of all employees with exactly 5 letters in their name.
8. Display the names and hire dates of all employees hired in 1981 or 1982.
9. Display the names and dates of employees with the column headers 'Name' and 'Start Date'
10. Display the names and hire dates of all employees in the order they were hired.
11. Display the names and salaries of all employees in reverse salary order.
12. Display 'ename of department deptno earned commission %' for each salesman in reverse salary order.
13. Display the department numbers of all departments employing a clerk.
14. Display the maximum, minimum and average salary and commission earned.
15. Display the department number, total salary payout and total commission payout for each department.

16. Display the department number, total salary payout and total commission payout for each department that pays at least one employee commission.
17. Display the department number and number of clerks in each department.
18. Display the department number and total salary of employees in each department that employs four or more people.
19. Display the employee number of each employee who manages other employees with the number of people he or she manages.

6.16 FURTHER READING AND REFERENCES

Professional SQL server 2000 Programming – Rob Vieira, Wrox Press Ltd, Shroff

SQL server 2000 black book- Patrick Dalton and Paul Whitehead, Dreamtech Press

Understanding SQL, Martin Gruber, B.P.B. Publications



SQL BUILT IN FUNCTIONS

Unit Structure

7.0 Objectives

7.1 Character or Text Functions

7.2 Date Functions

7.3 Conversion Functions

7.4 Unit End Exercise

7.5 Further Reading and References

7.1 OBJECTIVES

In this chapter we will look at the syntax and examples of different built in functions in SQL.

7.1 CHARACTER OR TEXT FUNCTIONS

The string functions perform operations on a string (char or varchar) input value and return a string or numeric value. Some of the character or text functions are as follows:

1. LOWER (str) – Would return a string with all characters of str converted to lower case.
2. UPPER (str) – Would return a string with all characters of str converted to upper case.
3. INITCAP (str) – Would return a string with the first character of every letter of str converted to upper case.
4. LTRIM (str) – Would return a string by removing any white spaces that are present at the beginning of the string.
5. RTRIM (str) – Would return a string by removing any white spaces that are present at the ending of the string.
6. TRIM (str) – Would return a string by removing any white spaces that are present at the beginning or ending of the string.

7. SUBSTR(str, m, n) – Would return a string by extracting n characters from str, starting at position m
8. LENGTH(str) – Would return the length of str.
9. LPAD (str, n, pad_char) – Would return a string by left padding the string str with the pad_char such that the length of the string becomes n.
10. RPAD (str, n, pad_char) – Would return a string by right padding the string str with the pad_char such that the length of the string becomes n.

For Example, we can use the above UPPER() text function with the column value as follows.

```
SELECT UPPER (CustName) FROM customers;
```

The following examples explains the usage of the above character or text functions

Function Name	Examples	Return Value
LOWER(str)	LOWER('No Problem')	no problem
UPPER(str)	UPPER('No Problem')	NO PROBLEM
INITCAP(str)	INITCAP('no problem')	No Problem
LTRIM(str)	LTRIM (' No Problem')	No Problem
RTRIM (str)	RTRIM ('No Problem ')	No Problem
TRIM (str)	TRIM (' No Problem ')	No Problem
SUBSTR(str,m, n)	SUBSTR ('No Problem', 3, 7)	Problem
LENGTH (str)	LENGTH ('No Problem')	10
LPAD(str,n, pad_char)	LPAD ('No Problem', 12, '*')	** No Problem
RPAD(str,n, pad_char)	RPAD ('No Problem', 12, '*')	No Problem **

Check Your Progress!!!

1. Write a SQL query that would display the Names and Cities of the customers in upper case.
2. Write a SQL query that would display the first 3 characters if cities of all customers.
3. Write a SQL query that would display the customer names and their lengths.

7.2 DATE FUNCTIONS

These are functions that take values that are of data type DATE as input and return values of data types DATE, except for the MONTHS_BETWEEN function, which returns a number as output. Some of the commonly used date functions are as follows:

1. ADD_MONTHS (date, n) – Returns a date value after adding 'n' months to the date 'x'.
2. MONTHS_BETWEEN (x1, x2) – Returns the number of months between dates x1 and x2.
3. NEXT_DAY(x, week_day) – Returns the next date of the 'week_day' on or after the date 'x' occurs.
4. LAST_DAY (x) – It is used to determine the number of days remaining in a month from the date 'x' specified.
5. SYSDATE – Returns the systems current date and time.

For Example the query would return the date of Friday after the sysdate.

```
select next_day ( sysdate, 'FRIDAY' ) from dual;
```

The below table provides the examples for the above functions

Function Name	Examples	Return Value
ADD_MONTHS ()	ADD_MONTHS ('19-Sep-12', 3)	19-Dec-12
MONTHS_BETWEEN()	MONTHS_BETWEEN('19-Sep-12', '19-Nov-81')	2
NEXT_DAY()	NEXT_DAY('10-Nov-12', 'Wednesday')	13-Nov-12
LAST_DAY()	LAST_DAY ('10-Nov-12')	30-Nov-12

Check Your Progress!!!

1. Write a SQL query that would display the system date.

7.3 CONVERSION FUNCTIONS

These are functions convert a value in one form to another form. For Ex: a null value into an actual value, or a value from one

datatype to another datatype. Some of the commonly used conversion functions available in SQL are:

1. TO_CHAR (x [, fmt]) – Returns x converted to a string in the given format.
2. TO_DATE (x [, date_format]) – Returns x converted to a string in the given format.
3. NVL (x, y) – If 'x' is NULL, returns y else returns x.

The below table provides the examples for the above functions

For Example, the following query would return \$3000

```
SELECT TO_CHAR (3000, '$9999') from dual
```

For Example, the following query would return 1

```
SELECT NVL (null, 1) from dual
```

7.4 UNIT ENDEXERCISE

1. Explain the SQL Built In String Functions.
2. Explain the SQL Built In Date Functions.
3. Explain the SQL Built In Conversion Functions.

7.5 FURTHER READING AND REFERENCES

Professional SQL server 2000 Programming – Rob Vieira, Wrox Press Ltd, Shroff

SQL server 2000 black book- Patrick Dalton and Paul Whitehead, Dreamtech Press

Understanding SQL, Martin Gruber, B.P.B. Publications



JOINS AND SUBQUERIES

Unit Structure

- 8.0 Objectives
- 8.1 Joins Introduction
- 8.2 Join Syntax
- 8.3 Join Types
- 8.4 Self Join
- 8.5 Subquery Introduction
- 8.6 Subquery Syntax
- 8.7 Different Forms of Subquery
- 8.8 Correlated Subquery
- 8.9 SQL UNION clause
- 8.10 Further Reading and References
- 8.11 Unit End Exercise

8.0 OBJECTIVES

In this chapter we will study joins – their syntax and types. We would further study subqueries and the SQL set operations like UNION, INTERSECT and SET DIFFERENCE.

8.1 JOINS INTRODUCTION

The process of forming pair of rows matching the contents of related columns is called the joining of tables. SQL Joins are used to relate information in different tables. A Join condition is a part of the SQL query that retrieves rows from two or more tables. A SQL Join condition is used in the SQL WHERE Clause of select, update, delete statements.

8.2 JOIN SYNTAX

The Syntax for joining two tables is:

```
SELECT col1, col2, col3...
FROM table_name1, table_name2
WHERE table_name1.col2 = table_name2.col1;
```

Consider the following two tables

Instructor

ID	Name	Dept_name	Salary
10101	Suresh	CS	10000
12121	Mahesh	IT	9000
15151	Ganesh	Chem	12000

Teaches

ID	Course_ID
10101	CS-101
12121	IT-202
76766	CH-301

The following query displays the details of the Instructors and the Course_ID they teach from the Instructor and Teaches tables:

```
Select I.ID, I.Name, I.Dept_name, T.Course_ID
from Instructor I, Teaches T
where I.ID = T.ID
```

8.3 JOIN TYPES

Joins can be classified into Equi join and Non Equi join.

1. Equi joins

These are simple SQL join condition which uses the equal to (=) comparison operator. Equi Joins are further classified as SQL Outer join and SQL Inner join.

2. SQL Non equi joins

These are simple SQL join condition which makes use of the comparison operators other than the equal to (=) operator. The comparison operators used are >, <, >=, <=

Consider the following tables for further Join examples

Customer (CustNum*, FirstName, LastName, Address, City, State, ZIP)

Order (Ord*, ODate, OAmount, CId#)

Products (PId*, PName, PStock)

Order_Details (Ord#, PId#, Qty)

EQUI JOINS

A join based on an exact match between two columns is called an equi join. An equi-join is further classified into two categories:

a. Inner Join:

The INNER JOIN creates and returns a new result table by combining column values of two tables, say table1 and table2 based upon the join condition. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-condition. If the join-condition is satisfied, column values for each matched pair of rows of table1 and table2 are combined into a result row.

Example the following query would display the customer names, the Id and Date of the orders they place.

```
Select C.CustName, O.Id, O.Date
from Custmer C, Order )
where C.CustNum = O.Cid
```

This query will result in all those rows when CustNum from Customer table would be equal to CId from Order Table. If the value of CustNum is null or CId is null those rows would be omitted.

b. Outer Join:

As mentioned above, Inner joins return rows only when there is at least one row from both tables that match the join condition. Inner joins eliminate the rows that do not match with a row from the other table. Outer joins, however, return all rows from at least one of the tables mentioned in the FROM clause, if these rows meet the WHERE or HAVING search conditions. All rows are retrieved from the left table referenced with a left outer join, and all rows from the right table referenced in a right outer join. All rows from both tables are returned in a full outer join.

Now if we want to display those rows of Customer Table or Order Table where CustNum or CId is null we would write an outer join as follows

```
Select C.CustName, O.Id, O.Date
from Custmer C, Order )
where C.CustNum* = O.CId
```

This query would include those rows of Customer table whose CustNum values are null, but would not include those rows of Order table whose CId values are null. Such a join is known as LEFT Outer Join.

```
Select C.CustName, O.Id, O.Date
from Custmer C, Order )
where C.CustNum =* O.CId
```

This query would include those rows of Order table whose CId values are null, but would not include those rows of Customer table whose CustName values are null. Such a join is known as RIGHT Outer Join.

```
Select C.CustName, O.Id, O.Date
from Custmer C, Order )
where C.CustNum =* O.CId
```

This query would include those rows of Order table whose CId values are null, as well as those rows of Customer table whose CustName values are null. Such a join is known as FULL Outer Join.

NON EQUI JOINS

An non equi join is a join statement that uses an unequal operation (i.e: <>, >, <, !=, BETWEEN, etc.) to match rows from different tables.

Consider the following tables

Employee (EId*, Ename, Salary, DeptNo#, Sales, Mgr#)

Dept (DeptNo*, DName, Target)

The following query would display the details of those employees whose sales is more than the department target.

```
Select E.ENAME, E.Sales, E.DeptNo, D.Target from  
Employee E, Dept D  
where E.DeptNo = D.DeptNo and E.Sales>D.Target
```

8.4 SELF JOIN

Whenever a table is joined with itself, it is referred to as a Self Join. Generally this happens whenever the table has a FOREIGN KEY that references its own PRIMARY KEY. To list a table two times in the same query, you must provide a table alias for at least one of instance of the table name. This table alias helps the query processor determine whether columns should present data from the right or left version of the table.

Consider the table Employee (EId*, Ename, Salary, DeptNo#, Sales, Mgr#)

Here the mgr column is foreign key referencing the EId column of the same table.

The following query would display the names of the employees and the names of their managers.

```
Select E1.Ename, E2.Name as 'Mgr' from Employee E1,  
Employee E2  
where E1.EId=E2.Mgr
```

Check Your Progress!!!

1. Explain with an example the syntax of join operation.
2. Distinguish between Equi and Non Equi Joins.
3. Write a short note on Self Joins.

8.5 SUBQUERY INTRODUCTION

A sub query is a query within a query. The results of the sub query are used by SQL to determine the results of the outer query that contains the sub query. The sub query appears within the WHERE or HAVING clause of a SQL statement. A sub query can be nested inside a SELECT, INSERT, UPDATE, or DELETE statement, or inside another sub query.

8.6 SUBQUERY SYNTAX

The following guidelines provide details about how to implement sub queries in outer queries:

- A subquery must be enclosed in parenthesis
- A subquery must include a SELECT clause and a FROM clause
- A subquery can include optional WHERE, GROUP BY, and HAVING clauses
- A subquery cannot include an ORDER BY clause
- Subqueries can be nested up to 32 levels
- A subquery in Comparison Test, should retrieve a single value of data
- A subquery in Set Membership Test, can retrieve a single column of data with multiple values

SQL statements that include a sub query usually take one of the following forms:

1. WHERE|HAVING expression comparison_operator [ANY | ALL] (subquery)
2. WHERE expression [NOT] IN (subquery)
3. WHERE [NOT] EXISTS (subquery)

SUBQUERY SEARCH CONDITIONS

1. **Sub Query Comparison Test (=, <, >, <=, >=)** – compares the value of an expression to the value produced by the sub query and returns a TRUE result if the comparison is true. The sub query in this case must produce a single value. If the sub query produces multiple rows, or multiple columns SQL reports an error condition. If the sub query produces no rows or produces a NULL value, the comparison test returns NULL.
2. **Sub Query Set Membership Test (IN)** – compares the value of an expression to a column of data values produced by a sub query and returns a TRUE result if the data value matches one of the values in the column. The sub query in this case produces a column of data values, and the WHERE clause of the main query checks to see whether a value the value of the expression matches one of the values retrieved by the sub query. The sub query form of the IN test is similar to the simple IN test, except that the set of values

is produced by a sub query instead of being explicitly listed in the statement.

3. **Sub Query Existence Test (EXISTS)** – checks whether a sub query produces any rows of query results. When a sub query is written with the keyword EXISTS, the sub query functions as an existence test. The WHERE clause of the outer query tests whether the rows that are returned by the sub query exist. The sub query does not actually produce any data; it returns a value of TRUE or FALSE.

SEARCH QUANTIFIERS – ANY OR ALL

The Set Membership test checks whether a data value is equal to some value in a column of sub query results. SQL provides two quantified tests, ANY and ALL, that extend this notion to other comparison operators, such as greater than (>) and less than (<).

1. **ANY** – The ANY test is used in conjunction with one of the six SQL comparison operators (=, <>, <, <=, >, >=) to compare a single test value to a column of data values produced by a sub query. To perform the test, SQL uses the specified comparison operator to compare the test value to each data value in the column, one at a time. If any of the individual comparisons yield a TRUE result, the ANY test returns a TRUE result.
2. **ALL** – The ALL test is used in conjunction with one of the six SQL comparison operators (=, <>, <, <=, >, >=) to compare a single test value to a column of data values produced by a sub query. To perform the test, SQL uses the specified comparison operator to compare the test value to each data value in the column, one at a time. If all of the individual comparisons yield a TRUE result, the ALL test returns a TRUE result.

8.7 DIFFERENT FORMS OF SUB QUERIES

SUBQUERIES WITHIN THE SELECT STATEMENT

Sub queries within the SELECT statement take the following form:

```
SELECT column_name [, column_name ]
FROM table1 [, table2 ]
WHERE expression|column_name COMARISON_TEST
      (SELECT column_name [, column_name ]
      FROM table1 [, table2 ]
      [WHERE])
```

Consider the following tables

Employee (EId*, Ename, Salary, DeptNo#, Sales, Mgr#)
 Dept (DeptNo*, DName, Target)

The following query displays the employees work in a department whose target is more than 10000.

```
Select Ename From Employee
Where DeptNo IN (Select DeptNo from Dept where Target >
10000)
```

The following query displays the employees working in Accounts department.

```
Select Ename from Employee
where DeptNo = (Select DeptNo from Dept where
DName='Accounts')
```

SUBQUERIES WITHIN THE INSERT STATEMENT

Sub queries within the INSERT statement take the following form:

```
INSERT INTO table_name [ (column1 [, column2 ]) ]
SELECT [ *|column1 [, column2 ]
FROM table1 [, table2 ]
[WHERE]
```

If we have another table New_Customer whose structure is the same as Customer table, the following query would insert the Customers based in Mumbai to New_Customer

```
INSERT into NewCustomer
Select * from Customer where City= 'Mumbai'
```

SUBQUERIES WITHIN THE UPDATE STATEMENT

Sub queries within the UPDATE statement take the following form:

```
UPDATE table
SET column_name = new_value
[ WHERE expression|column_name COMARISON_TEST
(SELECT COLUMN_NAME
FROM TABLE_NAME
[WHERE])]
```

The following query would change the increase the salaries of all employees working the in Marketing Department by 15 %

Update Employee
 Set salary = salary + 0.15* salary
 where DeptNo = (Select DeptNo from Dept where
 Dname= 'Marketing')

SUBQUERIES WITHIN THE DELETE STATEMENT

Sub queries within the DELETE statement take the following form:

```
DELETE FROM TABLE_NAME
WHERE expression|column_name COMARISON_TEST
(SELECT COLUMN_NAME
FROM TABLE_NAME
[ WHERE ])
```

The following query would delete all employees working in the Finance Department.

```
Delete from Employees where
DeptNo IN (Select DeptNo from Dept where Dname=
'Finance')
```

8.8 CORRELATED SUBQUERIES

In most cases, where an outer query has a sub query in the search condition, the sub query is executed once and the result is substituted into the WHERE clause of the outer query. But, sometimes, the sub query can make a reference to a column of the outer query. Such a kind of query is known as **correlated sub query**. In such queries, the sub query depends on the outer query for its values. This means that the sub query is executed repeatedly, once for each row that is selected by the outer query.

Check Your Progress!!!

1. Explain the different subquery search conditions.
2. Explain how subqueries can be used in different DML statements.
3. Write a note on Correlated Subqueries.

8.9 COMBINING QUERY RESULTS – THE UNION CLAUSE

SQL supports the capability to combine the results of two or more queries into a single table of query result with the UNION clause.

In order to use UNION the following conditions apply:

1. The number and the order of the columns must be the same in all queries.
2. The data types must be compatible.

SYNTAX

```
SELECT column1 [, column2 ]  
FROM table1 [, table2 ]  
[WHERE condition]
```

```
UNION [ALL]
```

```
SELECT column1 [, column2 ]  
FROM table1 [, table2 ]  
[WHERE condition]
```

By default, the UNION operation eliminates duplicate rows as part of its processing. If duplicate rows are to be retained then UNION ALL must be used. The same rules that apply to UNION apply to the UNION ALL operator.

There are two other SQL clauses similar to the UNION clause:

1. **SQL INTERSECT Clause:** is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement.
2. **SQL EXCEPT Clause:** combines two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement.

8.10 UNIT END EXERCISE

Consider the following relations:

Dept(deptno, dname, location)

Emp(empno, ename, job, mgr, sal, comm., deptno, hiredate)

Write queries for the following:

1. Display the name of each employee with his department name.
2. Display a list of all departments with the employees in each department.

3. Display all the departments with the manager for that department.
4. Display the names of each employee with the name of his/her boss.
5. Display the names of each employee with the name of his/her boss with a blank for the boss of the president.
6. Display the employee number and name of each employee who manages other employees with the number of people he or she manages.
7. Display the names and job titles of all employees with the same job as Jones.
8. Display the names and department name of all employees working in the same city as Jones.
9. Display the name of the employee whose salary is the lowest.
10. Display the names of all employees except the lowest paid.
11. Display the names of all employees whose job title is the same as anyone in the sales dept.
12. Display the names of all employees who work in a department that employs an analyst.
13. Display the names of all employees with their job title, their current salary and their salary following a 10% pay rise for clerks and a 7% pay rise for all other employees.
14. Display the names of ALL employees with the total they have earned (ie. salary plus commission).

8.11 FURTHER READING AND REFERENCES

Professional SQL server 2000 Programming – Rob Vieira, Wrox Press Ltd, Shroff

SQL server 2000 black book- Patrick Dalton and Paul Whitehead, Dreamtech Press

Understanding SQL, Martin Gruber, B.P.B. Publications



INDEXING

Unit Structure

9.0 Objectives

9.1 What is an Index?

9.2 Types of Index

9.3 Indexing Structures

9.4 Cost Comparison of Operations on Different kinds of Files

9.5 Index Definitions in SQL

9.6 Unit End Exercise

9.7 Further Reading and References

9.1 WHAT IS AN INDEX?

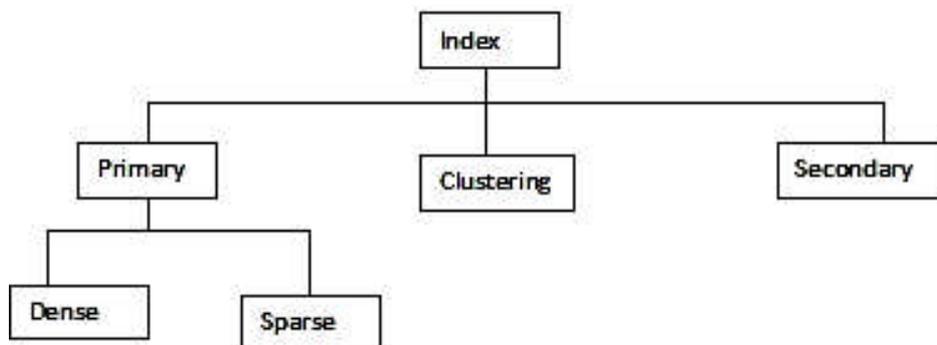
An index is a small table having only two columns. The first column contains a copy of the primary or candidate key of a table and the second column contains a set of pointers holding the address of the disk block where that particular key value can be found.

The advantage of using index lies in the fact is that index makes search operation perform very fast. Suppose a table has a several rows of data, each row is 20 bytes wide. If you want to search for the record number 100, the management system must thoroughly read each and every row and after reading $99 \times 20 = 1980$ bytes it will find record number 100. If we have an index, the management system starts to search for record number 100 not from the table, but from the index. The index, containing only two columns, may be just 4 bytes wide in each of its rows. After reading only $99 \times 4 = 396$ bytes of data from the index the management system finds an entry for record number 100, reads the address of the disk block where record number 100 is stored and directly

points at the record in the physical storage device. The result is a much quicker access to the record (a speed advantage of 1980:396).

The only minor disadvantage of using index is that it takes up a little more space than the main table. Additionally, index needs to be updated periodically for insertion or deletion of records in the main table. However, the advantages are so huge that these disadvantages can be considered negligible.

9.2 TYPES OF INDEX



9.2.1 Primary Index

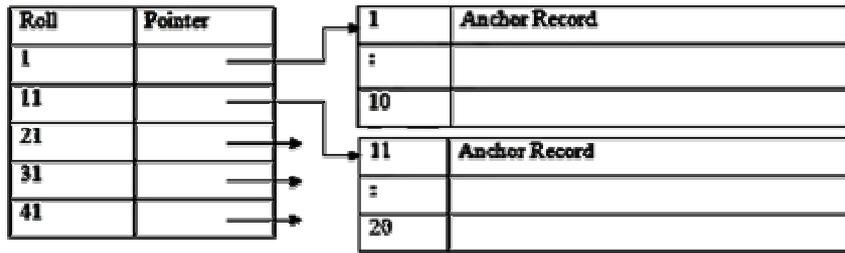
In primary index, there is a one-to-one relationship between the entries in the index table and the records in the main table. Primary index can be of two types:

1. **Dense primary index**: the number of entries in the index table is the same as the number of entries in the main table. In other words, each and every record in the main table has an entry in the index.

Roll	Pointer	Roll	
1	→	1	
2	→	2	
3	→	3	
4	→	4	
5	→	5	

2. **Sparse or Non-Dense Primary Index**:

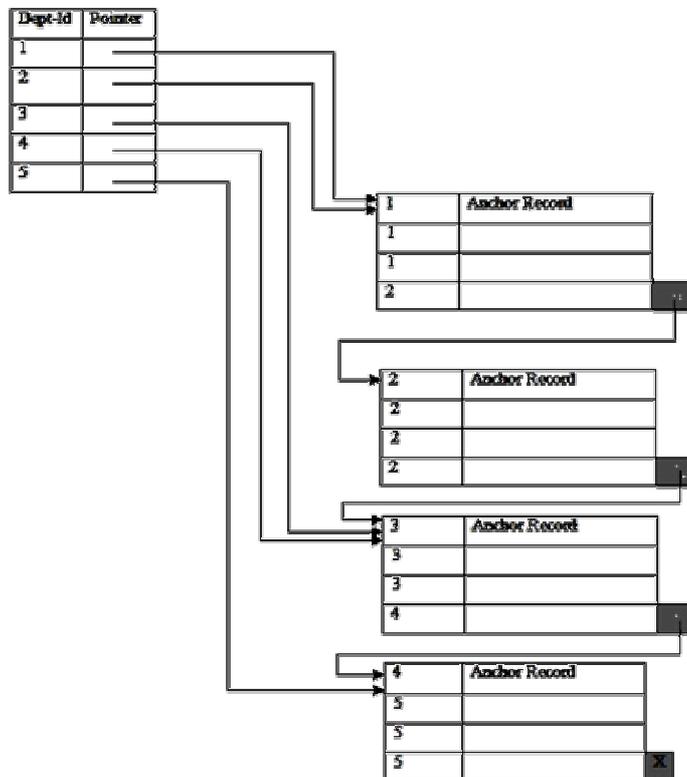
For large tables the Dense Primary Index itself begins to grow in size. To keep the size of the index smaller, instead of pointing to each and every record in the main table, the index points to the records in the main table in a gap. See the following example.



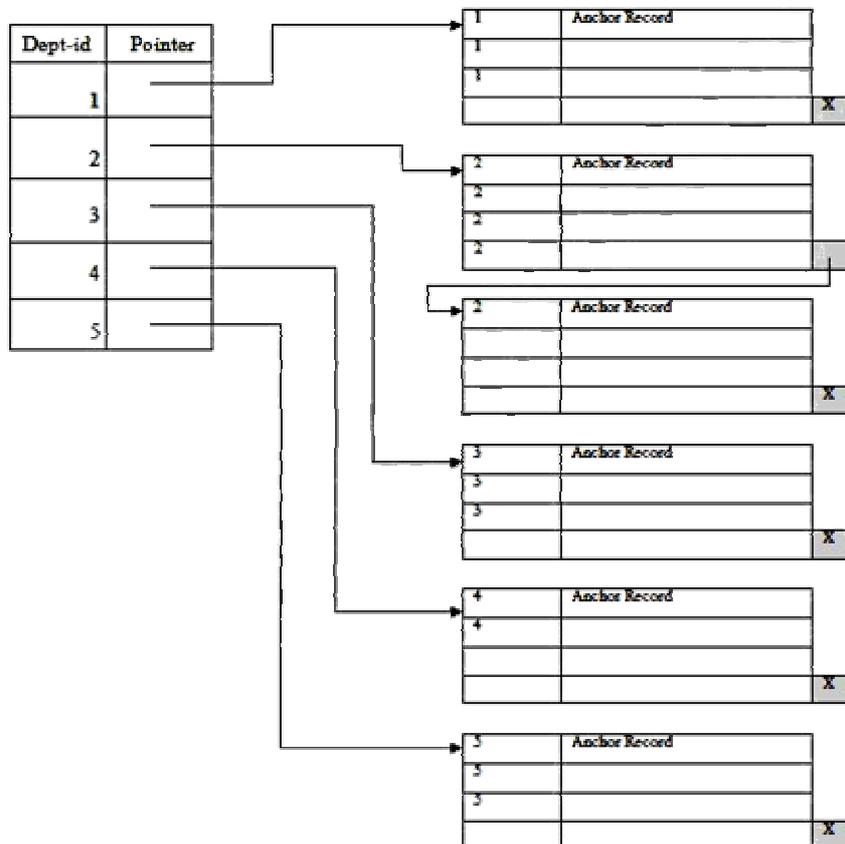
As you can see, the data blocks have been divided in to several blocks, each containing a fixed number of records (in our case 10). The pointer in the index table points to the first record of each data block, which is known as the Anchor Record for its important function. If you are searching for roll 14, the index is first searched to find out the highest entry which is smaller than or equal to 14. We have 11. The pointer leads us to roll 11 where a short sequential search is made to find out roll 14.

9.2.2 Clustering Index

It may happen sometimes that we are asked to create an index on a non-unique key, such as Dept-id. There could be several employees in each department. Here we use a clustering index, where all employees belonging to the same Dept-id are considered to be within a single cluster, and the index pointers point to the cluster as a whole.



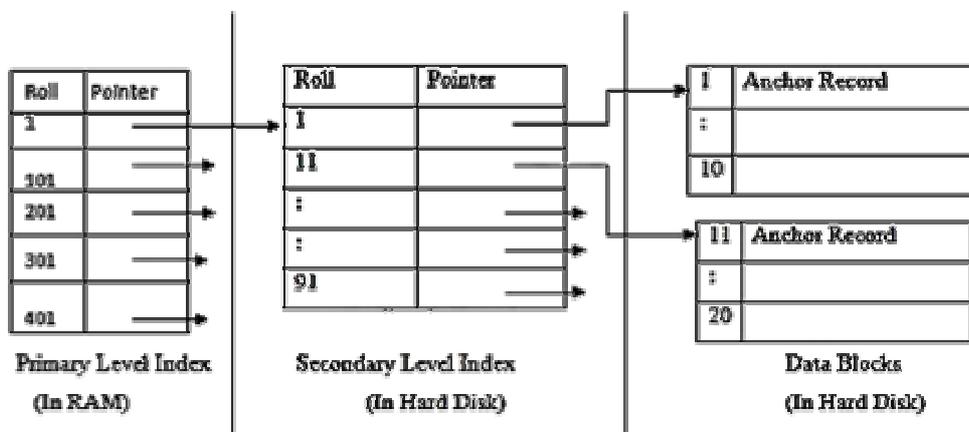
Let us explain this diagram. The disk blocks contain a fixed number of records (in this case 4 each). The index contains entries for 5 separate departments. The pointers of these entries point to the anchor record of the block where the first of the Dept-id in the cluster can be found. The blocks themselves may point to the anchor record of the next block in case a cluster overflows a block size. This can be done using a special pointer at the end of each block (comparable to the next pointer of the linked list organization). The previous scheme might become a little confusing because one disk block might be shared by records belonging to different cluster. A better scheme could be to use separate disk blocks for separate clusters. This has been explained in the next page.



In this scheme, as you can see, we have used separate disk block for the clusters. The pointers, like before, have pointed to the anchor record of the block where the first of the cluster entries would be found. The block pointers only come into action when a cluster overflows the block size, as for Dept-id 2. This scheme takes more space in the memory and the disk, but the organization is much better and cleaner looking.

9.2.3 Secondary Index

While creating the index, generally the index table is kept in the primary memory (RAM) and the main table, because of its size is kept in the secondary memory (Hard Disk). Theoretically, a table may contain millions of records (like the telephone directory of a large city), for which even a sparse index becomes so large in size that we cannot keep it in the primary memory. And if we cannot keep the index in the primary memory, then we lose the advantage of the speed of access. For very large table, it is better to organize the index in multiple levels. See the following example.



In this scheme, the primary level index, (created with a gap of 100 records, and thereby smaller in size), is kept in the RAM for quick reference. If you need to find out the record of roll 14 now, the index is first searched to find out the highest entry which is smaller than or equal to 14. We have 1. The adjoining pointer leads us to the anchor record of the corresponding secondary level index, where another similar search is conducted. This finally leads us to the actual data block whose anchor record is roll 11. We now come to roll 11 where a short sequential search is made to find out roll 14.

Check Your Progress!!!

Define the following terms:

1. Dense Primary Index
2. Sparse Primary Index
3. Clustering Index
4. Secondary Index

9.3 INDEXING STRUCTURES

9.3.1 Index in a Tree like Structure

We can use tree-like structures as index as well. For example, a binary search tree can also be used as an index. If we want to find out a particular record from a binary search tree, we have the added advantage of binary search procedure, that makes searching be performed even faster. A binary tree can be considered as a 2-way Search Tree, because it has two pointers in each of its nodes, thereby it can guide you to two distinct ways. Remember that for every node storing 2 pointers, the number of value to be stored in each node is one less than the number of pointers, i.e. each node would contain 1 value each.

9.3.2 Hash-Based Indexes

We can use hash based indices as well. Here we consider an Index as a collection of *buckets*.

A Bucket is a *Primary* page or primary page and overflow pages containing records or record-ids that share the same value of the hash function.

A Hashing function looks like as follows:

$h(r)$ = bucket in which record r belongs.

h is computed based on the *search key* fields of r .

The hash function is generated automatically by the DBMS. This kind of indexing is good for equality selection but not for range selection.

9.4 COST COMPARISON OF OPERATIONS ON DIFFERENT KINDS OF FILES

We ignore CPU costs, for simplicity:

Let

B: The number of data pages

R: Number of records per page

D: (Average) time to read or write disk page

The Cost of different operations of various kinds of files is tabulated below:

File Type	Scan	Equality Search	Range Search	Insert	Delete
Heap	BD	0:5BD	BD	2D	Search + D
Sorted	BD	Dlog2B	Dlog2B+# matches	Search + BD	Search + BD
Hashed	1:25BD	D	1:25BD	2D	Search + D

9.5 INDEX DEFINITIONS IN SQL

Creating an Index

```
CREATE [UNIQUE] INDEX INDEX_NAME
ON TABLE_NAME (COLUMN_NAME1, COLUMN_NAME2,
...)
```

For example,
create index c-index on customer(first_name)

Removing an Index

```
DROP INDEX INDEX_NAME
```

For example,
drop index c-index

Check Your Progress!!!

Explain with examples, the syntax to add and drop an index.

9.6 UNIT END EXERCISE

1. What is indexing? What is its need?
2. Explain different types of indexing?
3. Explain the different Indexing Structures.
4. Compare the cost of different operations of various file types.

9.7 FURTHER READING AND REFERENCES

Database Management Systems – Ramakrishnam, Gehrke , McGraw- Hill.

Fundamentals of Database Systems – Elmasri and Navathe, Pearson Education

Database Systems, Design, Implementation and Management – Peter Rob and Coronel, Thomson Learning

A First Course in Database Systems, Jeffrey D. Ullman, Jennifer Widom, Pearson Education



VIEWS

Unit Structure

10.0 Objectives

10.1 Introduction

10.2 Views – Advantages and Disadvantages

10.3 Creating Views

10.4 Updating a View

10.5 Dropping Views

10.6 Unit End Exercise

10.7 Further Reading and References

10.0 OBJECTIVES

In this chapter we will study another database object – the view. We will study how to create, update and delete it. We will also look at the advantages and disadvantages of views.

10.1 INTRODUCTION

A view is a permanently stored and named SQL query in the database. A view is a virtual table in the database whose contents are defined by a query. The rows and columns of a data visible through the view are the query results produced by the query that defines the view.

To the users, the view appears just like a real table with a set of named columns and rows of data – one can query a view as if one is querying a table. But unlike a real table, a view does not exist in the database as a stored set of data values.

10.2 VIEWS – ADVANTAGES AND DISADVANTAGES

Advantages

- **Security:** Every user can be given permission to access the database only through a small set of views that contain specific data the user is authorized to see. Thus, the user's access to stored data is restricted.
- **Query Simplicity:** A view can draw data from a number of tables or through a complex query and present it as a single table, thus turning such complex queries into single table query against the view.
- **Structural Simplicity:** Views can give the users a personalized view of the database structure, thus presenting the database as a set of virtual tables.
- **Insulation from change:** A view can present a consistent, unchanged image of the structure of the table, even if the underlying tables are split, restructured or renamed.
- **Data Integrity:** If the data is updated or entered through a view, the DBMS can check the data to ensure that it meets integrity constraints of the underlying table.

Disadvantages

- **Performance:** Whenever a view is queried, the DBMS translates the queries against the view into queries against the underlying tables. If the view is defined by a complex query, then even a simple query against the view becomes a complex query and it takes a long time to get executed.
- **Update restrictions:** A view can be updated only if certain conditions hold true on the query that forms the view. Such conditions are true only for simple views, thus complex views cannot be updated.

Check Your Progress!!!

1. Define a View.
2. Discuss the advantages and disadvantages of using a view.

10.3 CREATING VIEWS

The **CREATE VIEW** statement is used to create a view. The statement contains a name that has to be assigned to the view and a query that defines the contents of the view. To create a view, a user must have the appropriate system privilege to access the tables referenced in the query. The CREATE VIEW statement can optionally assign a name to each column in the newly created view. If the list of column names is omitted, each column of the view takes the name of the corresponding column in the query. If the query contains calculated columns or it produces two columns with identical names, then a list of column names must be specified. The CREATE VIEW statement has an optional WITH CHECK OPTION to ensure that all UPDATE and INSERTs on the view satisfy the condition(s) mentioned in the SQL query of the view definition i.e., the row is selectable using the view.

SYNTAX

```
CREATE VIEW view_name AS
SQL SELECT QUERY
[WITH CHECK OPTION]
```

Example

The following view is created from the Customer (CustNum, FirstName, LastName, Address, City, State, ZIP) table

```
Create view Customer_View As
Select CustNum, FirstName, LastName, Address, City
where City='Mumbai'
With Check Option
```

10.4 UPDATING A VIEW

One can insert new rows and update / delete existing rows from a view. Any such changes made in the view are also reflected in the base table(s) that are form the view.

One can update a view if the following conditions hold true on the query that forms the view:

- It should not contain the keyword DISTINCT
- It should not contain summary functions (COUNT, AVG, SUM, MAX, MIN)
- It should not contain set operators (UNION, INTERSECT, EXCEPT)

- It should not contain an ORDER BY clause.
- It should not be a join query, i.e. it should not retrieve data from multiple tables
- Its WHERE clause should not contain sub queries
- It should not contain GROUP BY or HAVING clause
- It should not contain calculated columns
- All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function

Thus, if a view satisfies all the above mentioned rules then it can be updated.

INSERTING ROWS INTO A VIEW

The following query would insert a new row in Customer_View
 insert into Customer_View values ('102', 'Ramesh', 'Gupta',
 'Bandra', 'Mumbai')

The above query would insert a row in the underlying table Customer with the values specified for CustNum, FirstName, LastName, Address and City columns and NULL values for state and ZIP columns.

The following query will not get executed because 'with check option' is used at the time of creating the view and the view displays only the customers of Mumbai city

insert into Customer_View values ('102', 'Ramesh', 'Gupta',
 'Bandra', 'Pune')

DELETING ROWS OF A VIEW

The following query would delete a row from Customer_View where CustNum is 102

delete from Customer_View where CustNum=102

The above query would delete the row from the underlying table Customer whose CustNum is 102.

UPDATING ROWS OF A VIEW

The following query would update the Address of CustNum 102

update Customer_View
 set Address = 'Khar'
 where CustNum=102

10.5 DROPPING A VIEW

A view like other database objects can be dropped if it is no longer needed. The syntax is to drop a view is:

```
DROP VIEW view_name;
```

The following query drops Customer_View

```
DROP VIEW CUSTOMER_VIEW;
```

Check Your Progress!!!

1. What conditions make a view updatable?
2. Explain with an example the syntax to create and drop a view.

10.6 UNIT END EXERCISE

Consider the following relations:

Dept(deptno, dname, location)

Emp(empno, ename, job, mgr, sal, comm., deptno, hiredate)

Write queries for the following:

1. Create a view called VIEWEMP which gives the same displays as EMP.
2. Create a view called DEPTSUM with two columns called DEPARTMENT and SUMSAL containing the name of each department and the sum of the salaries for all employees in the department.
3. Create a view called BOSS which has the name and number of each employee with the name and number of his or her manager (with blanks alongside any employee that has no manager). Give each column in the view a suitable name. Change one of the entries in the EMP table to give an employee a different manager and check this is reflected in the BOSS view.
4. Delete the DEPTSUM and BOSS views.
5. Create a view called SALES1 which has all the columns of the EMP table except the salary and commission columns, and with only the rows corresponding to employees in department number 30.

10.7 FURTHER READING AND REFERENCES

Professional SQL server 2000 Programming – Rob Vieira, Wrox Press Ltd, Shroff

SQL server 2000 black book- Patrick Dalton and Paul Whitehead, Dreamtech Press

Understanding SQL, Martin Gruber, B.P.B. Publications



STORED PROCEDURES

Unit Structure

- 11.0 Objectives
- 11.1 Introduction
- 11.2 General Syntax to create a procedure
- 11.3 How to execute a Stored Procedure?
- 11.4 Modifying a Stored Procedure
- 11.5 Benefits of Stored Procedures
- 11.6 System Stored Procedures
- 11.7 Unit End Exercise
- 11.8 Further Reading and References

11.0 OBJECTIVES

In this chapter we will study the concept of stored procedures and the benefits of the same. We will also study the syntax to create and execute a stored procedure.

11.1 INTRODUCTION

A stored procedure is a set SQL statements compiled into a single execution plan which are ready for execution. A SQL stored procedure is similar to procedures in other programming languages.

11.2 GENERAL SYNTAX TO CREATE A PROCEDURE:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[list of parameters]
IS|AS
  Declaration section
BEGIN
```

```

    Execution section
EXCEPTION
    Exception section
END;
```

The CREATE PROCEDURE statement is used to create a stored procedure. By using CREATE OR REPLACE form, the procedure is created if no other procedure with the same name exists or the existing procedure is replaced with the current code. If CREATE is used then an error would occur if an attempted to re-create it.

The list of parameters specifies what arguments /parameters the procedure can accept. The procedure can have any number of arguments. SQL allows 3 types of parameters as follows:

1. IN – Input parameter only, the value passed is input only. It is not returned to the invoking statement. All parameters are of type IN by default.
2. OUT – Output only. These parameters are used to pass values back to the invoking statement.
3. IN OUT – Both input and output. Values can be read from and passed back the invoking statement.

The execution code contains the actual code of the procedure. These are the statements that get executed when the procedure is called. The exception section is an optional block that contains code to handle exceptions that may occur in the procedure.

Example

```

Create PROCEDURE GetAllCustomers
As
BEGIN
Select * from Customers
END
```

```

CREATE PROCEDURE GetCustomerByCity(IN cityName
VARCHAR(30))
AS
BEGIN
SELECT *
FROM Customers
WHERE City = cityName;
END
```

11.3 HOW TO EXECUTE A STORED PROCEDURE?

The following are the two ways of executing a stored procedure

1. From the SQL prompt.
EXECUTE [or EXEC] procedure_name;
2. Within another procedure – simply use the procedure name.
procedure_name;

Example

```
EXECUTE GetAllCustomers
```

```
EXEC GetCustomerByCity('Mumbai')
```

11.4 MODIFYING A STORED PROCEDURE

If you need to modify an existing stored procedure, you simply replace the CREATE with ALTER.

```
ALTER PROCEDURE procedure_name [list of parameters]
AS
    Declaration section
BEGIN
    Execution section
EXCEPTION
    Exception section
END;
```

Check Your Progress!!!

1. Create a procedure that will display all employees drawing a salary more than as specified as a parameter.
2. Create a procedure that will display the details of the department whose name is passed as a parameter.

11.5 BENEFITS OF STORED PROCEDURES

- **Modular programming:** You can write a stored procedure once, then call it from multiple places in your application.
- **Performance:** Stored procedures provide faster code execution and reduce network traffic.
- **Faster execution:** Stored procedures are parsed and optimized as soon as they are created and the stored

procedure is stored in memory. This means that it will execute a lot faster than sending many lines of SQL code from your application to the SQL Server. Doing that requires SQL Server to compile and optimize your SQL code every time it runs.

- **Reduced network traffic:** If you send many lines of SQL code over the network to your SQL Server, this will impact on network performance. This is especially true if you have hundreds of lines of SQL code and/or you have lots of activity on your application. Running the code on the SQL Server (as a stored procedure) eliminates the need to send this code over the network. The only network traffic will be the parameters supplied and the results of any query.
- **Security:** Users can execute a stored procedure without needing to execute any of the statements directly. Therefore, a stored procedure can provide advanced database functionality for users who wouldn't normally have access to these tasks, but this functionality is made available in a tightly controlled way.

11.6 SYSTEM STORED PROCEDURES

SQL Server includes a large number of system stored procedures to assist in database administration tasks. Many of the tasks you can perform via Enterprise Manager can be done via a system stored procedure. For example, some of the things you can do with system stored procedures include:

- configure security accounts
- set up linked servers
- create a database maintenance plan
- create full text search catalogs
- configure replication
- set up scheduled jobs

11.7 UNIT END EXERCISE

1. What are Stored Procedures? What are their benefits?
2. Explain the syntax of creating, modifying and executing a stored procedure.
3. Write a note on System Stored Procedures.

11.8 FURTHER READING AND REFERENCES

Professional SQL server 2000 Programming – Rob Vieira, Wrox Press Ltd, Shroff

SQL server 2000 black book- Patrick Dalton and Paul Whitehead, Dreamtech Press

Understanding SQL, Martin Gruber, B.P.B. Publications



TRIGGERS IN SQL

Unit Structure

12.0 Objectives

12.1 Introduction

12.2 Trigger – Syntax

12.3 Triggers – Advantages and Disadvantages

12.5 Exercise

12.6 Further Reading and References

12.0 OBJECTIVES

In this chapter we will study another database object – triggers – their advantages and the syntax to create and use them.

12.1 INTRODUCTION

A trigger can be considered as a stored procedure that is run automatically by the database whenever some event (usually a table update) happens.

12.1 TRIGGER – SYNTAX

CREATING TRIGGERS

```
CREATE [OR REPLACE] TRIGGER trigger_name
BEFORE (or AFTER)
INSERT OR UPDATE [OF COLUMNS] OR DELETE
ON tablename
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW [WHEN (condition)]]
BEGIN
...
END;
```

The CREATE OR REPLACE syntax is same as with procedures. The BEFORE (or AFTER) in the trigger definition refers to when you want to run the trigger, either before the actual database modification (update, delete, insert) or after. The list of various statements, INSERT OR UPDATE [OF COLUMNS] OR DELETE refers to statements that would activate this trigger. You can specify all three, or just one.

[REFERENCING OLD AS o NEW AS n] – clause is used to reference the old and new values of the data being changed.

For Example

Create trigger neworder after insert on orders referencing new as new_ord for each row

```
begin
    update product p set stock=stock – new_ord.qty
where p.id = new_ord.prod_id
end
```

VIEWING TRIGGERS

One can see all user defined triggers by doing a select statement on USER_TRIGGERS table. Some of the columns of the USER_TRIGGERS table are as listed below:

Column Name	Description
TRIGGER_NAME	Name of the trigger
TRIGGER_TYPE	Type of the trigger (when it fires) - BEFORE/AFTER and STATEMENT/ROW
TRIGGERING_EVENT	Statement that will fire the trigger - INSERT,UPDATE and/or DELETE
TABLE_OWNER	Owner of the table that this trigger is associated with
TABLE_NAME	Name of the table that this trigger is associated with
COLUMN_NAME	The name of the column on which the trigger is defined over
REFERENCING_NAMES	Names used for referencing to OLD
WHEN_CLAUSE	WHEN clause must evaluate to true in order for triggering body to execute

Example

```
SELECT TRIGGER_NAME FROM USER_TRIGGERS;
```

The above statement produces the names of all triggers.

DROPPING A TRIGGER

A previously created trigger can be deleted from the database with the following command:

```
DROP TRIGGER trigger_name;
```

Example

```
DROP TRIGGER neworder
```

Check Your Progress!!!

1. Create a trigger that would increase the sales of the employee taking the order by the order amount for each new order being placed.
2. State and explain any five column names of the USER_TRIGGERS table.

12.3 TYPES OF PL/SQL TRIGGERS

Triggers can be classified in two categories depending on the level where it is triggered. They are:

1. **Row level trigger** – An event is triggered whenever a row is inserted, updated or deleted.
2. **Statement level trigger** – An event is triggered whenever a statement is executed.

12.4 TRIGGERS – ADVANTAGES AND DISADVANTAGES

Advantages

- Triggers provides an alternative way to check integrity.
- Triggers can catch the errors in business logic in the database level.
- With triggers, you don't have to wait to run the scheduled tasks. You can handle those tasks before or after changes being made to database tables.

Disadvantages

- SQL trigger only can provide extended validation and cannot replace all the validations. Some simple validations can be done in the application level.

- SQL Triggers executes invisibly from client-application which connects to the database server so it is difficult to figure out what happen underlying database layer.
- SQL Triggers run every updates made to the table therefore it adds workload to the database and cause system runs slower.

12.5 EXERCISE

1. What is a trigger?
2. What are the different types of triggers?
3. State the advantages and disadvantages of triggers.

12.6 FURTHER READING AND REFERENCES

Professional SQL server 2000 Programming – Rob Vieira, Wrox Press Ltd, Shroff

SQL server 2000 black book- Patrick Dalton and Paul Whitehead, Dreamtech Press

Understanding SQL, Martin Gruber, B.P.B. Publications

