2.1 SQL : Characteristics and Advantages

Structured Query Language (SQL) is the standard command set used to communicate with the relational database management systems. All tasks related to relational data management creating tables, querying the database for information, modifying the data in the database, deleting them, granting access to users and so on can be done using SQL.

➢ Characteristics of SQL

SQL usage by its very nature is extremly flexible. It uses a free form syntax that gives the user the ability to structure SQL statements in a way best suited to him. Each SQL request is parsed by the RDBMS before execution, to check for proper syntax and to optimize the request. Unlike certain programming languages, there is no need to start SQL statements in a particular column or be finished in a single line. The same SQL request can be written in a variety of ways.

➢ Advantages of SQL

The various advantages of SQL are :

- SQL is a high level language that provides a greater degree of abstraction than procedural languages.
- SQL enables the end-users and systems personnel to deal with a number of database management systems where it is available. Increased acceptance and availability of SQL are also in its favor.
- Applications written in SQL can be easily ported across systems. Such porting could be required when the underlying DBMS needs to be upgraded or changed.
- SQL specifies what is required and not how it should be done.
- The language while being simple and easy to learn can handle complex situations.
- All SQL operations are performed at a set level. One select statement can retrieve multiple rows, one modify statement can modify multiple rows. This set at a time feature of the SQL makes it increasingly powerful than the record at a time processing techniques employed in language like COBOL.

2.2 SQL Data Types and Literals

2.2.1 SQL Data Types

Data types are a classification of a particular type of information. It is easy for humans to distinguish between different types of data. SQL supports following data types :

1) Character (n)

This data type represents a fixed length string of exactly 'n' characters where 'n' is greater than zero and should be an integer.

Example :

```
Name character (10)
```

2) Varchar (n) or character varying (n)

This data type represents a varying length string whose maximum length is 'n' characters.

Example :

```
Name varchar (n)
```

3) Numeric (p, q)

This data type represents a decimal number 'p' digits and sign with assumed decimal point 'q' digits from the sign. Both 'p' and 'q' are integers.

Example :

```
Price numeric (6, 2)
```

4) Integer

An integer represents a signed integer decimal or binary.

Example :

```
Roll_No integer (3)
```

5) Small int

A small integer is a machine independent subset of the integer domain type.

Example :

```
Roll_No small int (3)
```

6) Real, double precision

Floating-point and double-precision floating point numbers with machine dependent precision.

7) Float (n)

A floating point number, with precision of at least n digits.

Example :

```
Rate float (5, 2)
```

8) Date

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

Example :

```
Date_of_birth    date
```

9) Time

The time of day, in hours, minutes and seconds. A variant, time (P) can be used to specify the number of fractional digits for seconds (the default being 0).

Example :

```
Arrival_time     time
```

10) Time stamp

A combination of date and time. A variant, timestamp (P), can be used to specify the number of fractional digits for seconds.

Date and time values can be specified like this :

```
date '2005-04-25'
time '09 : 10 : 25'
time stamp '2005-04-25  08 : 25 : 30.45'
```

Dates must be specified in the format - year followed by month followed by day, as shown. The seconds field of time or timestamp can have a fractional part, as in the timestamp above.

2.2.2 SQL Literals

There are four kinds of literal values supported in SQL. They are :

- Character string
- Bit string
- Exact numeric

- Approximate numeric

1) Character string

Character strings are written as a sequence of characters enclosed in single quotes. The single quote character is represented within a character string by two single quotes. Some example of character strings are :

- 'Computer Engg'
- 'Structured Query Language'

2) Bit string

A bit string is written either as a sequence of 0s and 1s enclosed in single quotes and preceded by the letter 'B' or as a sequence of hexadecimal digits enclosed in single quotes and preceded by the letter 'X' some examples are given below :

- B'1011011'
- B'1'
- B'0'
- X'A 5'
- X'1'

3) Exact numeric

These literals are written as a signed or unsigned decimal number possibly with a decimal point. Examples of exact numeric literals are given below :

- 9
- 90
- 90.00
- 0.9
- + 99.99
- – 99.99

4) Approximate numeric

Approximate numeric literals are written as exact numeric literals followed by the letter 'E', followed by a signed or unsigned integer. Some example are :

- 5E5
- 55.5E5
- + 55E–5
- 0.55E
- – 5.55E–9

2.3 SQL Operators

Operators and conditions are used to perform operations such as addition, subtraction or comparision on the data items in an SQL statement.

Different types of SQL operators are :

➢ **Arithmetic Operators**

Arithmetic operators are used in SQL expressions to add, subtract, multiply, divide and negate data values. The result of this expression is a number value.

Unary operators (B)	
+, −	Denotes a positive or negative expression
Binary operator (B)	
*	Multiplication
/	Division
+	Addition
−	Subtraction

Fig. 2.1.1 Arithmetic operators

Example :

```
update Emp_Salary
set salary = salary * 1.05 ;
```

➢ **Comparison Operators**

These are used to compare one expression with another. The comparison operators are given below :

Operator	Definition
=	Equality
!=, <>, ¬ =	Inequality
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
IN	Equal to any member of set
NOT IN	Not equal to any member of set
Is NULL	Test for nulls
Is NOT NULL	Test for anything other than nulls
LIKE	Returns true when the first expression matches the pattern of the second expression
ALL	Compares a value to every value in a list
ANY, SOME	Compares a value to each value in a list
EXISTS	True if subquery returns at least one row
BETWEEN x and y	>= x and <= y

Fig. 2.1.2 Comparison operators

Examples :

1) *Get the name of students who have secured first class.*

```
⇒ select student_name
  from student
  where percentage >= 60 and perc2entage < 67 ;
```

2) *Get the out of state students name*

```
⇒ select student_name
  from student
  where state <> 'Maharashtra' ;
```

3) *Get the names of students living in 'Pune'.*

```
⇒ select student_name
  from student
  where city = 'Pune' ;
```

4) *Display the names of students with no contact phone number.*

```
⇒ select student_name
  from student
  where Ph_No is NULL ;
```

5) *Display the names of students who have secured second class in exam.*

```
⇒ select student_name
  from student
  where percentage BETWEEN 50 AND 55 ;
```

➢ **Logical Operators**

A logical operator is used to produce a single result from combining the two separate conditions. Following figure shows logical operators and their definitions.

Operator	Definition
AND	Returns true if both component conditions are true ; otherwise returns false.
OR	Returns true if either component condition is true; otherwise returns false.
NOT	Returns true if the condition is false ; otherwise returns false.

Fig. 2.1.3 Logical operators

Examples :

1) *Display the names of students living in Pune and Bombay.*

```
⇒ select student_name
  from student
  where city = 'Pune' or city = 'Bombay' ;
```

2) *Display the names of students who have secured higher second class in exam.*

```
⇒ select student_name
  from student
  where percentage >=55 and percentage < 60 ;
```

➢ Set Operators

Set operators combine the results of two separate queries into a single result. Following table shows different s operators with definition.

Operator	Definition
UNION	Returns all distinct rows from both queries
INTERSECT	Returns common rows selected by both queries
MINUS	Returns all distinct rows that are in the first query, but not in second one.

Fig. 2.1.4 Set operators

Examples :

Consider following two relations -

Permanent_Emp {Emp_Code, Name, Salary}

Temporary_emp = {Emp_Code, Name, Daily_wages}

1) *Display name of all employees.*

```
⇒ select Name
  from Permanent_Emp
  Union
  select Name
  from Temporary_Emp ;
```

➢ Operator Precedence

Precedence defines the order that the DBMS uses when evaluating the different operators in the same expressio The DBMS evaluates operators with the highest precedence first before evaluating the operators of lowe precedence. Operators of equal precedence are evaluated from the left to right.

Fig. 2.1.5 shows the order of precedence.

Operator	Definition
:	Prefix for host variable
,	Variable separator
( )	Surrounds subqueries
"	Surrounds a literal
" "	Surrounds a table or column alias or literal text
( )	Overrides the normal operator precedence
+, –	Unary operators
*, /	Multiplication and division
+, –	Addition and subtraction
||	Character concatenation

NOT	Reverses the result of an expression
AND	True if both conditions are ture
OR	True if either conditions are ture
UNION	Returns all data from both queries
INTERSECT	Returns only rows that match both queries
MINUS	Returns only row that do not match both queries

Fig. 2.1.5 Operator precedence

2.4 DDL, DML, DCL, TCL

SQL provides set of commands for a variety of tasks including the following :

- Querying data
- Updating, inserting and deleting data
- Creating, modifying and deleting database objects
- Controlling access to the database
- Providing for data integrity and consistency.

For example, using SQL statements you can create tables, modify them, delete the tables, query the data in the tables, insert data into the tables, modify and delete the data, decide who can see the data and so on.

The SQL statement is a set of instructions to the RDBMS to perform an action. SQL statements are divided into the following categories :

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Data Query Language (DQL)
- Data Control Language (DCL)

1) Data Definition Language (DDL)

Data definition language is used to create, alter and delete database objects.

The commands used are create, alter and drop.

The principal data definition statements are :

- Create table, create view, create index
- Alter table
- Drop table, drop view, drop index

2) Data Manipulation Language (DML).

Data manipulation language commands let users insert, modify and delete the data in the database. SQL provides three data manipulation statements insert, update and delete.

3) Data Query Language (DQL)

This is one of the most commonly used SQL statements. This SQL statement enables the users to query one or more tables to get the information they want. SQL has only one data query statement 'select.'

4) Data Control Language (DCL)

The data control language consists of commands that control the user access to the database objects. Various DCL commands are : Commit, Rollback, Save point, Grant, Revoke.

2.5 Tables : Creating, Modifying, Deleting

The standard query language for relational databases is SQL (Structured Query Language). It is standardized and accepted by ANSI (American National Standards Institute). SQL is a fourth-generation high-level nonprocedural language, using a nonprocedural language query, a user requests data from the DBMS. The SQL language uses English - like commands such as CREATE, INSERT, DELETE, UPDATE, and DROP. The SQL language is standardized and its syntax is same across most DBMS packages.

➢ **Different types of SQL commands are -**

- *Data retrieval* retrives data from the database, for example, SELECT.
- *Data Manipulation Language (DML)* inserts new rows, changes existing rows, and removes unwanted rows, for example, INSERT, UPDATE, and DELETE.
- *Data Definition Language (DDL)* creates, changes, and removes a table structure, for example, CREATE, ALTER, DROP, RENAME, and TRUNCTATE.
- *Transaction Control* manages and changes logical transactions. Transactions are changes made to the data by DML statements grouped together, for example, COMMIT, SAVE POINT, and ROLLBACK.
- *Data Control Language (DCL)* gives and removes rights to database objects, for example GRANT, and REVOKE.

➢ **SQL DDL**

SQL DDL is used to define relational database of a system. The general syntax of SQL sentence is :

```
VERB (parameter 1, parameter 2, ... , parameter n);
```

In above syntax, parameters are separated by commas and the end of the verb is indicated by a semicolon. The relations are created using CREATE verb.

The different DDL commands are as follows :

- CREATE TABLE
- CREATE TABLE ... AS SELECT
- ALTER TABLE ... ADD
- ALTER TABLE ... MODIFY
- DROP TABLE

1) CREATE TABLE

This command is used to create a new relation and the corresponding syntax is :

```
CREATE TABLE relation_name
(field1 data type (size), field2 data type (size),..., fieldn data type (size));
```

Fig. 2.5.1 Syntax of CREATE TABLE

➢ **Example :**

The Modern Book House mostly supplies books to institutions which frequently buy books from them. Various relations used are Customer, Sales, Book Author, and Publisher. Design database scheme for the same.

1) The customer table definition is as follows :

```
SQL>create table Customer
   (Cust_no varchar(4) primary key,
   Cust_name varchar(25), Cust_add varchar(30),
   Cust_ph varchar(15));
```

Table created.

2) The sales table definition is as follows :

```
SQL>create table Sales
   (Cust_no varchar(4), ISBN varchar(15),
   Qty number(3),
   primary key(Cust_no, ISBN));
```

Table created.

3) The book table definition is as follows :

```
SQL>create table book
   (ISBN varchar(15) primary key,
   Title varchar(25), Pub_year varchar(4),
   Unit_price number(4),
   Author_name varchar(25));
```

Table created.

4) The author table definition is as follows :

```
SQL>create table Author
   (Author_name varchar(25) primary key,
   Country varchar(15));
```

Table created.

5) The publisher table definition is as follows :

```
SQL>create table Publisher
   (Publisher_name varchar(25) primary key,
   Pub_add varchar(30));
```

Table created.

2) CREATE TABLE ... AS SELECT

This type of create command is used to create the structure of a new table from the structure of existing table.

The generalized syntax of this form is shown in below :

```
CREATE TABLE relation_name 1
(field1, field2, ....., fieldn)
AS SELECT field1, field2, ....., fieldn
FROM relation_name 2;
```

➢ **Example :**

Create the structure for special customer from the structure of Customer table.

The required command is shown below.

```
SQL> create table Special_customer
   (Cust_no, Cust_name, Cust_add)
   as select Cust_no,  Cust_name, Cust_add
   from Customer;
```

Table created.

3) ALTER TABLE .... ADD ....

This is used to add some extra columns into an existing table. The generalized format is given below.

```
ALTER TABLE relation_name
ADD (new field1 datatype (size),
new field2 datatype (size), .....,
new fieldn datatype (size)) ;
```

➢ **Example :**

ADD customer phone number and fax number in the customer relation.

```
SQL> ALTER TABLE Customer
  ADD(Cust_ph_no varchar(15),
  Cust_fax_no varchar(15));
```

Table created.

4) ALTER TABLE .... MODIFY

This form is used to change the width as well as data type of existing relations. The generalized syntax of this form is shown below.

```
ALTER TABLE relation_name
MODIFY (field1 new data type (size), field2 new data type (size),
        ---------------
        fieldn new data type (size)) ;
```

➢ **Example :**

Modify the data type of the publication year as numeric data type.

```
SQL>ALTER TABLE Book
  MODIFY(Pub_year number(4));
```

Table created.

➢ **Restrictions of the Alter Table**

Using the alter table clause you cannot perform the following tasks :

- Change the name of the table
- Change the name of the column
- Drop a column
- Decrease the size of a column if table data exists.

5) DROP TABLE

This command is used to delete a table. The generalized syntax of this form is given below :

```
DROP TABLE relation-name
```

➢ **Example :** *Write the command for deleting special-customer relation.*

```
SQL>DROP TABLE Special_customer;
```

Table dropped.

6) Renaming a Table

You can rename a table provided you are the owner of the table. The general syntax is :

```
RENAME old table name TO new table name ;
```

➢ **Example :**

```
SQL>RENAME Test TO Test_info;
```

Table renamed.

7) Truncating a Table

Truncating a table is removing all records from the table. The structure of the table stays intact. The SQL language has a DELETE statement which can be used to remove one or more (or all) rows from a table. Truncation releases storage space occupied by the table, but deletion does not. The syntax is :

```
TRUNCATE TABLE table name ;
```

➢ **Example :**

```
SQL >TRUNCATE TABLE Student ;
```

8) Indexes

Indexes are optional structures associated with tables. We can create indexes explicitly to speed up SQL statement execution on a table. Similar to the indexes in books that helps us to locate information faster, an index provides faster access path to table data.

The index points directly to the location of the rows containing the value. Indexes are the primary means of reducing disk I/O when properly used. We create an index on a column or combinations of column.

➢ **Types of Index**

These are two types of index :

1) Simple index : It is created on a single column of a table.

Example :

```
Create index Item_index on order_details (Item_code) ;
```

2) Composite index : It is created on multiple columns of a table.

Example :

```
Create index Ordernoltem_index on Order_detail (Order_no, Item_code) ;
```

The indexes in the above examples do not enforce uniqueness i.e. the column included in the index can have duplicate values. To create a unique index, the key word 'unique' should be included in 'create index' command.

Example :

```
create unique index Client_index on Client_master (Client_no) ;
```

When the user defines a primary key or a unique key constraint, Oracle automatically creates unique indexes on the primary key column or unique key.

➢ **Dropping Indexes**

An index can be dropped by using the 'drop index' command.

Example :

```
Drop index Client_index ;
```

When the user drops the primary key, unique key constraint or the table, Oracle automatically drops the index on the primary key column, unique key or the table itself.

2.6 SQL DML Queries

SQL DML commands allows the user to manipulate the data in database. Different DML commands are explained in following sections.

➢ Basic Structure

The basic structure of an SQL expression consists of three clauses : *select*, *from*, and *where*.

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

A typical SQL query has the form :

```
select A1, A2, ....., An
from r1, r2, ....., rn
where p ;
```

Where,

A_i - represents an attribute

r_i - represents relation

P - is a predicate

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

2.6.1 SELECT Query and Clauses

2.6.1.1 SELECT Query

This command is used to display all fields/or set of selected fields for all/selected records from a relation.

➢ Different forms of select clause are given below :

- **Form 1 : Use of select clause for displaying selected fields**

Example : *Find the names of all publishers in the book relation.*

```
SQL> Select publisher_name from book;
```

Output :

PUBLISHER_NAME
PHI
Technical
Nirali
Technical
SciTech

Above query displays all publisher_names, from Book relation. Therefore, some publishers name will get displayed repeatedly. SQL allows duplicates in relations as well as in the results of SQL expressions.

- **Form 2 : Use of select for displaying distinct values**

For elimination of duplicates the keyword **distinct** is used. The above query is rewritten as,

```
SQL> select distinct publisher_name from book;
```

Output :

PUBLISHER_NAME
Nirali
PHI
SciTech
Technical

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

```
SQL> select all  publisher_name from book;
```

Output :

PUBLISHER_NAME
PHI
Technical
Nirali
Technical
SciTech

- **Form 3 : Use of select for displaying all fields**

The asterisk symbol "*" can be used to denote "all attributes". A select clause of the form select * indicates that all attributes of all relations appearing in the from clause are selected.

Example :

```
SQL> select * from book;
```

Output :

ISBN	TITLE	PUB_ YEAR	UNIT_PRICE	AUTHOR_NAME	PUBLISHER_NAME
1001	Oracle	2004	399	Arora	PHI
1002	DBMS	2004	400	Basu	Technical
2001	DOS	2003	250	Sinha	Nirali
2002	ADBMS	2004	450	Basu	Technical
2003	Unix	2000	300	Kapoor	SciTech

```
SQL> select * from author;
```

Output :

AUTHOR_NAME	COUNTRY
Arora	U.S.
Kapoor	Canada
Basu	India
Sinha	India

```
SQL> select * from publisher;
```

Output :

PUBLISHER_NAME	PUB_ADD
PHI	Delhi
Technical	Pune MainBazar
Nirali	Mumbai
SciTech	Chennai

- **Form 4 : Select clause with arithmetic expression**

The select clause may also contain arithmetic expressions involving the operators +, –, *, and/operating on constants or attributes of tuples.

Example

```
SQL> select title, unit_price * 10 from book;
```

Output :

TITLE	UNIT_PRICE * 10
Oracle	3990
DBS	4000
DOS	2500
ADBMS	4500
Unix	3000

The above query returns a relation that is the same as the book relation with attributes title as it is and unit_price, will get multiplied by 10.

2.6.1.2 The where Clause

The *where* clause is used to select specific rows satisfying given predicate.

Example : *"Find the titles of books published in year 2004".*

This query can be written in SQL as :

```
SQL> select title from book where pub_year = '2004';
```

Output :

TITLE
Oracle
DBMS
ADBMS

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

➢ **Between**

SQL includes a **between** comparison operator to specify that a value be less than or equal to some value and greater than or equal to some other value.

Example : *"Find the titles of book having price between 300 to 400".*

```
SQL> select Title from Book
     where Unit_price between 300 and 400;
```

Output :

TITLE
Oracle
DBMS
Unix

Or

```
SQL> select Title from Book
     where Unit_price >= 300 and Unit_price <= 400;
```

Output :

TITLE
Oracle
DBMS
Unix

Similarly, we can use the **not between** comparision operator.

2.6.1.3 The from Clause

The *from* clause defines a cartesian product of the relations in the clause.

Example : *"Find the titles of books with author name and country published in year 2004".*

```
SQL> select Title, Book.author_name, Country
  from Book, Author
```

```
where Book.author_name = Author.author_name
and Pub_year = '2004';
```

Output :

TITLE	AUTHOR_NAME	COUNTRY
Oracle	Arora	U.S
DBMS	Basu	India
ADBMS	Basu	India

Notice that SQL uses the notation *relation_name.attribute_name* to avoid ambiguity in cases where an attribute appears in the schema of more than one relation.

2.6.2 The Rename Operation

SQL provides a mechanism for renaming both relations and attributes. It uses the **as** clause, taking the form :

```
Old-name as new-name
```

The **as** clause can appear in both the select and from clause.

Example :

```
SQL>select Title, Unit_price * 1.15 as New_price
  from Book;
```

Output :

TITLE	NEW_PRICE
Oracle	458.85
DBMS	460
DOS	287.5
ADBMS	517.5
Unix	345

The above query calculates the new price as Unit_price * 1.15 and displays the result under the column name NEW_PRICE.

2.6.3 Tuple Variables

Tuple variables are defined in the from clause by the way of **as** clause.

Example : *"Find the titles of Books with author name and author country".*

The above query is written in SQL using tuple variable **as** :

```
SQL>select Title, B.Author_name, Country
  from Book B, Author A
  where B.Author_name = A.Author_name;
```

Output :

TITLE	AUTHOR_NAME	COUNTRY
Oracle	Arora	U.S
DBMS	Basu	India
DOS	Sinha	India
ADBMS	Basu	India
Unix	Kapoor	Canada

Tuple variables are most useful for comparing two tuples in the same relation. Suppose we want to display the book titles that have price greater than at least one book published in year 2004.

```
SQL>    select distinct B1.Title
        from Book B1, Book B2
        where B1.Unit_price > B2.Unit_price
        and B2.Pub_year = '2004';
```

Output :

TITLE
ADBMS
DBMS

2.6.4 String Operations

SQL specifies strings by enclosing them in single quotes, for example : 'DBMS'. A single quote character that is part of a string can be specified by using two single quote characters ; for example the string : "It's right" can be specified by 'Its right'.

The most commonly used operation on strings is pattern matching using the operator like. We describe patterns by using two special characters.

- Percent (%) : The % character matches any substring.
- Underscore (_) : The _ character matches any character.

Patterns are case sensitive, that is, upper case characters do not match lowercase character or vice-versa.

Example :

- 'computer%' - matches any string beginning with 'computer'.
- ' %Engg' - matches any string containing "Engg" as a substring, for example, "Computer Engg department", "Mechanical Engg".
- '_s%' - matches any string with second character 's'.
- '_ _ _' - matches any string of exactly three characters.
- '_ _ _%' - matches any string of at least three characters.

➢ **Example of SQL queries**

1) *Find the names of author's from author table where the first two characters of name are 'Ba' ;*

```
SQL>select Author_name
   from Author
   where Author_name like 'Ba%';
```

Output :

AUTHOR_NAME
Basu

2) *Select author_name from author where the second character of name is 'r' or'a' ;*

```
SQL>select Author_name
   from Author
   where Author_name like '_r%' or Author_name like '_a%';
```

Output :

AUTHOR_NAME
Arora
Kapoor
Basu

3) *Display the names of all publishers whose address includes the substring 'Main';*

```
SQL>select Publisher_name
   from Publisher
   where Pub_add like '%Main%';
```

Output :

PUBLISHER_NAME
Technical

2.6.5 Ordering the Display of Tuples

SQL uses order by clause to display the tuples in the result of the query to appear in sorted order.

➢ **Examples :**

1) *Display all titles of books with price in ascending order of titles.*

```
SQL>select Title, Unit_price
   from Book
   order by Title;
```

Output :

TITLE	UNIT_PRICE
ADBMS	450
DBMS	400
DOS	250
Oracle	399
Unix	300

2) *Display all titles of books with price, and year in descreading order of year.*

```
SQL>select Title, Unit_price, Pub_year
  from Book
  order by Pub_year desc;
```

TITLE	UNIT_PRICE	PUB_YEAR
Oracle	399	2004
DBMS	400	2004
ADBMS	450	2004
DOS	250	2003
Unix	300	2000

2.6.6 Null Values

SQL allows the use of null values to indicate absence of information about the value of an attribute.

We can use the special keyword null in a predicate to test for null value.

Let no. of records of customer relation are :

```
SQL> select * from Cust;
```

Output :

Cust_no	Cust_name	Cust_ph_no
C101	Telco	5346772
C102	Bajaj	5429810
C103	Mahindra	

Example : *Find all customers from customer relation with null values for cust_ph_no field.*

```
SQL>select Cust_name
  from Cust
  where Cust_ph_no is null;
```

Cust_name
Mahindra

➤ **Not Null :**

The predicate is not null tests for the absence of a null value.

Example : *Find all customers from customer relation where phone is not null.*

```
SQL>select cust_name
   from cust
   where cust_ph_no is not null;
```

Output :

Cust_name
Telco
Bajaj

2.6.7 Group by

Group by clause is used to group the rows based on certain criteria. For example, you can group the rows in a Book table by the publisher name, or in an employee table, you can group the employees by the department and so on. Group by is usually used in conjuction with aggregate functions like **sum, avg, min, max**, etc.

➤ **Examples :**

1) *Display total price of all books (publisherwise).*

```
SQL>select Publisher_name, sum(Unit_price) "Total Book Amount"
   from Book
   group by Publisher_name;
```

Publisher_name	Total Book Amount
Nirali	250
PHI	399
SciTech	300
Techanical	850

2) *"On every book author receives 15 % display royalty. Display total royalty amount received by each author till date"*

```
SQL>select Author_name, sum(Unit_price * 0.15) "Royalty Amount"
   from Book
   group by Author_name;
```

Output :

Author_name	Royalty Amount
Arora	59.85
Basu	127.5
Kapoor	45
Sinha	37.5

2.6.8 Having

The **having** clause tells SQL to include only certain groups produced by the **group by** clause in the query result set. **Having** is equivalent to the **where** clause and is used to specify the search criteria or search condition when **group by** clause is specified.

➢ **Examples**

1) *Display publisherwise total price of books published, except for publisher 'PHI'.*

```
SQL>select Publisher_name, sum(Unit_price) "Total Book Amount"
  from Book
  group by Publisher_name
  having Publisher_name <> 'PHI';
```

Output :

Publisher_name	Total Book Amount
Nirali	250
SciTech	300
Technical	850

2) *Display royalty amount received by those authors whose second character of name contains 'a'.*

```
SQL>select Author_name, sum(Unit_price * 0.15) "Royalty Amount"
  from Book
  group by Author_name
  having Author_name like '_a%';
```

Output :

Author_name	Royalty Amount
Basu	127.5
Kapoor	45

2.6.9 Aggregate Functions

Aggregate functions are functions that take a collection of values as input and return a single value. SQL has five built-in aggregate functions :

- Average : **avg**
- Minimum : **min**
- Maximum : **max**
- Total : **sum**
- Count : **count**

1) avg :

Syntax : avg ([Distinct | All] n)

Purpose : Returns average value of n, ignoring null values.

Example :

```
SQL>select avg(Unit_price) "Average Price"
  from Book;
```

Output :

Average Price
359.8

2) min :

Syntax : min ([Distinct | All] expr)

Purpose : Returns minimum value of expression

Example :

```
SQL> select min(Unit_price) "Minimum Price"
  from Book;
```

Output :

Minimum Price
250

3) max :

Syntax : max ([Distinct | All] expr)

Purpose : Returns maximum value of expression

Example :

```
SQL>select max(Unit_price) "Maximum Price"
  from Book;
```

Output :

Maximum Price
450

4) sum :

Syntax : sum ([Distinct | All] n)

Purpose : Returns sum of values of n.

Example :

```
SQL>select sum(Unit_price) "Total"
  from Book;
```

Output :

Total
1799

5) count :

Syntax : count ([Distinct | All] expr)

Purpose : Returns the number of rows where expression is not null.

Example :

```
SQL>select count(Title) "No. of Books"
  from Book;
```

Output :

No. of Books
5

2.6.10 Set Operations

The SQL operations **union**, **intersect**, and **except** operate on relations and correspond to the relational-algebra operations $\cup$, $\cap$ and $-$.

Consider two tables :

i) Depositor (Customer_name, Account_no)

ii) Borrower (Customer_name, Loan_no)

```
SQL> select * from Depositor;
```

Output :

Customer_name	Account_no
John	1001
Sita	1002
Vishal	1003
Ram	1004

```
SQL> select * from Borrower;
```

Output :

Customer_name	Loan_no
John	2001
Tonny	2003
Rohit	2004
Vishal	2002

2.6.10 The Union Operation

Union clause merges the output of two or more queries into a single set of rows and column.

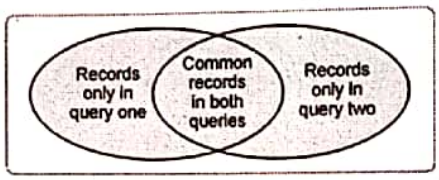

Fig. 2.6.1 Output of union clause

Output = Records only in query one + records only in query two + A single set of records which is common in both queries.

Example : *Find all custmers having a loan, an account, or both at the bank.*

```
SQL>select Customer_name
   from Borrower
   union
   select Customer_name
   from Depositor;
```

Output :

Customer_name
John
Ram
Rohit
Sita
Tonny
Vishal

The union operation automatically elimintes duplicates.

If we want to retain all duplicates, we must write union all in place of union.

```
SQL>select Customer_name
  from Borrower
  union all
  select Customer_name
  from Depositor;
```

Output :

CUSTOMER_NAME
John
Tonny
Rohit
Vishal
John
Sita
Vishal
Ram

The restrictions on using a union are as follows :

- Number of columns in all the queries should be same.
- The datatype of the column in each query must be same.
- Union cannot be used in subqueries.
- You cannot use aggregate functions in union.

2.6.10.2 The Intersect Operation

The intersect clause outputs only rows produced by both the queries intersected i.e. the intersect operation returns common records from the output of both queries.

Output = A single set of records which are common in both queries.

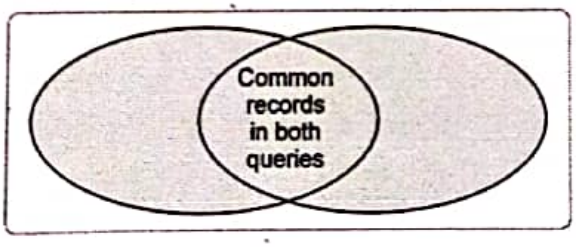

Fig. 2.6.2 Output of intersect clause

Example : *Find all customers who have an account and loan at the bank.*

```
SQL>select Customer_name
   from Depositor
   intersect
   select Customer_name
   from Borrower;
```

Output :

Customer_name
John
Vishal

The intersect operation automatically eliminates duplicates. If we want to retain all duplicates we must use intersect all in place of intersect.

```
SQL >select Customer_name
   from Depositor
   intersect all
   select Customer_name
   from Borrower ;
```

2.6.10.3 The Except Operation

The Except also called as Minus outputs rows that are in first table but not in second table.

Output = Records only in query one.

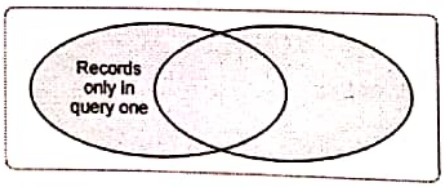

Fig. 2.6.3 Output of the except (Minus) clause

Example : *Find all customers who have an account but no loan at the bank.*

```
SQL>select Customer_name
   from Depositor
   minus
   select Customer_name
   from Borrower;
```

Output :

Customer_name
Ram
Sita

2.6.11 Nested Queries

SQL provides a mechanism for nesting subqueries. A subquery is a *select* from where expression that is nested within another query. Mostly subqueries are used to perform tests for set membership to make set comparisons and determine set cardinality.

2.6.11.1 Set Memberships

SQL uses **in** and **not in** constructs for set membership tests.

I) IN :

The **in** connective tests for set membership, where the set is a collection of values produced by a **select** clause.

Examples :

1) *Display the title, author, and publisher name of all books published in 2000, 2002 and 2004.*

```
SQL>select Title, Author_name, Publisher_name, Pub_year
   from Book
   where Pub_year in('2000','2002','2004');
```

Title	Author_name	Publisher_name	Pub_year
Oracle	Arora	PHI	2004
DBMS	Basu	Technical	2004
ADBMS	Basu	Technical	2004
Unix	Kapoor	SciTech	2000

2) *Get the details of author who have published book in year 2004.*

```
SQL>select * from Author
   where Author_name in (select Author_name
                from Book
                where Pub_year = '2004');
```

Output :

Author_name	Country
Arora	U.S
Basu	India

II) Not IN

The **not in** connective tests for the absence of set membership.

Examples

1) *Display title, author, and publisher name of all books except those which are published in year 2002, 2004 and 2005.*

```
SQL>select Title, Author_name, Publisher_name, Pub_year
  from Book
  where Pub_year not in ('2002', '2004','2005');
```

Output :

Title	Author_name	Publisher_name	Pub_year
DOS	Sinha	Nirali	2003
Unix	Kapoor	SciTech	2000

2) *Get the titles of all books written by authors not living in India.*

```
SQL>select Title
  from Book
  where Author_name not in (select Author_name from author
  where country = 'India');
```

Output :

Title
Oracle
Unix

2.6.11.2 Set Comparison

Nested subqueries are used to compare sets. SQL uses various comparison operators such as <, <=, >, >=, =, any, all, and some, etc to compare sets.

Examples

1) *Display the titles of books that have price greater than at least one book published in year 2004.*

For given example, we write the SQL query as :

```
SQL>select distinct B1.Title
  from Book B1, Book B2
  where B1.Unit_price > B2.Unit_price
  and B2.Pub_year = '2004';
```

Output :

Title
ADBMS
DBMS

SQL offers alternative style for writting preceding query.

The phrase 'greater than at least one' is represented in SQL by >some.

Using > some, we can rewrite the query as :

```
SQL>select distinct Title
   from Book
   where Unit_price >some (select Unit_pricefrom Book where Pub_year = '2004');
```

Output :

Title
ADBMS
DBMS

The subquery generates the set of all unit price values of books published in year 2004. The **>some** comparision in the **where** clause of the outer select is true if the unit price value of the tuple is greater than at least one member of the set of all unit price values of books published in year 2004.

- SQL allows **<some , >some, <=some, >=some, =some, and <> some** comparisons.
- **=some** is identical to **in** and **<> some** is identical to **not in.**
- The keyword **any** is similar to **some.**

➢ **All**

- The "> all" corresponds to the phrase 'greater than all'.
- SQL allows <all, <=all, >=all, >all, <>all, =all comparisions.
- <> all is identical to not in construct.

1) *Display the titles of books that have price greater than all the books publisher in year 2004.*

```
SQL>select distinct title
   from Book
   where Unit_price > all (select Unit_price
                  from Book
                  where Pub_year = '2004');
```

Output:

no rows selected

2) *Display the name of author who have received highest royalty amount.*

```
SQL>select Author_name
   from Book
   group by Author_name
   having sum(Unit_price * 0.15) >= all(select sum(Unit_price * 0.15) from Book group by Author_name);
```

Output :

Author_name
basu

2.6.11.3 Test for Empty Relations

Exists is a test for non empty set. It is represented by an expression of the form 'Exists (select ..... from ....) ;' an expression evaluates to true only if the result of evaluating the subquery represented by the (select ..... from .....) is nonempty.

Consider two relations :

i) Book_Info = {Book_ID, Title, Author_name, Publisher_name, Pub_year}

ii) Order_Info = {Order_no, Book_ID, Order, Date, Qty, Price}

Records of Book_Info and Order_Info relations are :

```
SQL> select * from Book_Info;
```

Book_id	Title	Author_name	Publisher_name	Pub_year
1001	Oracle	Arora	PHI	2004
1002	DBMS	Basu	Technical	2004
2001	DOS	Sinha	Nirali	2003
2002	ADBMS	Basu	Technical	2004
2003	Unix	Kapoor	SciTech	2000

```
SQL> select * from Order_Info;
```

Order_no	Book_id	Order_date	Qty	Price
1	1001	10-10-2004	100	399
2	1002	11-01-2004	50	400

Consider the following example :

```
"Get the names of all books for which order is placed".
SQL>select Title
   from Book_Info
   where exists(select * from Order_Info
         where Book_Info.book_id = Order_Info.Book_id) ;
```

Output :

Title
Oracle
DBMS

In above SQL, first the subquery 'select * from Order_Info where Book_Info.Book_Id = Order_Info.Book_I evaluated. Then the outer query is evaluated which displays the titles of books returned by inner query.

Similar to the exists we can use not exists also.

Example : *Display the titles of books for which order is not placed.*

```
SQL>select Title
   from Book_info
   where not exists(select * from Order_Info
            where Book_Info.Book_id = Order_Info.Book_Id)
```

Output :

Title
DOS
ADBMS
Unix

2.6.12 Complex Queries

Complex queries are often hard or impossible to write as a single SQL block. There are two ways for composing multiple SQL blocks to express a complex query.

i) Derived relations

ii) With clause

2.6.12.1 Derived Relations

SQL allows a subquery expression to be used in the from clause. If we use such as expression, then we must give the result relation a name, and we can rename the attributes. For renaming as clause is used.

Example :

```
(select Branch_name, avg (Balance)
   from Account
   group by Branch_name)
as result (Branch_name, Avg_balance) ;
```

This subquery generates a relation consisting of the names of all branches and their corresponding average account balances. The subquery result is named as *result* and the attributes as **Branch_name**, and **Avg_balance**.

➤ **The use of a subquery expression in the from clause is given below :**

1) *"Find the average account balance of those branches where the average account balance is greater than $1200".*

```
SQL>select Branch_name,avg(Balance)
   from(select Branch_name, avg(Balance)
        from Account
        group by Branch_name)
        as Branch_avg(Branch_name,avg(Balance))
              where Avg_balance > 1200 ;
```

2) *'Find the maximum across all branches of the total balance at each branch.*

```
SQL>select max (Tot_balance)
   from (select Branch_name, sum (Balance)
   from Account
   group by Branch_name)
   as Branch_total (Branch_name, Tot_balance);
```

2.6.12.2 The with Clause

The with clause provides a way of defining a temporary view whose definition is available only to the query in which the with clause occurs. Consider the following query, which selects accounts with the maximum balance ; if there are many accounts with the same maximum balance, all of them are selected.

```
with Max_balance (Value) as
   select max (Balance)
   from Account
select Account_number
from Account, Max_balance
where Account.Balance = Max_balance. Value ;
```

2.6.13 Database Modification using SQL Insert, Update and Delete Queries

Different operations that modify the contents of the database are :

- Delete
- Insert
- Update

2.6.13.1 Delete

A delete request is expressed in same way as a query. The delete operation is used to delete all or specific rows from database. We cannot delete values of particular attributes.

Syntax :

```
delete from r
where p
```

where,

r - relation

p - predicate

The **delete** statement first finds all tuples in *r* for which P(t) is true, and then deletes them from *r*. The where clause is omitted if all tuples in *r* are to be deleted.

A **delete** command operates only on one relation. If we want to delete tuples from several relations, we must use one **delete** command for each relation.

➢ **Examples :**

1) *Delete all books from Book relation where publishing year is less than 1997.*

```
SQL >delete from Book
   where Pub_year < 1997 ;
```

2) *Delete all books from Book relation where publishing year is between 1997 to 1999.*

```
SQL >delete from Book
  where Pub_year between 1997 and 1999 ;
```

3) *Delete all books of authors living in country 'U.K.' from Book relation.*

```
SQL > delete from Book
    where Author_name in
    (select Author_name
    from Author
    where Country = 'U.K.');
```

4) *Delete all books having price less than avg price of books.*

```
SQL >delete from Book
   where Unit_price <
   (select avg (Unit_price)
   from Book) ;
```

5) Delete all books from Book relation.

```
SQL > delete from Book ;
```

2.6.13.2 Insertion

To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. The attribute values for inserted tuples must be members of the attribute's domain.

Consider following customer relation.

```
Customer = {Cust_no, Cust_name, Cust_add, Cust_ph}
```

Example : *Insert a new customer record with customer no. as 05, customer_Name as 'Pragati', Address - 'ABC' and ph. no. as 9822398910.*

```
SQL >insert into Customer
   values (05, 'Pragati', 'ABC', 9822398910) ;
```

In above example, the values are specified in the order in which the corresponding attributes are listed in the relation schema. SQL allows the attributes to be specified as part of the insert statement.

Above query is rewritten as :

```
SQL > insert into Customer (cust_no, cust_name, cust_address, cust_ph)
values (05, 'pragati', 'ABC', 9822398910) ;
```

➢ **We can insert tuples in a relation on the basis of the result of a query.**

Consider two relations :

- Student = {Roll_No, Name, Flag}
- Defaulter_student = {Roll_No, Name}

Suppose we want to insert tuples from Student relation to Defaulter_student where flag is 'D'.

The query for above statement is :

```
SQL >insert into Defaulter_student
   select Roll_no, Name
```

```
from Student
where flag = 'D';
```

It is also possible to assign values only to particular attributes while inserting a tuple.

Example :

```
SQL >insert into Customer
    values (05, 'Pragati', 'ABC', null);
```

2.6.13.3 Updates

Using update statement, we can change value in a tuple or all the tuples of a relation.

Consider following relation :

Employee = {Emp_code, Name, Salary} :

No. of records of employee relation are :

```
SQL> select * from Employee;
```

Emp_code	Name	Salary
1	Ram	10000
2	Jim	7000
3	John	9000
4	Sita	11000

Examples :

1) *Increase salary of all employees by 15 %.*

```
SQL>update Employee
    set Salary = Salary * 1.15;
```

rows updated.

The updated salary values are :

```
SQL>select * from Employee;
```

Output :

Emp_code	Name	Salary
1	Ram	11500
2	Jim	8050
3	John	10350
4	Sita	12650

2) *Increase salary of employees who earn less than 9,000 by 15 %.*

```
SQL>update Employee
    set Salary = Salary * 1.15
    where Salary < 9000;
```

1 row updated.

The updated salary values are :

```
SQL>Select * from Employee ;
```

Emp_code	Name	Salary
1	Ram	10000
2	Jim	8050
3	John	9000
4	Sita	11000

3) *Increase salary of employees by 15 % who earn less than the average salary of employees.*

```
SQL> update Employee
  set Salary = Salary * 1.15
  where Salary < (select avg(Salary) (from Employee);
```

2 rows updated.

The updated salary values are :

```
SQL> select * from Employee;
```

Emp_code	Name	Salary
1	Ram	10000
2	Jim	8050
3	John	10350
4	Sita	11000

4) *Increase salary of employees by 15 % who earn less than 9000 and for remaining employees give 5 % salary raise.*

For above example, we require two update statements :

```
i) update Employee
   set Salary = Salary * 1.15
   where Salary < 9000 ;

ii) update employee
   set Salary = Salary * 1.05
   where Salary >= 9000 ;
```

SQL provides a case construct, which we can use to perform both the updates with a single update statement.

```
update Employee
set Salary = case
    when Salary < 9000 then
          Salary * 1.15
    else  Salary * 1.05
end ;
```

The general form of case statement is :

```
case
    when pred 1 then result 1
    when pred 2 then result 2
    --------
    when pred n then resultn
end else result
```

The operation returns result i ; where i is the first of pred 1, pred 2, ..., pred n, that is satisfied ; if none of predicate is satisfied the operation returns result 0.

2.7 Views : Creating, Dropping, Updating using Views

A view is object that gives the user a logical view of data from an underlying table or tables. You can res what users can view by allowing them to see only a few attributes/columns from a table.

Views may be created for the following reasons :

- simplifies queries
- can be queried as a base table
- provides data security

2.7.1 Creation of Views

Syntax :

```
CREATE VIEW viewname as
SELECT columnname, columnname
FROM tablename
WHERE columnname = expression list ;
```

Example :

Create view on Book table which contains two fields title, and author name.

```
SQL>create view V_Book as
   select Title,Author_name
   from Book;
View created.
SQL> select * from V_Book;
```

Output :

Title	Author_name
Oracle	Arora
DBMS	Basu
DOS	Sinha
ADBMS	Basu
Unix	Kapoor

2.7.2 Selecting Data from a View

Example : *Display all the titles of books written by author 'Basu'.*

```
SQL> select Title from V_Book
    where Author_name = 'Basu';
```

Output :

Title
DBMS
ADBMS

2.7.3 Updatable Views

Views can also be used for data manipulation i.e. the user can perform Insert, Update, and the Delete operations on the view. The views on which data manipulation can be done are called *Updatable Views*, views that do not allow data manipulation are called *Readonly Views*. When you give a view name in the Update, Insert, or Delete statement, the modifications to the data will be passed to the underlying table.

For the view to be updatable, it should meet following criteria :

- The view must be created on a single table.
- The primary key column of the table should be included in the view.
- Aggregate functions cannot be used in the select statement.
- The select statement used for creating a view should not include *Distinct*, *Group by*, or *Having* clause.
- The select statement used for creating a view should not include subqueries.
- It must not use constants, strings or value expressions like total/5.

2.7.4 Destroying a View

A view can be dropped by using the DROP VIEW command.

Syntax :

```
DROP VIEW viewname;
```

Example :

```
DROP VIEW V_Book;
```

2.8 Index

Index can be created on selected column(s) to facilitate *fast search*. Without index, a "SELECT * FROM products WHERE productID=x" needs to match with the productID column of all the records in the products table. If productID column is indexed (e.g., using a binary tree), the matching can be greatly improved (via the binary tree search).

You should index columns which are frequently used in the WHERE clause; and as JOIN columns.

The drawback about indexing is cost and space. Building and maintaining indexes require computations and memory spaces. Indexes facilitate fast search but deplete the performance on modifying the table (INSERT/UPDATE/DELETE) and need to be justified. Nevertheless, relational databases are typically optimized for queries and retrievals, but NOT for updates.

In MySQL, the keyword KEY is synonym to INDEX.

In MySQL, indexes can be built on :

1. a single column (column-index)
2. a set of columns (concatenated-index)
3. on unique-value column (UNIQUE INDEX or UNIQUE KEY)
4. on a prefix of a column for strings (VARCHAR or CHAR), e.g., first 5 characters.

There can be more than one indexes in a table. Index are automatically built on the primary-key column(s).

You can build index via CREATE TABLE, CREATE INDEX or ALTER TABLE.

```
CREATE TABLE tableName (
  ......
  [UNIQUE] INDEX|KEY indexName (columnName, ...),
-- The optional keyword UNIQUE ensures that all values in this column are distinct
    -- KEY is synonym to INDEX
  ......
PRIMAY KEY (columnName, ...)  -- Index automatically built on PRIMARY KEY column
);

CREATE [UNIQUE] INDEX indexName ON tableName(columnName, ...);

ALTER TABLE tableName ADD UNIQUE|INDEX|PRIMARY KEY indexName (columnName, ...)
SHOW INDEX FROM tableName;
```

Example :

```
mysql> CREATE TABLE employees (
        emp_no      INT UNSIGNED  NOT NULL AUTO_INCREMENT,
        name      VARCHAR(50)   NOT NULL,
        gender      ENUM ('M','F')    NOT NULL,
        birth_date      DATE          NOT NULL,
        hire_date     DATE          NOT NULL,
        PRIMARY KEY (emp_no)  -- Index built automatically on primary-key column
      );
Query OK, 0 rows affected (0.05 sec)

mysql> DESCRIBE employees;
```

Field	Type	Null	Key	Default	Extra
emp_no	int(10) unsigned	NO	PRI	NULL	auto_increment
name	varchar(50)	NO			
gender	enum('M','F')	NO			

birth_date	date	NO			
hire_date	date	NO			

5 rows in set (0.02 sec)

```
mysql> SHOW INDEX FROM employees \G
*************************** 1. row ***************************
        Table : employees
   Non_uniqu : 0
     Key_name : PRIMARY
  Seq_in_index : 1
  Column_name : emp_no
    Collation : A
  Cardinality : 0
     Sub_part : NULL
       Packed : NULL
         Null :
   Index_type : BTREE
      Comment :
1 row in set (0.06 sec)
```

```
mysql> CREATE TABLE departments (
        dept_no    CHAR(4)     NOT NULL,
        dept_name  VARCHAR(40)  NOT NULL,
        PRIMARY KEY  (dept_no),   -- Index built automatically on primary-key column
        UNIQUE INDEX (dept_name)  -- Build INDEX on this unique-value column
       );
Query OK, 0 rows affected (0.02 sec)
```

```
mysql> DESCRIBE departments;
```

Field	Type	Null	Key	Default	Extra
dept_no	char(4)	NO	PRI		
dept_name	varchar(40)	NO	UNI		

2 rows in set (0.05 sec)

```
mysql> SHOW INDEX FROM departments \G
*************************** 1. row ***************************
        Table : departments
   Non_unique : 0
     Key_name : PRIMARY
  Seq_in_index : 1
```

```
        Column_name : dept_no
          Collation : A
        Cardinality : 0
           Sub_part : NULL
             Packed : NULL
               Null :
         Index_type : BTREE
            Comment :
*************************** 2. row ***************************
              Table : departments
         Non_unique : 0
           Key_name : dept_name
        Seq_in_index : 1
        Column_name : dept_name
          Collation : A
        Cardinality : 0
           Sub_part : NULL
             Packed : NULL
               Null :
         Index_type : BTREE
            Comment :
2 rows in set (0.00 sec)
```

2.9 Predicates and Joins

Join is a query in which data is retrieved from two or more table. A join matches data from two or more t based on the values of one or more columns in each table.

➢ **Need for joins**

In a database where the tables are normalized, one table may not give you all the information about a part entity. For example, the Employee table gives only the department ID, so if you want to know the depar name and the manager name for each employee, then you will have to get the information from the Emp and Department table. In other words, you will have to join two tables. So for comprehensive data analysis must assemble data from several tables. The relational model having made you to partition your data an them in different tables for reducing data redundancy and improving data independence - relies on th operation to enable you to perform adhoc queries that will combine the related data which resides in more one table.

Different types of Joins are :

- Inner join
- Outer join
- Natural join.

2.9.1 Inner Join

Inner Join returns the matching rows from the tables that are being joined.

Consider following two relations :

i) Employee (Emp_name, City)

ii) Employee_salary (Emp_name, Department, Salary)

These two relations are shown in Fig 2.9.1 and Fig. 2.9.2.

Employee	
Emp_name	City
Hari	Pune
OM	Mumbai
Smith	Nashik
Jay	Solapur

Fig. 2.9.1 The Employee relation

Employee_salary		
Emp_name	Department	Salary
Hari	Computer	10000
Om	IT	7000
Bill	Computer	8000
Jay	IT	5000

Fig. 2.9.2 The Employee_salary relation

Example 1 :

```
select Employee.Emp_name, Employee_salary.Salary
from Employee inner join Employee_salary on
Employee.Emp_name = Employee_salary.Emp_name ;
```

Fig. 2.9.3 shows the result of above query

Emp_name	Salary
Hari	10000
Om	7000
Jay	5000

Fig. 2.9.3 The result of Employee inner join employee_salary operation with selected fields from employee and employee_salary relation

Example 2 :

```
select *
from Employee inner join Employee_salary on
Employee.Emp_name = Employee_salary.Emp_name ;
```

The result of above query is shown in Fig. 2.9.4.

Emp_name	City	Emp_name	Department	Salary
Hari	Pune	Hari	Computer	10000
Om	Mumbai	Om	IT	7000
Jay	Solapur	Jay	IT	5000

Fig. 2.9.4 The result of Employee inner join Employee_salary operation with all fields from Employee and Employee relations

As shown in Fig. 2.9.4 the result consists of the attributes of the left-hand-side relation followed by the attributes of the right-hand-side relation. Thus the Emp_name attribute appears twice in result first is from Employee relation and second is from Employee_salary relation.

2.9.2 Outer Join

When tables are joined using inner join, rows which contain matching values in the join predicate are returned. Sometimes, you may want both matching and non-matching rows returned for the tables that are being joined. This kind of an operation is known as an **outer join**.

An outer join is an extended form of the inner join. In this, the rows in one table having no matching rows in the other table will also appear in the result table with nulls.

➢ Types of outer join

The outer join can be any one of the following :

- Left outer
- Right outer
- Full outer

➢ Join types and conditions

Join operations take two relations and return another relation as the result. Each of the variants of the join operation in SQL consists of a *join type* and a *join condition*.

- ***Join type :*** It defines how tuples in each relation that do not match with any tuple in the other relation, are treated. Following Fig. 2.9.5 shows various allowed join types.

Join types
Inner join
Left outer join
Right outer join
Full outer join

Fig. 2.9.5 Join types

- ***Join conditions :*** The join condition defines which tuples in the two relations match and what attributes are present in the result of the join.

Following Fig. 2.9.6 shows allowed join conditions.

Join conditions
natural
on <predicate>
using (A1, A2, ..., An)

Fig. 2.9.6 Join conditions

The use of join condition is mandatery for outer joins, but is optional for inner join (if it is omitted, a cartesian product results). Syntactically, the keyword **natural** appears before the join type, whereas the on and **using** conditions appear at the end of the join expression.

The join condition using (A1, A2, ..., An) is similar to the natural join condition, except that the join attributes are the attributes A1, A2, ..., An, rather than all attributes that are common to both relations. The attributes A1, A2, ..., An must consist of only attributes that are common to both relations, and they appear only once in the result of the join.

Example : Employee **full outer join** Emp_salary using (Emp_name)

2.9.2.1 Left Outer Join

The left outer join returns matching rows from the tables being joined, and also non-matching rows from the left table in the result and places null values in the attributes that come from the right table.

Example :

```
select Employee.Emp_name, Salary
from Employee left outer join Employee_salary
on Employee.Emp_name = Employee_salary.Emp_name ;
```

The result of above query is shown in Fig. 2.9.7.

Emp_name	Salary
Hari	10000
Om	7000
Jay	5000
Smith	null

Fig. 2.9.7 The result of Employee left outer join Employee_salary with selected fields from Employee and Employee_salary relations.

➢ **Left outer join operation is computed as follows :**

First, compute the result of inner join as before. Then, for every tuple *t* in the left hand side relation Employee that does not match any tuple in the right-hand-side relation Employee_salary in the inner join, add a tuple *r* to the result of the join : The attributes of tuple *r* that are derived from the left-hand-side relation are filled with the from tuple *t*, and remaining attributes of *r* are filled with null values as shown in Fig. 2.9.7.

2.9.2.2 Right Outer Join

The right outer join operation returns matching rows from the tables being joined, and also non-matching rows from the right table in the result and places null values in the attributes that comes from the left table.

Example :

```
Select Employee.Emp_name, City, Salary from Employee right outer join
Employee_salary on Employee.Emp_name = Employee_salary.Emp_name ;
```

The result of preceding query is shown in Fig. 2.9.8.

Emp_name	City	Salary
Hari	Pune	10000
Om	Mumbai	7000
Jay	Solapur	5000
Bill	null	8000

Fig. 2.9.8 The result of outer join operation with selected fields from Employee and Employee_salary relations

2.10 Data Constraints

Besides the column name, datatype and length, there is other parameters, that can be passed to the DBA at cell creation time.

These data constraints are checked by DBA when data is assigned to columns. If the data being loaded fails any of the data constraint checks fired by the DBA, the DBA will not load the data into the column, reject the entered record and will display error message to the user.

These constraints are given a constraint name and the DBA stores the constraints with it's name and instructions internally along with the column itself.

The constraints can either be placed at the column level or at the table level.

1) Column level constraints

If the constraints are defined along with the column definition, it is called a *column level constraint*. Column level constraint can be applied to any column. If the constraint spans across multiple columns, the user has to use table level constraint.

2) Table level constraint

If the data constraints attached to a specific column in a table references the contents of another column in the table then the user will have to use table level constraints.

➢ **Examples of different constraints that can be applied on the table are as follows :**

- Null Value Concept
- Primary Key Concept
- Unique Key Concepts
- Default Value Concepts
- Foreign Key Concepts
- Check Integrity Constraints

2.10.1 NULL Value Concept

While creating tables, if a row lacks a data value for a particular column, that value is said to be null. Columns of any data types may contain null values unless the column was defined as not null when the table was created.

➢ **Principles of NULL Values**

- Setting a null value is appropriate when the actual value is unknown, or when a value would not be meaningful.
- A null value is not equivalent to a value of zero.
- A null value will evaluate to null in any expression. Example : null multiplied by 10 is null.

- When a column name is defined as not null, then that column becomes a mandatory column. It implies that the user is forced to enter data into that column.

2.10.2 Primary Key Concept

A primary key is one or more columns in a table used to uniquely identify each row in the table. Primary key values must not be null and must be unique across the column.

A multicolumn primary key is called a *composite primary key*.

➤ **Examples :**

1) Column level primary key constraint

```
SQL>CREATE TABLE student
   (roll_no number(5) PRIMARY KEY,
   name varchar(25) NOT NULL,
   address varchar(25) NOT NULL,
   ph_no varchar(15));
```

Table created.

2) Table level primary key constraint.

```
SQL>CREATE TABLE student1
   (roll_no number(5),
   name varchar(25) NOT NULL,
   address varchar(25) NOT NULL,
   ph_no varchar(15),
   PRIMARY KEY(roll_no));
```

Table created.

2.10.3 Unique Key Concept

Unique key is used to ensure that the information in the column for each record is unique, as with license number. A table may have many unique keys.

➤ **Example :**

```
CREATE TABLE special_customer
(customer_code number(5) PRIMARY KEY,
customer_name varchar(25) NOT NULL,
customer_address varchar(30) NOT NULL,
license_no varchar(15) constraint uk_ license_no UNIQUE);
```

2.10.4 Default Value Concept

At the time of column creation a 'default value' can be assigned to it. When the user is loading a record with values and leaves this column empty, the DBA will automatically load this column with the default value specified. The data type of the default value should match the data type of the column. You can use the default clause to specify any default value you want.

➢ **Example**

```
SQL>CREATE TABLE employee
   (emp_code number(5),
   emp_name varchar(25) NOT NULL,
   emp_address varchar(30) NOT NULL,
   ph_no varchar(15),
   married char(1) DEFAULT 'M',
   PRIMARY KEY(emp_code));
```

Table created.

2.10.5 Foreign Key Concept

Foreign key represents relationship between tables. The existence of a foreign key implies that the table with the foreign key is related to the primary key table from which the foreign key is derived.

The foreign key/references constraint

- rejects an INSERT or UPDATE of a value, if a corresponding value does not currently exist in the primary key table.
- rejects a DELETE, if it would invalidate a REFERENCES constraint.
- must refer a PRIMARY KEY or UNIQUE column in primary key table.
- will refer the PRIMARY KEY of the primary key table if no column or group of columns is specified in the constraint.
- must refer a table, not a view or cluster.
- requires that the FOREIGN KEY column(s) and the CONSTRAINT column(s) have matching data types.

➢ **Example**

```
SQL>CREATE TABLE book
   (ISBN varchar(15) PRIMARY KEY,
   title varchar(25), pub_year varchar(4),
   unit_price number(4),
   author_name varchar(25) references author,
   publisher_name varchar(25) references publisher);
```

Table created.

2.10.6 Check Integrity Constraint

The CHECK contraint defines a condition that every row must satisfy. There can be more than one CHECK constraint on a column and the CHECK constraint can be defined at column as well as the table level.

At the column level, the constraint is defined by,

```
DeptID Number (2) CONSTRAINT ck-deptID
CHECK (( DeptID >= 10) and (DeptID <= 99));
```

and at the table level by

```
CONSTRAINT ck-deptId
CHECK ((DeptID >=10) and (DeptID <= 99));
```

➢ **Following are a few examples of appropriate CHECK constraints.**

- Add CHECK constraint on the Emp_Code column of the Employee so that every Emp_Code should start with 'E'.
- Add CHECK constraint on the City column of the Employee so that only the cities 'Mumbai', 'Pune', 'Nashik', 'Solapur' are allowed.

➢ **Example**

```
SQL>CREATE TABLE  employee
   (emp_code number(5)CONSTRAINT ck_empcode CHECK (emp_code like 'E%'),
   emp_name varchar(25) NOT NULL,
   city varchar(30) CONSTRAINT ck_city
   CHECK (city IN('Mumbai', 'Pune', 'Nashik', 'Solapur' )),
   salary number(5),
   PRIMARY KEY(emp_code));
```

Table created.

➢ **Restrictions on CHECK constraints**

- The condition must be a Boolean expression that can be evaluated using the values in the row being inserted or updated.
- The condition cannot contain subqueries or sequences.
- The condition cannot include the SYS DATE, UID, USER or USER ENV SQL functions.

➢ **Defining Integrity constraints in the ALTER TABLE command :**

You can also define integrity constraints using the constraint clause in the ALTER TABLE command.

Consider following existing tables :

1) Student with definition :

```
SQL>CREATE TABLE student
   (roll_no number(3),
   name varchar(25));
```

2) Test with definition :

```
SQL>CREATE TABLE test
   (roll_no number(3),
   subject_ID number(2),
   marks number(2));
```

3) Subject_Info with definition :

```
SQL>CREATE TABLE subject_info
   (subject_ID number(2) PRIMARY KEY,
   subject_name varchar(15));
```

The following examples show the definitions of several integrity constraints.

1) *ADD PRIMARY KEY constraints on column roll_no in student table.*

```
SQL>ALTER TABLE student
    ADD PRIMARY KEY(roll_no);
```

Table altered.

2) *Modify column marks to include NOT NULL constraint.*

```
SQL>ALTER TABLE test
    MODIFY (marks number(3) NOT NULL);
```

➢ **Dropping integrity constraints in the ALTER TABLE command**

You can drop an integrity constraint if the rule that it enforces is no longer true or if the constraint is no longer needed. Drop the constraint using the ALTER TABLE command with the DROP clause.

The following examples illustrate the dropping of the integrity constraints.

1) *Drop the primary key of student table.*

```
SQL>ALTER TABLE student
    DROP PRIMARY KEY;
```

Table altered.

2) *Drop unique key constraint on column license-no in table customer.*

```
SQL>ALTER TABLE customer
    DROP CONSTRAINT license_ukey;
```

2.11 PL/SQL : Concept of Stored Procedures and Functions

PL/SQL supports two types of subprograms. They are :

- Procedures
- Functions

Procedures are usually used to perform any specific task and functions which are used to compute a value.

2.11.1 Stored Procedures

A procedure is a subprogram that performs a specific action. The syntax for creating a procedure is given below.

```
create or replace procedure <proc_name> [parameter_list] is
< local declaration >
begin
   (executable statements)
   [exception] (exception handlers)
end ;
```

A procedure has two parts :

- Specification
- Body

The procedure specification begins with the keyword **procedure** and ends with the procedure name or parameter list. The procedure body begins with the keyword **is** and ends with the keyword **end**.