**CHAPTER I Introduction to Structured Query Language**

* 1. **Normalization**

SQL allows users to access data in relational database management systems, such as Oracle, Sybase, Informix, Microsoft SQL Server, Access and others by allowing users to describe the data the user wishes to see. SQL also allows users to define the data in a database and manipulate that data.

DBMS's store data in files declared by system managers before new tables are created (on large systems), but the system stores the data in a special format and may spread data from one table over several files. In the database world, a set of files created for a database is called a tablespace. In general, on small systems, everything about a database (definitions and all table data) is kept in one file.

**Normalization** is a technique of database design that suggests that certain criteria be used when constructing a table layout (deciding what columns each table will have and creating the key structure), where the idea is to **eliminate redundancy** of non-key data across tables. Normalization is usually referred to in terms of forms and I will introduce only the first three, even though it is somewhat common to use other, more advanced forms (fourth, fifth etc.).

1. **First Normal Form** refers to moving data into separate tables where the data in each table is of a similar type and by giving each table a primary key.
2. **Second Normal Form** involves taking out data off to other tables that is only dependent of a part of the key.
3. **Third Normal Form** involves getting rid of anything in the tables that doesn't depend solely on the primary key. Only include information that is dependent on the key and move off data to other tables that are independent of the primary key and create a primary keys for the new tables.

**1.2 SQL**

There are two basic types in SQL :

1. **DDL** (Data Definition Language) refers to the Create Table statement.
2. **DML** (Data Manipulation Language) refers to the Select, Update, Insert and Delete statements.

**1.2.1 Queries**

A **single-row query** is a query that returns one row as its result and **a multiple-row query** is a query that returns more than one row as its result. Whether a query returns one row or more than one row is entirely dependent on the design (or schema) of the tables of the database. As query-writer, you must be aware of the schema, be sure to include enough conditions and structure your SQL statement properly, so that you will get the desired result (either one row or multiple rows). E.g., if you wanted to be sure that a query of the Supplier table returned only one row, consider an equal condition of the primary key-column (Supp\_no).

Three reasons immediately come to mind as to why this is important.

First, getting multiple rows when you were expecting only one, or vice-versa, may mean that the query is erroneous, that the database is incomplete, or simply, you learned something new about your data.

If you are using an update or delete statement, you had better be sure that the statement that you write performs the operation on the desired row (or rows)...or else, you might be deleting or updating more rows than you intend.

Any queries written in Embedded SQL must be carefully thought out as to the number of rows returned. If you write a single-row query, only one SQL statement may need to be performed to complete the programming logic required. If your query, on the other hand, returns multiple rows, you will have to use the Fetch statement and quite probably, some sort of looping structure in your program will be required to iterate processing on each returned row of the query.

Another design question...the term "relationships" usually refers to the relationships among primary and foreign keys between tables. This concept is important because when the tables of a relational database are designed, these relationships must be defined because they determine which columns are or are not primary or foreign keys.

* 1. **Relationships**

**1.3.1 One to One Relationship (“1 ..1”)**

A One-to-One Relationship means that you have a primary key column that is related to a foreign key column and that for every primary key value, there is one foreign key value. E.g., in the first example, the EmpAdd , we add an EmpIDNo column. Then, the EmpAdd is related to the EmpStat Table (second example table) by means of that EmpIDNo. Specifically, each employee in the EmpAdd has statistics (one row of data) in the EmpStat Table. Even though this is a contrived example, this is a "1-1" relationship.

The other two kinds of relationships may or may not use logical primary key and foreign key constraints.

**1.3.2 One to Many Relationship (“1 .. M”)**

This means that for every column value in one table, there is one or more related values in another table. Key constraints may be added to the design, or possibly just the use of some sort of identifier column may be used to establish the relationship. An example would be that for every Supp\_no in the Supplier table, there are one or more (zero is permissible too) Items bought in the Item table.

**1.3.3 Many to Many Relationship (“M .. M”)**

This relationship does not involve keys generally and usually involves identifying columns. The unusual occurence of a "M-M" means that one column in one table is related to another column in another table and for every value of one of these two columns, there are one or more related values in the corresponding column in the other table (and vice-versa), or more a common possibility, two tables have a 1-M relationship to each other (two relationships, one 1-M going each way).

**1.4 Simple Select Statement**

In a relational database, data is stored in tables. An example table would relate Social Security Number, Name and Address:

EmpAdd Table :

SSN FirstName LastName Address City State

512687458 Joe Smith 83 First Street Howard Ohio

758420012 Mary Scott 842 Vine Ave. Losantiville Ohio

102254896 Sam Jones 33 Elm St. Paris New York

876512563 Sarah Ackerman 440 U.S. 110 Upton Michigan

Now, let's say you want to see the address of each employee.

SELECT FirstName, LastName, Address, City, State

FROM EmpAdd ; [[1]](#footnote-1)\*

The following is the results of your query of the database:

First Name Last Name Address City State

Joe Smith 83 First Street Howard Ohio

Mary Scott 842 Vine Ave Losantiville Ohio

Sam Jones 33 Elm St. Paris New York

Sarah Ackerman 440 U.S. 110 Upton Michigan

To explain what you just did, you asked for the all of data in the EmpAdd and specifically, you asked for the columns called FirstName, LastName, Address, City and State. Note that column names and table names do not have spaces. They must be typed as one word; and that the statement ends with a semicolon (;). The general form for a SELECT statement, retrieving all of the rows in the table is:

SELECT ColumnName, ColumnName, ...

FROM TableName;

To get all columns of a table without typing all column names, use:

SELECT \* FROM Tablename;

Each database management system (DBMS) and database software has different methods for logging in to the database and entering SQL commands.

**1.4.1 Conditional Selection**

The WHERE clause is used only when certain rows of the table are to be displayed, based on the criteria described in that WHERE clause.

Let's look at a new example table of Employee Status:

EmpStat Table

EmpIDNo Salary Benefits Position

010 75000 15000 Manager

105 65000 15000 Manager

152 60000 15000 Manager

215 60000 12500 Manager

244 50000 12000 Staff

300 45000 10000 Staff

335 40000 10000 Staff

400 32000 7500 Entry-Level

441 28000 7500 Entry-Level

If you want to display all the detail information of the Managers then, we can add a condition using a 'where' clause in the previous query.

SELECT \*

FROM empstat

WHERE empidno = 152 ;

To check the condition, we can use the various Relational Operators which are as follows :

**1.4.2 Relational Operators**

= Equal

<> or != Not Equal

< Less Than

> Greater Than

<= Less Than or Equal To

>= Greater Than or Equal To

If you wanted to see the EMPIDNO's of those earning more than 50,000, use the following:

SELECT empidno

FROM empstat

WHERE salary >= 50000;

Notice that the >= (greater than or equal to) sign is used, as we wanted to see those who earned greater than 50,000 or equal to 50,000 listed together. This displays :

EMPIDNO

------------

010

105

152

215

244

The WHERE description, SALARY >= 50000 is known as a condition. The same can be done for text columns:

SELECT empidno

FROM empstat

WHERE position = 'Manager';

This displays the ID Numbers of all Managers. Generally, with text columns, make sure that any text that appears in the statement is surrounded by single quotes (').

**1.4.3 Compound Conditions**

We can combine multiple conditions using **'Logical Operators'** : **AND** and **OR**.

The AND operator joins two or more conditions and displays a row only if that row's data satisfies ALL conditions listed (i.e. all conditions hold true). E.g., to display all staff earning over 40,000, use:

SELECT empidno

FROM empstat

WHERE salary > 40000 AND position = 'Staff';

The OR operator joins two or more conditions, but returns a row if ANY of the conditions listed hold true. To see all those who make less than 40,000 or have less than 10,000 in benefits listed together, use the following query:

SELECT empidno

FROM empstat

WHERE salary < 40000 ORbenefits < 10000;

AND & OR can be combined e.g.

select empidno

from empstat

where position = 'Manager' **and** salary > 60000 **or** benefits > 12000;

First SQL finds the rows where the salary is greater than 60,000 and the position column is equal to Manager, then taking this new list of rows, SQL then sees if any of these rows satisfies the previous AND condition or the condition that the Benefits column is greater then 12,000. Subsequently, SQL only displays this second new list of rows, keeping in mind that anyone with Benefits over 12,000 will be included as the OR operator includes a row if either resulting condition is True. *Also note that the AND operation is done first*.

To generalize this process, SQL performs the AND operation(s) to determine the rows where the AND operation(s) hold true (remember: all of the conditions are true), then these results are used to compare with the OR conditions and only display those remaining rows where the conditions joined by the OR operator hold true.

To perform OR's before AND's, like if you wanted to see a list of employees making a large salary (>50,000) or have a large benefit package (>10,000) and that happen to be a manager, use parentheses:

SELECT empidno

FROM empstat

WHERE position = 'Manager' and (salary > 50000 or benefit > 10000);

**1.4.4 IN & BETWEEN**

Suppose we want a list of employees working as Manager or Clerk, the query would be

SELECT empidno FROM empstat

WHERE poisition = ‘Manager’ OR position = ‘Clerk’;

To simplify such compound conditions we can use operator ‘in’.

SELECT empidno

FROM empstat

WHERE position IN ('Manager', ‘Clerk’);

or to list those making greater than or equal to 30,000, but less than or equal to 50,000, use:

SELECT empidno

FROM empstat

WHERE salary BETWEEN 30000 AND 50000;

To list everyone not in this range, try:

SELECT empidno

FROM empstat

WHERE salary NOT BETWEEN 30000 AND 50000;

Similarly, NOT IN lists all rows excluded from the IN list.

**1.4.5 Using LIKE**

Look at the EmpStat Table and say you wanted to see all people whose last names started with "L"; try:

SELECT empidno

FROM empadd

WHERE lastnameLIKE 'L%';

The percent sign (%) is used to represent any possible character (number, letter, or punctuation) or set of characters that might appear after the "L". To find those people with LastName's ending in "L", use '%L', or if you wanted the "L" in the middle of the word, try '%L%'. The '%' can be used for any characters, in that relative position to the given characters. **NOT LIKE** displays rows not fitting the given description.

**1.4.6 DISTINCT and Eliminating Duplicates**

Let's say that you want to list the ID and names of only those people who have sold an antique. Obviously, you want a list where each seller is only listed once--you don't want to know how many Item a person sold, just the fact that this person sold one. This means that you will need to tell SQL to eliminate duplicate sales rows and just list each person only once. To do this, use the *DISTINCT* keyword.

First, we will need an equijoin to the Supplier table to get the detail data of the person's Lname and Fname. However, keep in mind that since the Cust\_no column in the Item table is a foreign key to the Supplier table, a seller will only be listed if there is a row in the Supplier table listing the ID and names. We also want to eliminate multiple occurences of the Cust\_no in our listing, so we use DISTINCT on the column where the repeats may occur.

**1.5 Order By Clause**

If we want the list alphabetized by position . Then, we will use the ORDER BY clause:

SELECT \*

FROM empstat

ORDER BY position;

**1.6 GROUP BY & HAVING CLAUSE**

One special use of GROUP BY is to associate an aggregate function (especially COUNT; counting the number of rows in each group) with groups of rows.

First, assume that the Item table has the Price column and each row has a value for that column. We want to see the price of the most expensive item bought by each owner. We have to tell SQL to group each owner's purchases and tell us the maximum purchase price:

SELECT cust\_no, max(price)

FROM Item

GROUP BY cust\_no;

Now, say we only want to see the maximum purchase price if the purchase is over 1000, so we use the HAVING clause:

SELECT cust\_no, max(price)

FROM Item

GROUP BY cust\_no

HAVING price > 1000;

**1.7 JOINS**

In this section, we will only discuss inner joins and equijoins, as in general, they are the most useful.

Good database design suggests that each table lists data only about a single entity and detailed information can be obtained in a relational database, by using additional tables and by using a join.

First, take a look at these example tables:

#### Supplier

Supp\_no Lname Fname

01 Jones Bill

02 Smith Bob

03 Lawson Patricia

04 Akins Jane

05 Fowler Sam

#### Customer

Cust\_no Cust\_name City

# 11 Ward New York

12 Clark Paris

13 Blake Dallas

14 Sharma Delhi

15 Black Melbourne

**Trans**

Supp\_no Cust\_no Item

01 11 Bed

02 13 Table

15 12 Chair

21 15 Mirror

50 11 Desk

01 13 Cabinet

02 15 Coffee Table

**Keys**

First, let's discuss the concept of keys.

A **Primary Key** is a column or set of columns that uniquely idenifies the rest of the data in any given row.

e.g., in the Supplier table, the Supp\_no column uniquely identifies that row.

This means two things:

1) No two rows can have the same Supp\_no and,

2) Even if two owners have the same first and last names, the Supp\_no column ensures that the two owners will not be confused with each other, because the unique Supp\_no column will be used throughout the database to track the owners, rather than the names.

A **Foreign Key** is a column in a table where that column is a primary key of another table, which means that any data in a foreign key column must have corresponding data in the other table where that column is the primary key.

In DBMS-speak, this correspondence is known as **Referential Integrity**.

E.g. in the Trans table, both the Supp\_no and Cust\_no are foreign keys.

**1.7.1 Performing a Join**

The purpose of these keys is so that data can be related across tables, without having to repeat data in every table--this is the power of relational databases. E.g., you can find the names of those who bought a chair without having to list the full name of the buyer in the Item table.

you can get the name by relating those who bought a chair with the names in the Supplier table through the use of the Supp\_no, which relates the data in the two tables.

To find the names of those who bought a chair, use the following query:

SELECT cust\_name, item

FROM Customer, Trans

WHERE Trans.cust\_no = Customer.cust\_no AND item = 'Chair';

Note the following about this query

1. Notice that both tables involved in the relation are listed in the FROM clause of the statement .
2. In the WHERE clause, first notice that the ITEM = 'Chair' part restricts the listing to those who have bought a chair.
3. Secondly, notice how the NO columns are related from one table to the next by use of the Trans.cust\_no = Customer.cust\_no clause.
4. Only where NO's match across tables and the item purchased is a chair (because of the AND), will the names from the Customer table be listed.

Because the joining condition used an equal sign, this join is called an '**EQUIJOIN'**.

Dot notation refers to prefixing the table names to column names, to avoid ambiguity, as such:

SELECT supplier.sup\_no, supplier.lname, supplier.fname

FROM supplier, Item

WHERE Item.supp\_no = supplier.supp\_no AND Item.item = 'Chair';

**1.8 Aliases**

Suppose we want to compare the values in two rows, e.g. the salaries of 2 people and we know the names of these people. Then we can scroll the same table i.e. emp table. For this we will need 2 different names for the same table which are called as aliases.

e.g. The above requirement can be satisfied by the following query :

SELECT a.ename, a.sal, b.ename, b.sal

FROM emp a, emp b

WHERE a.ename = 'BLAKE' AND b.ename = 'SMITH';

In this case 'a' and 'b' are the alias names given to the emp table.

**1.9 Subqueries**

Subquery is a form an SQL statement that appears inside another SQL statement. It is also termed as a 'Nested Query'.

Suppose we want the name of the person earning the maximum salary, we can first find out the maximum salary and then the name of the person who is earning it.

e.g.

SELECT ename, sal

FROM emp

WHERE sal = (select max(sal) FROM emp);

In this case the subquery will return the maximum salary value and the main query will use that value to test the condition in where clause.

If the user wants the information from a table dependant on the information from the another table, then also we can use subquery as follows :

Suppose we want the names of the people who are working in New york then:

SELECT ename

FROM emp

WHERE deptno = (SELECT deptno FROM dept WHERE dname = 'NEW YORK');

We can also use relational operators with subqueries.

Suppose we want to find the list of the people who are earning salary greater than the clerk who is earning hightest salary then

SELECT ename, sal

FROM emp

WHERE sal > (SELECT max(sal) FROM emp

WHERE job = 'CLERK');

**1.10 EXISTS & ALL**

EXISTS uses a subquery as a condition where, the condition is true if the subquery returns any rows and False if the subquery does not return any rows; this is a nonintuitive feature with few unique uses.

However, if a prospective customer wanted to see the list of Owners only if the shop dealt in Chairs, try:

SELECT fname, lname

FROM supplier

WHERE exists (SELECT \*

FROM Item

WHERE item = 'Chair');

If there are any Chairs in the Item column, the subquery would return a row or rows, making the EXISTS clause true, causing SQL to list the Antique Owners. If there had been no Chairs, no rows would have been returned by the outside query.

ALL is another unusual feature, as ALL queries can usually be done with different and possibly simpler methods; let's take a look at an example query:

SELECT cust\_no, item

FROM Item

WHERE price >= ALL (SELECT price

FROM Item);

This will return the largest priced item (or more than one item if there is a tie) and its buyer. The subquery returns a list of all Prices in the Item table and the outer query goes through each row of the Item table and if its Price is greater than or equal to every (or ALL) Prices in the list, it is listed, giving the highest priced Item. The reason ">=" must be used is that the highest priced item will be equal to the highest price on the list, because this Item is in the Price list.

**CHAPTER II WORKING WITH TABLES**

**2.1 Creating New Tables**

All tables within a database must be created at some point in time...

Create table table\_name

(col1 datatype(width) [constraint constraint\_name constraint] default default\_val,

col2 datatype(width) [constraint constraint\_name constraint] default default\_val,

column3…………);

**Datatypes** **:**

Some common generic data types are:

• **Char(x)** - A column of characters, where x is a number designating the maximum number of characters allowed (maximum length) in the column.

• **Integer** - A column of whole numbers, positive or negative.

• **Decimal(x, y)** -A column of decimal numbers, where x is the maximum length in digits of the decimal numbers in this column and y is the maximum number of digits allowed after the decimal point. The maximum (4,2) number would be 99.99.

• **Date** - A date column in a DBMS-specific format.

• **Logical** - A column that can hold only two values: TRUE or FALSE.

Constraints include the following possibilities (automatically enforced by the DBMS; failure causes an error to be generated):

**1. NULL or NOT NULL** Whereto accept null values or not

**2. UNIQUE** enforces that no two rows will have the same value for this column

**3. PRIMARY KEY** tells the database that this column is the primary key column. This column will always contain unique values and will be not null column. (only used if the key is a one column key, otherwise a PRIMARY KEY (column, column, ...) statement appears after the last column definition.

**4. CHECK** allows a condition to be checked for when data in that column is updated or inserted;

e.g., CHECK (PRICE > 0) causes the system to check that the Price column is greater than zero before accepting the value... sometimes implemented as the CONSTRAINT statement.

**5. DEFAULT** inserts the default value into the database if a row is inserted without that column's data being inserted; e.g., BENEFITS INTEGER DEFAULT = 10000

**6. FOREIGN KEY** works the same as Primary Key, but is followed by: REFERENCES <TABLE NAME> (<COLUMN NAME>), which refers to the referential primary key.

let's see how we would create the Orders table:

CREATE TABLE orders

(supp\_no INTEGER NOT NULL,

itemname CHAR(40) NOT NULL);

This statement gives the table name and tells the DBMS about each column in the table. Please note that this statement uses generic data types and that the data types might be different, depending on what DBMS you are using. As usual, check local listings. One other note, the NOT NULL means that the column must have a value in each row. If NULL was used, that column may be left empty in a given row.

We can create a new table based on the existing table.

CREATE table new\_table\_name (col1, col2, col3) as

(SELECT field1, field2, field3 FROM old\_table\_name [where condition] );

then For new table column names are optional if you want to keep same field names

**2.2 Altering Tables**

ALTER TABLE table\_name

[ADD (col\_name col\_datatype col\_constraint)] |

[DROP CONSTRAINT constraint\_name] |

[MODIFY existing\_col\_name new\_col\_datatype new\_constraint new default];

Let's add a column to the Item table to allow the entry of the price of a given Item:

alter table Item add (price decimal (8,2) null);

The data for this new column can be updated or inserted as shown later.

Columns cannot be removed from the existing table using this command.

**2.3 Adding Data**

To insert rows into a table, do the following:

INSERT INTO table\_name VALUES (val1, val2, val3, …..);

This inserts the data into the table, as a new row, column-by-column, in the pre-defined order.

e.g. -

INSERT INTO Item VALUES (21, 01, 'Ottoman', 200.00);

But if you want to change the order then :

INSERT INTO table\_name (col1, col2, col3, …..)

VALUES (val1, val2, val3, …..);

So, let's change the order and leave Price blank:

INSERT INTO Item (cust\_no, cust\_no, item)

VALUES (01, 21, 'Ottoman');

**2.4 Deleting Data**

Delete is used to delete one or multiple rows at a time.

DELETE FROM table\_name

[where condition];

Remember, if we don't give where clause i.e. condition, then all the rows of that table will be deleted.

Let's delete this new row back out of the database:

DELETE FROM Item

WHERE item = 'Ottoman';

But if there is another row that contains 'Ottoman', that row also will be deleted. Let's delete all rows (one, in this case) that contain the specific data we added before:

DELETE FROM Item

WHERE item = 'Ottoman' AND cust\_no = 01 AND cust\_no = 21;

**2.5 Updating Data**

UPDATE table\_name

SET col\_name1 = value1 [, col\_name2 = value2,…]

[where condition] ;

Let's update a Price into a row that doesn't have a price listed yet:

UPDATE Item SET price = 500.00 WHERE item = 'Chair';

This sets all Chair's Prices to 500.00. As shown above, more WHERE conditionals, using AND, must be used to limit the updating to more specific rows. Also, additional columns may be set by separating equal statements with commas.

**CHAPTER III INDEX, VIEWS AND OTHER FEATURES**

**3.1 Indexes**

Indexes allow a DBMS to access data quicker. The system creates this internal data structure (the index) which causes selection of rows, when the selection is based on indexed columns, to occur faster. This index tells the DBMS where a certain row is in the table given an indexed-column value, much like a book index tells you what page a given word appears.

CREATE [UNIQUE] INDEX index\_name

ON table\_name(col\_name1 [asc|dessc], col\_name2 …..);

Let's create an index for the Supp\_no in the Supplier column:

CREATE INDEX oid\_idx ON supplier (supp\_no);

Now on the names:

CREATE INDEX name\_idx ON supplier (lname, fname);

To get rid of an index, drop it:

DROP INDEX oid\_idx;

By the way, you can also "drop" a table, as well (careful!--that means that your table is deleted). In the second example, the index is kept on the two columns, aggregated together--strange behavior might occur in this situation.

Some DBMS's do not enforce primary keys; in other words, the uniqueness of a column is not enforced automatically. What that means is, if, e.g., I tried to insert another row into the Supplier table with an Supp\_no of 02, some systems will allow me to do that, even though, we do not, as that column is supposed to be unique to that table (every row value is supposed to be different). One way to get around that is to create a unique index on the column that we want to be a primary key, to force the system to enforce prohibition of duplicates:

CREATE UNIQUE index oid\_idx ON supplier (supp\_no);

**3.2 Views**

In SQL, you might have access to create views for yourself. What a view does is to allow you to assign the results of a query to a new, personal table, that you can use in other queries, where this new table is given the view name in your FROM clause. When you access a view, the query that is defined in your view creation statement is performed (generally) and the results of that query look just like another table in the query that you wrote invoking the view. E.g., to create a view:

CREATE VIEW antview AS

SELECT itemname FROM orders;

Now, write a query using this view as a table, where the table is just a listing of all Items Desired from the Orders table:

SELECT cust\_no

FROM Item, antview

WHERE itemname = item;

This query shows all Cust\_no's from the Item table where the Item in that table happens to appear in the Antview view, which is just all of the Items Desired in the Orders table. The listing is generated by going through the Antique Items one-by-one until there's a match with the Antview view. Views can be used to restrict database access, as well as, in this case, simplify a complex query.

**3.3 SET OPERATORS**

**UNION**

There are occasions where you might want to see the results of multiple queries together, combining their output; use UNION. To merge the output of the following two queries, displaying the ID's of all Buyers, plus all those who have an Order placed:

SELECT cust\_no

FROM supplier

UNION

SELECT supp\_no

FROM orders;

Notice that SQL requires that the Select list (of columns) must match, column-by-column, in data type. In this case Cust\_no and Supp\_no are of the same data type (integer). Also notice that SQL does automatic duplicate elimination when using UNION (as if they were two "sets"); in single queries, you have to use DISTINCT.

The outer join is used when a join query is "united" with the rows not included in the join and are especially useful if constant text "flags" are included. First, look at the query:

SELECT supp\_no, 'is in both Orders & Item'

FROM orders, Item

WHERE supp\_no = cust\_no

UNION

SELECT cust\_no, 'is in Item only'

FROM Item

WHERE cust\_no not in (select supp\_no

from orders);

The first query does a join to list any owners who are in both tables and putting a tag line after the ID repeating the quote. The UNION merges this list with the next list. The second list is generated by first listing those ID's not in the Orders table, thus generating a list of ID's excluded from the join query. Then, each row in the Item table is scanned and if the Cust\_no is not in this exclusion list, it is listed with its quoted tag. There might be an easier way to make this list, but it's difficult to generate the informational quoted strings of text.

This concept is useful in situations where a primary key is related to a foreign key, but the foreign key value for some primary keys is NULL. E.g., in one table, the primary key is a salesperson and in another table is customers, with their salesperson listed in the same row. However, if a salesperson has no customers, that person's name won't appear in the customer table.

The outer join is used if the listing of all salespersons is to be printed, listed with their customers, whether the salesperson has a customer or not--that is, no customer is printed (a logical NULL value) if the salesperson has no customers, but is in the salespersons table. Otherwise, the salesperson will be listed with each customer.

INTERSECT and MINUS are like the UNION statement, except that INTERSECT produces rows that appear in both queries and MINUS produces rows that result from the first query, but not the second.

**3.4 Report Generation Features:**

The COMPUTE clause is placed at the end of a query to place the result of an aggregate function at the end of a listing, like

COMPUTE SUM (price);

Another option is to use break logic: define a break to divide the query results into groups based on a column, like

BREAK ON cust\_no

Then, to produce a result after the listing of a group, use

COMPUTE SUM OF price on cust\_no

If, e.g., you used all three of these clauses (BREAK first, COMPUTE on break second, COMPUTE overall sum third), you would get a report that grouped items by their Cust\_no, listing the sum of Prices after each group of a Cust\_no's items, then, after all groups are listed, the sum of all Prices is listed, all with SQL-generated headers and lines.

**CHAPTER IV FUNCTIONS**

**4.1 Aggregate Functions**

Let us discuss five important aggregate functions: SUM, AVG, MAX, MIN and COUNT. They are called aggregate functions because they summarize the results of a query, rather than listing all of the rows.

1. **SUM( )** gives the total of all the rows, satisfying any conditions, of the given column, where the given column is numeric.
2. **AVG( )** gives the average of the given column.
3. **MAX( )** gives the largest figure in the given column.
4. **MIN( )** gives the smallest figure in the given column.
5. **COUNT(\*)** gives the number of rows satisfying the conditions.

Looking at the tables at the top of the document, let's look at three examples:

SELECT SUM(salary), AVG(salary)

FROM empstat;

This query shows the total of all salaries in the table and the average salary of all of the entries in the table.

SELECT MIN(benefits)

FROM empstat

WHERE position = 'Manager';

This query gives the smallest figure of the Benefits column, of the employees who are Managers.

SELECT count(\*)

FROM empstat

WHERE position = 'Staff';

This query tells you how many employees have Staff status (3).

* 1. **Mathematical Functions**

**ABS(X)** Absolute value-converts negative numbers to positive, or leaves positive numbers alone

**CEIL(X)** X is a decimal value that will be rounded up.

**FLOOR(X)** X is a decimal value that will be rounded down.

**GREATEST(X,Y)**  Returns the largest of the two values.

**LEAST(X,Y)** Returns the smallest of the two values.

**MOD(X,Y)** Returns the remainder of X / Y.

**POWER(X,Y)** Returns X to the power of Y.

**ROUND(X,Y)** Rounds X to Y decimal places. If Y is omitted, X is rounded to the nearest integer.

**SIGN(X)** Returns a minus if X < 0, else a plus.

**SQRT(X)** Returns the square root of X.

**4.3 Character Functions**

**LEFT(<string>,X)** Returns the leftmost X characters of the string.

**RIGHT(<string>,X)** Returns the rightmost X characters of the string.

**UPPER(<string>)** Converts the string to all uppercase letters.

**LOWER(<string>)** Converts the string to all lowercase letters.

**INITCAP(<string>)** Converts the string to initial caps.

**LENGTH(<string>)** Returns the number of characters in the string.

**<String>||<string>** Combines the two strings of text into one, concatenated string, where the first string is immediately followed by the second.

**LPAD(<string>,X,'\*')** Pads the string on the left with the \* (or whatever character is inside the quotes), to make the string X characters long.

**RPAD(<string>,X,'\*')** Pads the string on the right with the \* (or whatever character is inside the quotes), to make the string X characters long.

**SUBSTR(<string>,X,Y)** Extracts Y letters from the string beginning at position X.

**NVL(<column>,<value>)** The Null value function will substitute <value> for any NULLs for in the <column> If the current value of <column> is not NULL, NVL has no effect.

**INSTR(char1, char2[,n[,m]])** Searches char1 beginning with its nth character for the mth occurance of char2 and returns the position of the character in char1 that is the first character of this occurance.

**LTRIM(char[,set])** This function removes the characters from the left of char, with initial characterss removed up to the first character not in set.

**SOUNDEX(char)** The function returns a character string containing the phonetic representation of char. This function allows you to compare words that are spelled differently, but sound alike in English.

**DECODE(expr, search, result[,search,result]...[,default])**

The function compares expr to each search value one by one. If expr is equal to a search, Oracle returns the corresponding result. If no match is found, Oracle returns default, or, if default is omitted, returns null.

* 1. **Date Functions**

**TO\_DATE(char [,fmt])** Converts char of CHAR or VARCHAR2 datatype to a value of DATE datatype.If you omit fmt, char must be in the default date format.

**TO\_CHAR(d[,fmt])** Converts d of DATE datatype to a value of VARCHAR2 datatype in the format specified by the date format fmt. If you omit fmt, d is converted to a VARCHAR2 value in the default date format.

**LAST\_DAY(d)** Returns the date of the last day of the month that contains d.

**MONTHS\_BETWEEN(d1,d2)** Returns number of months between dates d1 and d2. If d1 is later than d2, result is positive else negative. If d1 and d2 are either the same days of the months or both last days of months, the result is always integer else Oracle calculates the fractional portion of the result based on a 31-day month.

**NEW\_TIME(d,z1,z2)** Returns the date and time in time zone z2 when date and time in time zone z1 are d. The arguments z1 and z2 can by text strings like

‘AST’ or ‘ADT’ Atlantic Standard or Daylight Time

‘BST’ or ‘BDT’ Bering Standard or Daylight Time

**NEXT\_DAY(d, char)** Returns the date of the first weekday named by char that is later than the date d. The argument char must be a day of the week. The return value has the same hours, minutes, seconds compunents as the argument d.

**ADD\_MONTHS(d,n)** Returns the date d plus n months. The argument n can be any integer.

1. \* As a style we will write all keywords in SQL statements capitalized. [↑](#footnote-ref-1)