SQL NOTES

#SQL PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a database table.
Primary keys must contain unique values.
A primary key column cannot contain NULL values.
Each table should have a primary key, and each table can have only ONE primary key.

#SQL FOREIGN KEY Constraint

A FOREIGN KEY in one table points to a PRIMARY KEY in another table.
The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.
The FOREIGN KEY constraint also prevents that invalid data form being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

#SQL UNIQUE Constraint

The UNIQUE constraint uniquely identifies each record in a database table.
The UNIQUE and PRIMARY KEY constraints both provide a guarantee for uniqueness for a column or set of columns.
A PRIMARY KEY constraint automatically has a UNIQUE constraint defined on it.
Note that you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

#CANDIDATE KEY:

All keys in a table which uniquely identifies a row are called candidate key. Primary key is good eg of candidate key. Eg employee id and pan id are both candidate key but we will select employee id as primary key because not all employee will have PAN card hence pan column will loose race for primary key.

#Characteristics of Surrogate Key

A surrogate key should have NO hidden codes or meaning.
A surrogate key should be generated automatically - preferrably by the database
A surrogate key should in fact UNIQUELY identify a row in a table.

Here a surrogate represents an object in the database itself. The surrogate is internally generated by the system and is invisible to the user or application.

* the value is unique system-wide, hence never reused;
* the value is system generated;
* the value is not manipulable by the user or application;
* the value contains no semantic meaning;
* the value is not visible to the user or application;
* the value is not composed of several values from different domains.

#JOINS

Types of join

* JOIN: Return rows when there is at least one match in both tables[INNER JOIN is the same as JOIN.]
* LEFT JOIN: Return all rows from the left table, even if there are no matches in the right table
* RIGHT JOIN: Return all rows from the right table, even if there are no matches in the left table
* FULL JOIN: Return rows when there is a match in one of the tables

#Join/Inner JOin
The INNER JOIN keyword return rows when there is at least one match in both tables. If there are rows in "Persons"
that do not have matches in "Orders", those rows will NOT be listed.

SELECT a.LastName, a.FirstName, b.OrderNo FROM Persons a INNER JOIN Orders b
ON a.P_Id=b.P_Id ORDER BY a.LastName

#FULL JOIN
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo FROM Persons FULL JOIN Orders
ON Persons.P_Id=Orders.P_Id ORDER BY Persons.LastName

eg Table Person has 5 rows and table Order has 5 rows top 3 are matching with each other rest two are not matching then in inner join result will contain only 3 rows but in case of full join result will contain 3+2+2=7 rows with few data missing for last two entries.

#Cross join
CROSS JOIN returns the Cartesian product of rows from tables in the join. In other words, it will produce rows which combine each row from the first table with each row from the second table. [5]
Example of an explicit cross join:
SELECT * FROM employee CROSS JOIN department;

#Backup copy
Make a Backup Copy - Now we want to make an exact copy of the data in our "Persons" table.
We use the following SQL statement:
SELECT * INTO Persons_Backup FROM Persons

SELF JOIN:
We use Self Join, if we have a table that references itself. For example, In the Employee Table below MANAGERID column references EMPLOYEEID column. So the table is said to referencing itself. This is the right scenario where we can use Self Join. Now I want to write a query that will give me the list of all Employee Names and their respective Manager Names. In order to achieve this I can use Self Join. In the Table below,Raj is the manager for Pete,Prasad and Ben. Ravi is the manager for Raj and Mary. Ravi does not have a manager as he is the president of the Company. Employee Table

	EMPLOYEEID	NAME	MANAGERID
1	101	Mary	102
2	102	Ravi	NULL
3	103	Raj	102
4	104	Pete	103
5	105	Prasad	103
6	106	Ben	103

The query below is an example of Self Join. Both E1 and E2 refer to the same Employee Table. In this query we are joining the Employee Table with itself.

```
SELECT E1.[NAME],E2.[NAME] AS [MANAGER NAME]
FROM EMPLOYEE E1
INNER JOIN EMPLOYEE E2
ON E2.EMPLOYEEID =E1.MANAGERID
```

If we run the above query we only get 5 rows out of the 6 rows as shown in Results1 below.

	NAME	MANAGER NAME
1	Mary	Ravi
2	Raj	Ravi
3	Pete	Raj
4	Prasad	Raj
5	Ben	Raj

#VIEW

SQL CREATE VIEW Statement

In SQL, a view is a virtual table based on the result-set of an SQL statement. A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

SQL CREATE VIEW Syntax

```
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition
```

#SQL BASIC

1>The SQL SELECT DISTINCT Statement

In a table, some of the columns may contain duplicate values. This is not a problem, however, sometimes you will want to list only the different (distinct) values in a table.

The DISTINCT keyword can be used to return only distinct (different) values.

```
SELECT DISTINCT column_name(s) FROM table_name
```

eg: select distinct(firstname) from table

2>Insert

```
INSERT INTO table_name VALUES (value1, value2, value3,...)
```

The second form specifies both the column names and the values to be inserted:

```
INSERT INTO table_name (column1, column2, column3,...) VALUES (value1, value2, value3,...)
```

3>Update

```
UPDATE table_name SET column1=value, column2=value2,... WHERE some_column=some_value
```

update abc set name='abc' where id=10

4>Delete

```
DELETE FROM table_name WHERE some_column=some_value
```

delete from abc where id>10

5>SQL COUNT() Function

The COUNT() function returns the number of rows that matches a specified criteria.

SQL COUNT(column_name) Syntax

The COUNT(column_name) function returns the number of values (NULL values will not be counted) of the specified column:

SELECT COUNT(column_name) FROM table_name

Eg:
Select count(*) from master_organisation : total no of rows eg 200
Select count(tan) from master_organisation : total no. of rows eg 200
Select count(distinct tan) from master_organisation : total no of rows with only unique tan eg 100 rest are duplicate.

6>FIRST() : it will fetch first row from table
Select First(name) from employee
Same is LAST()

7>MAX()
Select MAX(salary) from employee : employee with max salary will get selected
Same : MIN()

8>SUM() : select SUM(salary) from employee >>gv sum of all the salary

9>GROUP BY
The GROUP BY Statement
The GROUP BY statement is used in conjunction with the aggregate functions to group the result-set by one or more columns.
SQL GROUP BY Syntax
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name

SELECT Customer,SUM(OrderPrice) FROM Orders GROUP BY Customer

GROUP BY More Than One Column
We can also use the GROUP BY statement on more than one column, like this:
SELECT Customer,OrderDate,SUM(OrderPrice) FROM Orders GROUP BY Customer,OrderDate

10>HAVING
The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

SQL HAVING Syntax
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
HAVING aggregate_function(column_name) operator value

SELECT Customer,SUM(OrderPrice) FROM Orders GROUP BY Customer HAVING SUM(OrderPrice)<2000

11>UCASE()/LCASE()
SELECT UCASE(LastName) as LastName,FirstName FROM Persons
Ucase: convert text into upper case same with Lcase
12>MID() : to extract letters from a column
SELECT MID(City,1,4) as SmallCity FROM Persons

City name = Hyderabad
o/p will be : Hyde

13>LEN() : to find length of string
Select len(city) from <table>
Eg for “New Delhi” o/p will be : 9

14>ROUND()
SELECT ROUND(column_name,decimals) FROM table_name
Eg:
Price=100.456
Select round(Price,1) from table output=100.4

Database Normalization Basics

* database normalization
* 1nf
* 2nf
* 3nf
* bcnf

In this article, we'll introduce the concept of normalization and take a brief look at the most common normal forms. Future articles will provide in-depth explorations of the normalization process.

What is Normalization?
Normalization is the process of efficiently organizing data in a database. There are two goals of the normalization process: eliminating redundant data (for example, storing the same data in more than one table) and ensuring data dependencies make sense (only storing related data in a table). Both of these are worthy goals as they reduce the amount of space a database consumes and ensure that data is logically stored.
The Normal Forms
The database community has developed a series of guidelines for ensuring that databases are normalized. These are referred to as normal forms and are numbered from one (the lowest form of normalization, referred to as first normal form or 1NF) through five (fifth normal form or 5NF). In practical applications, you'll often see 1NF, 2NF, and 3NF along with the occasional 4NF. Fifth normal form is very rarely seen and won't be discussed in this article.

Before we begin our discussion of the normal forms, it's important to point out that they are guidelines and guidelines only. Occasionally, it becomes necessary to stray from them to meet practical business

requirements. However, when variations take place, it's extremely important to evaluate any possible ramifications they could have on your system and account for possible inconsistencies. That said, let's explore the normal forms.

[1]First Normal Form (1NF)

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

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

[2]Second Normal Form (2NF)

Second normal form (2NF) further addresses the concept of removing duplicative data:

* Meet all the requirements of the first normal form.
* Remove subsets of data that apply to multiple rows of a table and place them in separate tables.
* Create relationships between these new tables and their predecessors through the use of foreign keys.

[3]Third Normal Form (3NF)

Third normal form (3NF) goes one large step further:

* Meet all the requirements of the second normal form.
* Remove columns that are not dependent upon the primary key.

[4]Boyce-Codd Normal Form (BCNF or 3.5NF)

The Boyce-Codd Normal Form, also referred to as the "third and half (3.5) normal form", adds one more requirement:

* Meet all the requirements of the third normal form.
* Every determinant must be a candidate key.

[5]Fourth Normal Form (4NF)

Finally, fourth normal form (4NF) has one additional requirement:

* Meet all the requirements of the third normal form.
* A relation is in 4NF if it has no multi-valued dependencies.

Remember, these normalization guidelines are cumulative. For a database to be in 2NF, it must first fulfill all the criteria of a 1NF database.