Chapter 4

SQL (Structured Query Language)

-Niraj K.C.

Table of Content

- Background, Basic structure,
- Set operation, aggregate functions, null values,
- Nested sub queries, views,
- Modification of database, joined relationship,
- Data-definition Language

SQL???

- SQL is Structured Query Language, which is a computer language for storing, manipulating and retrieving data stored in relational database.
- SQL is the standard language for Relation Database System. All relational database management systems like MySQL, MS Access, Oracle, Sybase, Informix, postgres and SQL Server use SQL as standard database language.
- Also, they are using different dialects/languages, such as:
  - MS SQL Server using T-SQL,
  - Oracle using PL/SQL,
  - MS Access version of SQL is called JET SQL (native format) etc

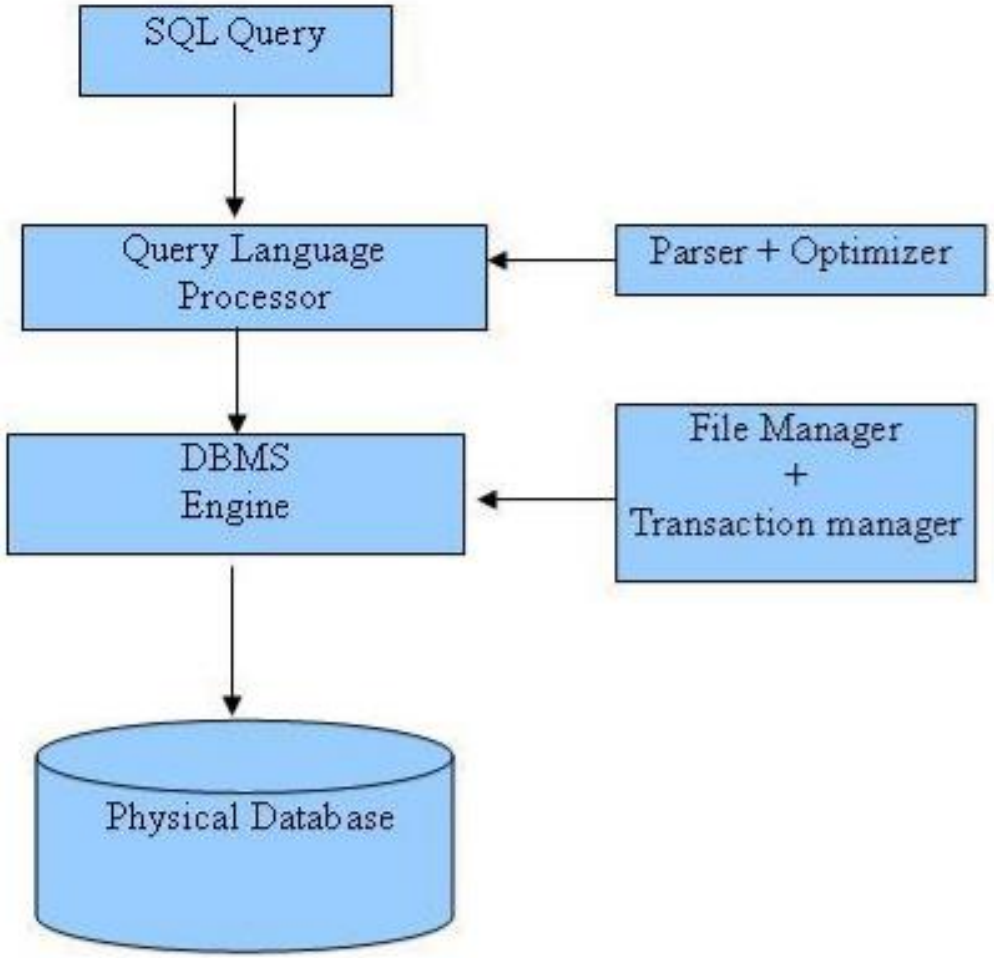
Why SQL?

- Allows users to access data in relational database management systems.
- Allows users to describe the data.
- Allows users to define the data in database and manipulate that data.
- Allows users to create and drop databases and tables
- Allows users to create view, stored procedure, functions in a database.
- Allows users to set permissions on tables, procedures and views

History

- 1970 Dr. E. F. "Ted" of IBM is known as the father of relational databases. He described a relational model for databases.
- 1974 Structured Query Language appeared.
- 1978 IBM worked to develop Codd's ideas and released a product named System/R.
- 1986 IBM developed the first prototype of relational database and standardized by ANSI.
- The first relational database was released by Relational Software and its later becoming Oracle.

SQL Process/Architecture



SQL Commands

DDL - Data Definition Language:

Command	Description
CREATE	Creates a new table, a view of a table, or other object in database
ALTER	Modifies an existing database object, such as a table.
DROP	Deletes an entire table, a view of a table or other object in the database.

DML - Data Manipulation Language:

Command	Description
INSERT	Creates a record
UPDATE	Modifies records
DELETE	Deletes records

DCL - Data Control Language:

Command	Description
GRANT	Gives a privilege to user
REVOKE	Takes back privileges granted from user

DQL - Data Query Language:

Command	Description
SELECT	Retrieves certain records from one or more tables

NULL value

- A NULL value in a table is a value in a field that appears to be blank, which means a field with a NULL value is a field with no value.
- It is very important to understand that a NULL value is different than a zero value or a field that contains spaces.
- A field with a NULL value is one that has been left blank during record creation.

SQL Constraints

- Constraints are the rules enforced on data columns on table.
- Constraints are used to limit the type of data that can go into a table.
- Constraints ensures the accuracy and reliability of the data in the database.
- Constraints could be column level or table level,
  - Column level constraints are applied only to one column
  - Table level constraints are applied to the whole table.

Commonly Used Constraints available in SQL

- **NOT NULL** Constraint: Ensures that a column cannot have NULL value.
- **DEFAULT** Constraint: Provides a default value for a column when none is specified.
- **UNIQUE** Constraint: Ensures that all values in a column are different.
- **PRIMARY Key**: Uniquely identified each rows/records in a database table.
- **FOREIGN Key**: Uniquely identified a rows/records in any another database table.
- **CHECK Constraint**: The CHECK constraint ensures that all values in a column satisfy certain conditions.
- **INDEX**: Use to create and retrieve data from the database very quickly.

Example 1:

For example, the following SQL creates a new table called CUSTOMERS and adds five columns, three of which, ID and NAME and AGE, specify not to accept NULLs:

```
CREATE TABLE CUSTOMERS(
    ID   INT                  NOT NULL,
    NAME VARCHAR (20)     NOT NULL,
    AGE  INT                  NOT NULL,
    ADDRESS  CHAR (25) ,
    SALARY   DECIMAL (18, 2),
    PRIMARY KEY (ID)
);
```

If CUSTOMERS table has already been created, then to add a NOT NULL constraint to SALARY column in Oracle and MySQL, you would write a statement similar to the following:

```
ALTER TABLE CUSTOMERS
   MODIFY SALARY  DECIMAL (18, 2) NOT NULL;
```

Example 2

- The following SQL creates a new table called CUSTOMERS and adds five columns. Here, SALARY column is set to 5000.00 by default, so in case INSERT INTO statement does not provide a value for this column. then by default this column would be set to 5000.00.

```
CREATE TABLE CUSTOMERS(
    ID   INT              NOT NULL,
    NAME VARCHAR (20)     NOT NULL,
    AGE  INT              NOT NULL,
    ADDRESS  CHAR (25) ,
    SALARY   DECIMAL (18, 2) DEFAULT 5000.00,
    PRIMARY KEY (ID)
);
```

If CUSTOMERS table has already been created, then to add a DEFAULT constraint to SALARY column, you would write a statement similar to the following:

```
ALTER TABLE CUSTOMERS
  MODIFY SALARY  DECIMAL (18, 2) DEFAULT 5000.00;
```

Drop Default Constraint: To drop a DEFAULT constraint, use the following SQL:

```
ALTER TABLE CUSTOMERS
  ALTER COLUMN SALARY DROP DEFAULT;
```

UNIQUE Constraint: The UNIQUE Constraint prevents two records from having identical values in a particular column.

In the CUSTOMERS table, for example, you might want to prevent two or more people from having identical age.

Example 3

- For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, AGE column is set to UNIQUE, so that you can not have two records with same age:

```
CREATE TABLE CUSTOMERS(
 ID   INT                                                    NOT NULL,
 NAME VARCHAR (20)                                           NOT NULL,
      AGE  INT                                               NOT NULL,
      ADDRESS  CHAR (25) ,
      SALARY   DECIMAL (18, 2),
      PRIMARY KEY (ID)
);
```

If CUSTOMERS table has already been created, then to add a UNIQUE constraint to AGE column, you would write a statement similar to the following:

```
ALTER TABLE CUSTOMERS
   MODIFY AGE INT NOT NULL UNIQUE;
```

You can also use following syntax, which supports naming the constraint in multiple columns as well:

```
ALTER TABLE CUSTOMERS
   ADD CONSTRAINT myUniqueConstraint UNIQUE(AGE, SALARY);
```

DROP a UNIQUE Constraint: To drop a UNIQUE constraint, use the following SQL:

```
ALTER TABLE CUSTOMERS
   DROP CONSTRAINT myUniqueConstraint;
```

If you are using MySQL, then you can use the following syntax:

```
ALTER TABLE CUSTOMERS
   DROP INDEX myUniqueConstraint;
```

SQL Joins

- SQL Joins clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

Consider the following two tables,

CUSTOMERS table

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Another table is ORDERS

OID	DATE	CUSTOMER_ID	AMOUNT
102	2009-10-08 00:00:00	3	3000
100	2009-10-08 00:00:00	3	1500
101	2009-11-20 00:00:00	2	1560
103	2008-05-20 00:00:00	4	2060

Now, let us join these two tables in our SELECT statement as follows:

SQL> SELECT ID, NAME, AGE, AMOUNT FROM CUSTOMERS, ORDERS WHERE CUSTOMERS.ID = ORDERS.CUSTOMER_ID;

ID	NAME	AGE	AMOUNT
3	kaushik	23	3000
3	kaushik	23	1500
2	Khilan	25	1560

SQL Join Types

- INNER JOIN: returns rows when there is a match in both tables.
- LEFT JOIN: returns all rows from the left table, even if there are no matches in the right table.
- RIGHT JOIN: returns all rows from the right table, even if there are no matches in the left table.
- FULL JOIN: returns rows when there is a match in one of the tables.
- CARTESIAN JOIN: returns the Cartesian product of the sets of records from the two or more joined tables.

INNER JOIN

- The most frequently used and important of the joins is the INNER JOIN. They are also referred to as an EQUIJOIN.
- The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate.
- The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate.
- When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

Syntax: The basic syntax of INNER JOIN is as follows:

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

Example 4:

Consider the following two tables, (a) CUSTOMERS table is as follows

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

(b) Another table is ORDERS as follows:

OID	DATE	ID	AMOUNT
102	2009-10-08 00:00:00	3	3000
100	2009-10-08 00:00:00	3	1500
101	2009-11-20 00:00:00	2	1560
103	2008-05-20 00:00:00	4	2060

Now, let us join these two tables using INNER JOIN as follows:

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS
     INNER JOIN ORDERS
     ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

LEFT JOIN

- The SQL LEFT JOIN returns all rows from the left table, even if there are no matches in the right table.
- This means that if the ON clause matches 0 (zero) records in right table, the join will still return a row in the result, but with NULL in each column from right table.
- This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

Syntax:

The basic syntax of **LEFT JOIN** is as follows:

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

Now, let us join these two tables using LEFT JOIN as follows:

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS
     LEFT JOIN ORDERS
     ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  1 | Ramesh   |   NULL | NULL                |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|  5 | Hardik   |   NULL | NULL                |
|  6 | Komal    |   NULL | NULL                |
|  7 | Muffy    |   NULL | NULL                |
+----+----------+--------+---------------------+
```

RIGHT JOIN

- The SQL RIGHT JOIN returns all rows from the right table, even if there are no matches in the left table.
- This means that if the ON clause matches 0 (zero) records in left table, the join will still return a row in the result, but with NULL in each column from left table.
- This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.

Syntax:

The basic syntax of **RIGHT JOIN** is as follows:

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

Now, let us join these two tables using RIGHT JOIN as follows:

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS
     RIGHT JOIN ORDERS
     ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

FULL JOIN

- The SQL FULL JOIN combines the results of both left and right outer joins.
- The joined table will contain all records from both tables, and fill in NULLs for missing matches on either side.

Syntax:

The basic syntax of **FULL JOIN** is as follows:

```
SELECT table1.column1, table2.column2...
FROM table1
FULL JOIN table2
ON table1.common_filed = table2.common_field;
```

Now, let us join these two tables using FULL JOIN as follows:

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS
     FULL JOIN ORDERS
     ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:

```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|    1 | Ramesh   |   NULL | NULL                |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|    5 | Hardik   |   NULL | NULL                |
|    6 | Komal    |   NULL | NULL                |
|    7 | Muffy    |   NULL | NULL                |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

CARTESIAN JOIN

- The CARTESIAN JOIN or CROSS JOIN returns the Cartesian product of the sets of records from the two or more joined tables.
- Thus, it equates to an inner join where the join-condition always evaluates to True or where the join condition is absent from the statement.

Syntax:

The basic syntax of **INNER JOIN** is as follows:

```
SELECT table1.column1, table2.column2...
FROM  table1, table2 [, table3 ]
```

Now, let us join these two tables using INNER JOIN as follows:

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
     FROM CUSTOMERS, ORDERS;
```

This would produce the following result:

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  1 | Ramesh   |   3000 | 2009-10-08 00:00:00 |
|  1 | Ramesh   |   1500 | 2009-10-08 00:00:00 |
|  1 | Ramesh   |   1560 | 2009-11-20 00:00:00 |
|  1 | Ramesh   |   2060 | 2008-05-20 00:00:00 |
|  2 | Khilan   |   3000 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1500 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  2 | Khilan   |   2060 | 2008-05-20 00:00:00 |
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1560 | 2009-11-20 00:00:00 |
|  3 | kaushik  |   2060 | 2008-05-20 00:00:00 |
|  4 | Chaitali |   3000 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   1500 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   1560 | 2009-11-20 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|  5 | Hardik   |   3000 | 2009-10-08 00:00:00 |
|  5 | Hardik   |   1500 | 2009-10-08 00:00:00 |
|  5 | Hardik   |   1560 | 2009-11-20 00:00:00 |
|  5 | Hardik   |   2060 | 2008-05-20 00:00:00 |
|  6 | Komal    |   3000 | 2009-10-08 00:00:00 |
|  6 | Komal    |   1500 | 2009-10-08 00:00:00 |
|  6 | Komal    |   1560 | 2009-11-20 00:00:00 |
|  6 | Komal    |   2060 | 2008-05-20 00:00:00 |
|  7 | Muffy    |   3000 | 2009-10-08 00:00:00 |
|  7 | Muffy    |   1500 | 2009-10-08 00:00:00 |
|  7 | Muffy    |   1560 | 2009-11-20 00:00:00 |
|  7 | Muffy    |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

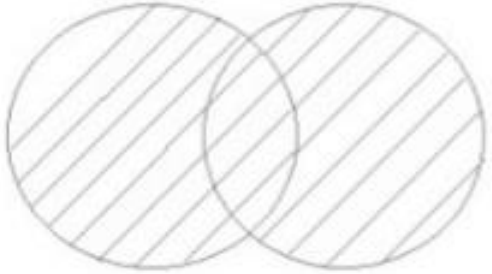
Set Operations

- SQL supports few Set operations to be performed on table data to get meaningful results from data under different specials conditions.
- Queries containing set operators are called compound queries.
- Set operations :

- Union
- Union All
- Intersect
- Minus

Union

- Union combines results of two or more selected statements.
- Eliminate duplicate rows from its result set.
- All distinct rows selected by either query.
- Numbers of columns and data type must be same in the both the tables.



Example of UNION

The **First** table,

ID	Name
1	abhi
2	adam

The **Second** table,

ID	Name
2	adam
3	Chester

Union SQL query will be,

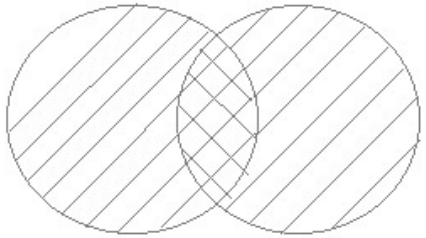
```
select * from First
UNION
select * from second
```

The result table will look like,

ID	NAME
1	abhi
2	adam
3	Chester

Union ALL

- All rows selected by either query, including all duplicates.
- Operation is similar to Union. But it also shows the duplicate rows.



Example of Union All

The **First** table,

ID	NAME
1	abhi
2	adam

The **Second** table,

ID	NAME
2	adam
3	Chester

Union All query will be like,

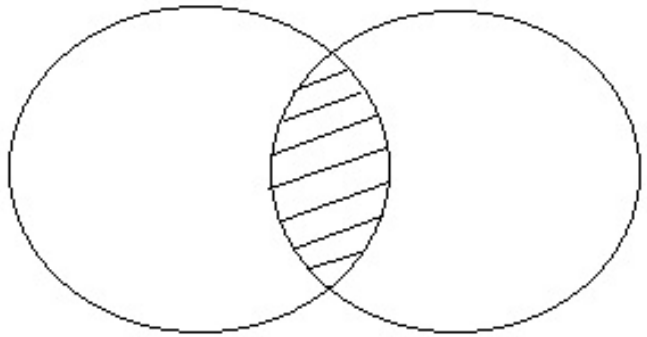
```
select * from First
UNION ALL
select * from second
```

The result table will look like,

ID	NAME
1	abhi
2	adam
2	adam
3	Chester

Intersect

- All distinct rows selected by both queries.
- Intersect operation is used to combine two SELECT statements, but it only returns the records which are common from both SELECT statements.
- In case of **Intersect** the number of columns and data type must be same.
- MySQL does not support INTERSECT operator.



Example of Intersect

The **First** table,

ID	NAME
1	abhi
2	adam

The **Second** table,

ID	NAME
2	adam
3	Chester

Intersect query will be,

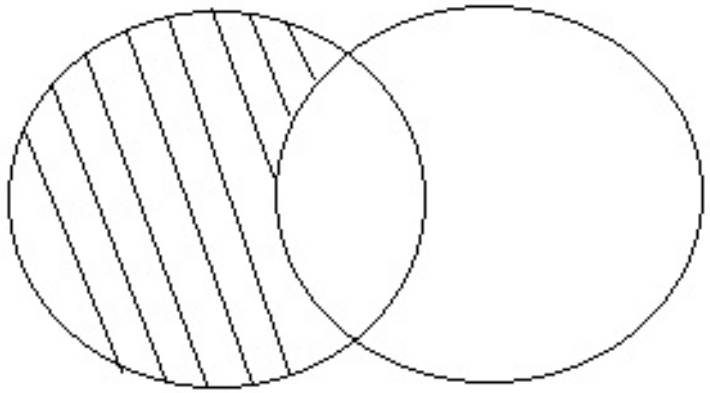
```
select * from First
INTERSECT
select * from second
```

The result table will look like

ID	NAME
2	adam

Minus

- All distinct rows selected by the first query but not the second.
- Minus operation combines result of two Select statements and return only those result which belongs to first set of result.



Example of Minus

The **First** table,

ID	NAME
1	abhi
2	adam

The **Second** table,

ID	NAME
2	adam
3	Chester

Minus query will be,

```
select * from First
MINUS
select * from second
```

The result table will look like,

ID	NAME
1	abhi

SQL Sequence

- Sequence is a feature supported by some database systems to produce unique values on demand.
- Some DBMS like **MySQL** supports AUTO_INCREMENT in place of Sequence.
- AUTO_INCREMENT is applied on columns, it automatically increments the column value by 1 each time a new record is entered into the table.
- Sequence is also some what similar to AUTO_INCREMENT but its has some extra feature

Creating Sequence

Syntax to create sequences is,

```
CREATE Sequence sequence-name
start with initial-value
increment by increment-value
maxvalue maximum-value
cycle|nocycle
```

initial-value specifies the starting value of the Sequence, **increment-value** is the value by which sequence will be incremented and **maxvalue** specifies the maximum value until which sequence will increment itself. **cycle** specifies that if the maximum value exceeds the set limit, sequence will restart its cycle from the begining. **No cycle** specifies that if sequence exceeds **maxvalue** an error will be thrown.

Example to create Sequence

The sequence query is following

```
CREATE Sequence seq_1
start with 1
increment by 1
maxvalue 999
cycle ;
```

Example to use Sequence

The **class** table,

ID	NAME
1	abhi
2	adam
4	alex

The sql query will be,

```
INSERT into class value(seq_1.nextval,'anu');
```

Result table will look like,

ID	NAME
1	abhi
2	adam
4	alex
1	anu

Once you use nextval the sequence will increment even if you don't Insert any record into the table.

SQL View

- A view in SQL is a logical subset of data from one or more tables.
- View is used to restrict data access.

Syntax for creating a View,

```
CREATE or REPLACE view view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition
```

Example of Creating a View

Consider following **Sale** table,

oid	order_name	previous_balance	customer
11	ord1	2000	Alex
12	ord2	1000	Adam
13	ord3	2000	Abhi
14	ord4	1000	Adam
15	ord5	2000	Alex

SQL Query to Create View

```
CREATE or REPLACE view sale_view as select * from Sale where customer = 'Alex';
```

The data fetched from select statement will be stored in another object called **sale_view**. We can use create seperately and replace too but using both together works better.

Example of Displaying a View

Syntax of displaying a view is similar to fetching data from table using Select statement.

```
SELECT * from sale_view;
```

Force View Creation

force keyword is used while creating a view. This keyword force to create View even if the table does not exist. After creating a force View if we create the base table and enter values in it, the view will be automatically updated.

Syntax for forced View is,

```
CREATE or REPLACE force view view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition
```

Update a View

Update command for view is same as for tables.

Syntax to Update a View is,

```
UPDATE view-name
set value
WHERE condition;
```

If we update a view it also updates base table data automatically.

Read-Only View

We can create a view with read-only option to restrict access to the view.

Syntax to create a view with Read-Only Access

```
CREATE or REPLACE force view view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition with read-only
```

The above syntax will create view for read-only purpose, we cannot Update or Insert data into read-only view. It will throw an error.

Types of View

There are two types of view,

- Simple View
- Complex View

Simple View	Complex View
Created from one table	Created from one or more table
Does not contain functions	Contain functions
Does not contain groups of data	Contains groups of data

SQl Aggregate Functions

- An aggregate function allows you to perform a calculation on a set of values to return a single scalar value.
- We often use aggregate functions with the GROUP BY and HAVING clauses of the SELECT statement.
- The following are the most commonly used SQL aggregate functions:
  - AVG – calculates the average of a set of values.
  - COUNT – counts rows in a specified table or view.
  - MIN – gets the minimum value in a set of values.
  - MAX – gets the maximum value in a set of values.
  - SUM – calculates the sum of values.

  Note that all aggregate functions above ignore NULL values except for the COUNT function.

SQL aggregate functions syntax

To call an aggregate function, you use the following syntax:

```
aggregate_function (DISTINCT | ALL expression)
```

Let's examine the syntax above in greater detail:

- First, specify an aggregate function that you want to use e.g., `MIN`, `MAX`, `AVG`, `SUM` or `COUNT`.
- Second, put `DISTINCT` or `ALL` modifier followed by an expression inside parentheses. If you explicitly use `DISTINCT` modifier, the aggregate function ignores duplicate values and only consider the unique values. If you use the `ALL` modifier, the aggregate function uses all values for calculation or evaluation. The `ALL` modifier is used by default if you do not specify any modifier explicitly.

SQL aggregate function examples

COUNT function example

To get the number of the products in the `products` table, you use the `COUNT` function as follows:

```
SELECT
    COUNT(*)
FROM
    products;
```

	COUNT(*)
▶	77

AVG function example

To calculate the average units in stock of the products, you use the `AVG` function as follows:

```
SELECT
    AVG(unitsinstock)
FROM
    products;
```

	AVG(unitsinstock)
▶	40.5065

To calculate units in stock by product category, you use the AVG function with the GROUP BY clause as follows:

```
SELECT
    categoryid, AVG(unitsinstock)
FROM
    products
GROUP BY categoryid;
```

categoryid	AVG(unitsinstock)
1	46.5833
2	42.2500
3	29.6923
4	39.3000
5	44.0000
6	27.5000
7	20.0000
8	58.4167

SUM function example

To calculate the sum of units in stock by product category, you use the `SUM` function with the `GROUP BY` clause as the following query:

```
SELECT
    categoryid, SUM(unitsinstock)
FROM
    products
GROUP BY categoryid;
```

	categoryid	sum(unitsinstock)
▶	1	559
	2	507
	3	386
	4	393
	5	308
	6	165
	7	100
	8	701

MIN function example

To get the minimum units in stock of products in the products table, you use the MIN function as follows:

```
SELECT
    MIN(unitsinstock)
FROM
    products;
```

	MIN(unitsinstock)
▶	0

More information on the MIN function.

MAX function example

To get the maximum units in stock of products in the products table, you use the MAX function as shown in the following query:

```
SELECT
    MAX(unitsinstock)
FROM
    products;
```

	Max(unitsinstock)
▶	125