# EXPERIMENT 2

**Aim**: Simulate query optimization by applying an SQL query on your selected database.

**DATABASE SCHEMA:**

| /\*==============================================================\*/ /\* Table: Customer \*/ /\*==============================================================\*/ create table Customer (  Id int **not** null,  FirstName varchar(40) **not** null,  LastName varchar(40) **not** null,  City varchar(40) null,  Country varchar(40) null,  Phone varchar(20) null,  constraint PK\_CUSTOMER primary key (Id) ) ;  /\*==============================================================\*/ /\* Index: IndexCustomerName \*/ /\*==============================================================\*/ create index IndexCustomerName on Customer ( LastName ASC, FirstName ASC ) ;  /\*==============================================================\*/ /\* Table: ShopOrder \*/ /\*==============================================================\*/ create table ShopOrder (  Id int **not** null,  OrderDate datetime **not** null,  OrderNumber varchar(10) null,  CustomerId int **not** null,  TotalAmount decimal(12,2) null default 0,  constraint PK\_ORDER primary key (Id) ) ;  /\*==============================================================\*/ /\* Index: IndexOrderCustomerId \*/ /\*==============================================================\*/ create index IndexOrderCustomerId on ShopOrder ( CustomerId ASC ) ;  /\*==============================================================\*/ /\* Index: IndexOrderOrderDate \*/ /\*==============================================================\*/ create index IndexOrderOrderDate on ShopOrder ( OrderDate ASC ) ;  /\*==============================================================\*/ /\* Table: OrderItem \*/ /\*==============================================================\*/ create table OrderItem (  Id int **not** null,  OrderId int **not** null,  ProductId int **not** null,  UnitPrice decimal(12,2) **not** null default 0,  Quantity int **not** null default 1,  constraint PK\_ORDERITEM primary key (Id) ) ;  /\*==============================================================\*/ /\* Index: IndexOrderItemOrderId \*/ /\*==============================================================\*/ create index IndexOrderItemOrderId on OrderItem ( OrderId ASC ) ;  /\*==============================================================\*/ /\* Index: IndexOrderItemProductId \*/ /\*==============================================================\*/ create index IndexOrderItemProductId on OrderItem ( ProductId ASC ) ;  /\*==============================================================\*/ /\* Table: Product \*/ /\*==============================================================\*/ create table Product (  Id int **not** null,  ProductName varchar(50) **not** null,  SupplierId int **not** null,  UnitPrice decimal(12,2) null default 0,  Package varchar(30) null,  IsDiscontinued bit **not** null default 0,  constraint PK\_PRODUCT primary key (Id) ) ;  /\*==============================================================\*/ /\* Index: IndexProductSupplierId \*/ /\*==============================================================\*/ create index IndexProductSupplierId on Product ( SupplierId ASC ) ;  /\*==============================================================\*/ /\* Index: IndexProductName \*/ /\*==============================================================\*/ create index IndexProductName on Product ( ProductName ASC ) ;  /\*==============================================================\*/ /\* Table: Supplier \*/ /\*==============================================================\*/ create table Supplier (  Id int **not** null,  CompanyName varchar(40) **not** null,  ContactName varchar(50) null,  ContactTitle varchar(40) null,  City varchar(40) null,  Country varchar(40) null,  Phone varchar(30) null,  Fax varchar(30) null,  constraint PK\_SUPPLIER primary key (Id) ) ;  /\*==============================================================\*/ /\* Index: IndexSupplierName \*/ /\*==============================================================\*/ create index IndexSupplierName on Supplier ( CompanyName ASC ) ;  /\*==============================================================\*/ /\* Index: IndexSupplierCountry \*/ /\*==============================================================\*/ create index IndexSupplierCountry on Supplier ( Country ASC ) ;  alter table ShopOrder  add constraint FK\_ORDER\_REFERENCE\_CUSTOMER foreign key (CustomerId)  references Customer (Id) ;  alter table OrderItem  add constraint FK\_ORDERITE\_REFERENCE\_ORDER foreign key (OrderId)  references ShopOrder (Id) ;  alter table OrderItem  add constraint FK\_ORDERITE\_REFERENCE\_PRODUCT foreign key (ProductId)  references Product (Id) ;  alter table Product  add constraint FK\_PRODUCT\_REFERENCE\_SUPPLIER foreign key (SupplierId)  references Supplier (Id) ; |
| --- |

QUERIES:

| SELECT \* FROM agents; |
| --- |

Purpose: This is used to give details of all agents of the company.

| A007|Ramasundar|Bangalore|0.15|077-25814763| A003|Alex |London|0.13|075-12458969| A008|Alford|New York|0.12|044-25874365| A011|Ravi Kumar|Bangalore|0.15|077-45625874| A010|Santakumar|Chennai|0.14|007-22388644| A012|Lucida|San Jose|0.12|044-52981425| A005|Anderson|Brisban|0.13|045-21447739| A001|Subbarao|Bangalore|0.14|077-12346674| A002|Mukesh|Mumbai|0.11|029-12358964| A006|McDen|London|0.15|078-22255588| A004|Ivan|Torento|0.15|008-22544166| A009|Benjamin|Hampshair|0.11|008-22536178| |
| --- |

| SELECT \* FROM customer; |
| --- |

Purpose: This is used to give details of all the customers of the company/

| C00013|Holmes|London|London|UK|2|6000|5000|7000|4000|BBBBBBB|A003 C00001|Micheal|New York|New York|USA|2|3000|5000|2000|6000|CCCCCCC|A008 C00020|Albert|New York|New York|USA|3|5000|7000|6000|6000|BBBBSBB|A008 C00025|Ravindran|Bangalore|Bangalore|India|2|5000|7000|4000|8000|AVAVAVA|A011 C00024|Cook|London|London|UK|2|4000|9000|7000|6000|FSDDSDF|A006 C00015|Stuart|London|London|UK|1|6000|8000|3000|11000|GFSGERS|A003 C00002|Bolt|New York|New York|USA|3|5000|7000|9000|3000|DDNRDRH|A008 C00018|Fleming|Brisban|Brisban|Australia|2|7000|7000|9000|5000|NHBGVFC|A005 C00021|Jacks|Brisban|Brisban|Australia|1|7000|7000|7000|7000|WERTGDF|A005 C00019|Yearannaidu|Chennai|Chennai|India|1|8000|7000|7000|8000|ZZZZBFV|A010 C00005|Sasikant|Mumbai|Mumbai|India|1|7000|11000|7000|11000|147-25896312|A002 C00007|Ramanathan|Chennai|Chennai|India|1|7000|11000|9000|9000|GHRDWSD|A010 C00022|Avinash|Mumbai|Mumbai|India|2|7000|11000|9000|9000|113-12345678|A002 C00004|Winston|Brisban|Brisban|Australia|1|5000|8000|7000|6000|AAAAAAA|A005 C00023|Karl|London|London|UK|0|4000|6000|7000|3000|AAAABAA|A006 C00006|Shilton|Torento|Torento|Canada|1|10000|7000|6000|11000|DDDDDDD|A004 C00010|Charles|Hampshair|Hampshair|UK|3|6000|4000|5000|5000|MMMMMMM|A009 C00017|Srinivas|Bangalore|Bangalore|India|2|8000|4000|3000|9000|AAAAAAB|A007 C00012|Steven|San Jose|San Jose|USA|1|5000|7000|9000|3000|KRFYGJK|A012 C00008|Karolina|Torento|Torento|Canada|1|7000|7000|9000|5000|HJKORED|A004 C00003|Martin|Torento|Torento|Canada|2|8000|7000|7000|8000|MJYURFD|A004 C00009|Ramesh|Mumbai|Mumbai|India|3|8000|7000|3000|12000|Phone No|A002 C00014|Rangarappa|Bangalore|Bangalore|India|2|8000|11000|7000|12000|AAAATGF|A001 C00016|Venkatpati|Bangalore|Bangalore|India|2|8000|11000|7000|12000|JRTVFDD|A007 C00011|Sundariya|Chennai|Chennai|India|3|7000|11000|7000|11000|PPHGRTS|A010 |
| --- |

| SELECT \* FROM orders; |
| --- |

Purpose: This is used to give details of all the orders of the company

| 200100|1000|600|2008-08-01|C00013|A003|SOD 200110|3000|500|2008-04-15|C00019|A010|SOD 200107|4500|900|2008-08-30|C00007|A010|SOD 200112|2000|400|2008-05-30|C00016|A007|SOD 200113|4000|600|-2008-06-10|C00022|A002|SOD 200102|2000|300|-2008-05-25|C00012|A012|SOD 200114|3500|2000|-2008-08-15|C00002|A008|SOD 200122|2500|400|2008-09-16|C00003|A004|SOD 200118|500|100|2008-07-20|C00023|A006|SOD 200119|4000|700|2008-09-16|C00007|A010|SOD 200121|1500|600|2008-09-23|C00008|A004|SOD 200130|2500|400|2008-07-30|C00025|A011|SOD 200134|4200|1800|2008-09-25|C00004|A005|SOD 200108|4000|600|2008-02-15|C00008|A004|SOD 200103|1500|700|2008-05-15|C00021|A005|SOD 200105|2500|500|2008-07-18|C00025|A011|SOD 200109|3500|800|2008-07-30|C00011|A010|SOD 200101|3000|1000|2008-07-15|C00001|A008|SOD 200111|1000|300|2008-07-10|C00020|A008|SOD 200104|1500|500|2008-03-13|C00006|A004|SOD 200106|2500|700|2008-04-20|C00005|A002|SOD 200125|2000|600|2008-10-10|C00018|A005|SOD 200117|800|200|2008-10-20|C00014|A001|SOD 200123|500|100|2008-09-16|C00022|A002|SOD 200120|500|100|2008-07-20|C00009|A002|SOD 200116|500|100|2008-07-13|C00010|A009|SOD 200124|500|100|2008-06-20|C00017|A007|SOD 200126|500|100|2008-06-24|C00022|A002|SOD 200129|2500|500|2008-07-20|C00024|A006|SOD 200127|2500|400|2008-07-20|C00015|A003|SOD 200128|3500|1500|2008-07-20|C00009|A002|SOD 200135|2000|800|2008-09-16|C00007|A010|SOD 200131|900|150|2008-08-26|C00012|A012|SOD 200133|1200|400|2008-06-29|C00009|A002|SOD |
| --- |

| SELECT \* FROM orders WHERE CUST\_CODE="C00022"; |
| --- |

Purpose: This is used to give details of all the orders belonging to a customer whose CUST\_CODE is C00022

| 200113|4000|600|2008-06-10|C00022|A002|SOD 200123|500|100|2008-09-16|C00022|A002|SOD 200126|500|100|2008-06-24|C00022|A002|SOD |
| --- |

| SELECT \* FROM orders WHERE AGENT\_CODE = 'A002'; |
| --- |

Purpose: This is used to give details of all orders of agent with AGENT\_CODE = A002

| 200113|4000|600|-2008-06-10|C00022|A002|SOD 200106|2500|700|2008-04-20|C00005|A002|SOD 200123|500|100|2008-09-16|C00022|A002|SOD 200120|500|100|2008-07-20|C00009|A002|SOD 200126|500|100|2008-06-24|C00022|A002|SOD 200128|3500|1500|2008-07-20|C00009|A002|SOD 200133|1200|400|2008-06-29|C00009|A002|SOD |
| --- |

| SELECT \* FROM orders WHERE ORD\_DATE='2008-07-13'; |
| --- |

Purpose: This is used to give details of all orders that took place on 13th July 2008

| 200113|4000|600|2008-06-10|C00022|A002|SOD 200106|2500|700|2008-04-20|C00005|A002|SOD 200123|500|100|2008-09-16|C00022|A002|SOD 200120|500|100|2008-07-20|C00009|A002|SOD 200126|500|100|2008-06-24|C00022|A002|SOD 200128|3500|1500|2008-07-20|C00009|A002|SOD 200133|1200|400|2008-06-29|C00009|A002|SOD |
| --- |

| SELECT \* FROM agents WHERE WORKING\_AREA='London'; |
| --- |

Purpose: This is used to show details of all agents working in London

| A003|Alex |London|0.13|075-12458969| A006|McDen|London|0.15|078-22255588| |
| --- |

| SELECT \* FROM agents order by COMMISSION ASC; |
| --- |

Purpose: This is used to show details of all agents arranged in ascending order of their commission. Can be used during audits or performance discussion meetings.

| A002|Mukesh|Mumbai|0.11|029-12358964| A009|Benjamin|Hampshair|0.11|008-22536178| A008|Alford|New York|0.12|044-25874365| A012|Lucida|San Jose|0.12|044-52981425| A003|Alex |London|0.13|075-12458969| A005|Anderson|Brisban|0.13|045-21447739| A010|Santakumar|Chennai|0.14|007-22388644| A001|Subbarao|Bangalore|0.14|077-12346674| A007|Ramasundar|Bangalore|0.15|077-25814763| A011|Ravi Kumar|Bangalore|0.15|077-45625874| A006|McDen|London|0.15|078-22255588| A004|Ivan|Torento|0.15|008-22544166| |
| --- |

| SELECT AVG(OUTSTANDING\_AMT) FROM customer WHERE OUTSTANDING\_AMT>5000; |
| --- |

Purpose: This is used to show the average outstanding amount of all customers filtered by a range greater than 5000

| 9000.0 |
| --- |

| SELECT CUST\_CODE, CUST\_NAME, CUST\_COUNTRY, MAX(OUTSTANDING\_AMT) FROM customer; |
| --- |

Purpose: This is used to show details of that customer who has the maximum outstanding amount.

| C00009|Ramesh|India|12000 |
| --- |

| SELECT \* FROM customer GROUP BY AGENT\_CODE ORDER BY COUNT(\*) DESC; |
| --- |

Purpose: This shows details of all customers grouped by AGENT\_CODE ordered in descending order.

| C00009|Ramesh|Mumbai|Mumbai|India|3|8000|7000|3000|12000|Phone No|A002 C00003|Martin|Torento|Torento|Canada|2|8000|7000|7000|8000|MJYURFD|A004 C00004|Winston|Brisban|Brisban|Australia|1|5000|8000|7000|6000|AAAAAAA|A005 C00002|Bolt|New York|New York|USA|3|5000|7000|9000|3000|DDNRDRH|A008 C00011|Sundariya|Chennai|Chennai|India|3|7000|11000|7000|11000|PPHGRTS|A010 C00015|Stuart|London|London|UK|1|6000|8000|3000|11000|GFSGERS|A003 C00023|Karl|London|London|UK|0|4000|6000|7000|3000|AAAABAA|A006 C00016|Venkatpati|Bangalore|Bangalore|India|2|8000|11000|7000|12000|JRTVFDD|A007 C00014|Rangarappa|Bangalore|Bangalore|India|2|8000|11000|7000|12000|AAAATGF|A001 C00010|Charles|Hampshair|Hampshair|UK|3|6000|4000|5000|5000|MMMMMMM|A009 C00025|Ravindran|Bangalore|Bangalore|India|2|5000|7000|4000|8000|AVAVAVA|A011 C00012|Steven|San Jose|San Jose|USA|1|5000|7000|9000|3000|KRFYGJK|A012 |
| --- |

| SELECT \* FROM customer where CUST\_CODE IN (SELECT CUST\_CODE FROM orders WHERE ORD\_AMOUNT>4000); |
| --- |

Purpose: This is a nested query. It shows details of all customers whose order amount is greater than 4000

| C00004|Winston|Brisban|Brisban|Australia|1|5000|8000|7000|6000|AAAAAAA|A005 C00007|Ramanathan|Chennai|Chennai|India|1|7000|11000|9000|9000|GHRDWSD|A010 |
| --- |

| SELECT customer.CUST\_NAME, orders.ORD\_AMOUNT, orders.ORD\_DATE from orders INNER JOIN customer on orders.CUST\_CODE=customer.CUST\_CODE; |
| --- |

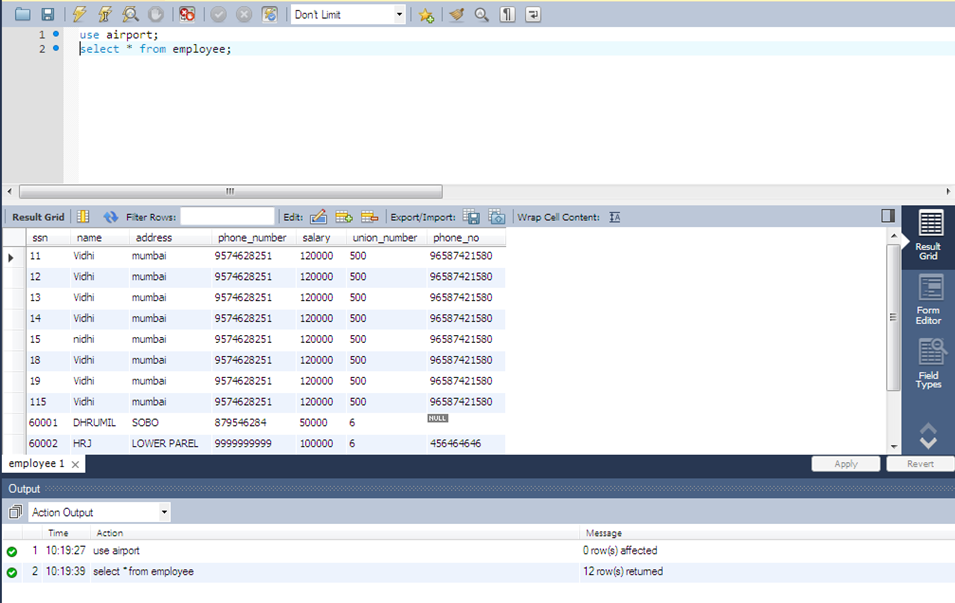
Purpose: This query does an inner join of the Customer table and the order table.

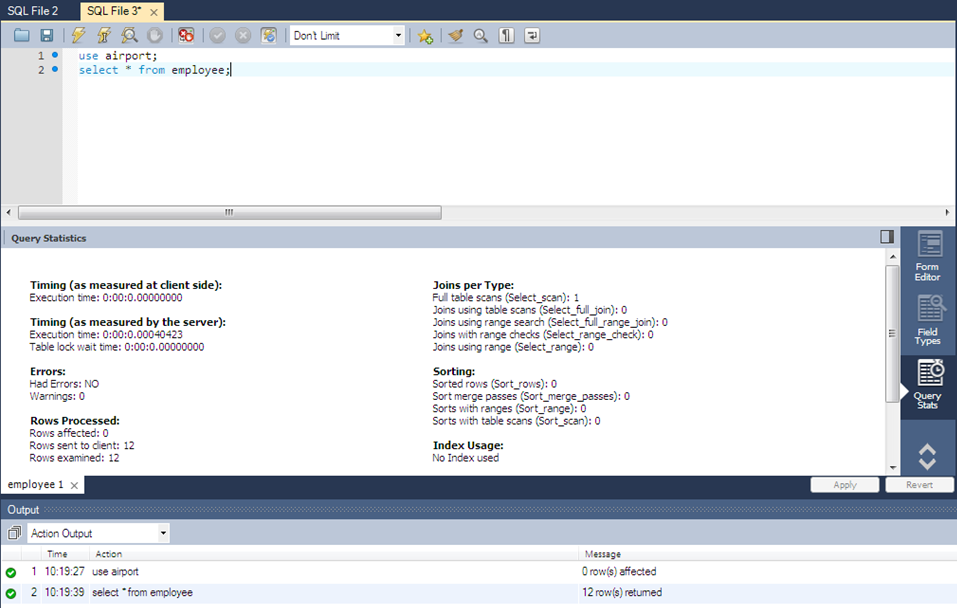
| Holmes|1000|2008-08-01 Yearannaidu|3000|2008-04-15 Ramanathan|4500|2008-08-30 Venkatpati|2000|2008-05-30 Avinash|4000|-2008-06-10 Steven|2000|-2008-05-25 Bolt|3500|-2008-08-15 Martin|2500|2008-09-16 Karl|500|2008-07-20 Ramanathan|4000|2008-09-16 Karolina|1500|2008-09-23 Ravindran|2500|2008-07-30 Winston|4200|2008-09-25 Karolina|4000|2008-02-15 Jacks|1500|2008-05-15 Ravindran|2500|2008-07-18 Sundariya|3500|2008-07-30 Micheal|3000|2008-07-15 Albert|1000|2008-07-10 Shilton|1500|2008-03-13 Sasikant|2500|2008-04-20 Fleming|2000|2008-10-10 Rangarappa|800|2008-10-20 Avinash|500|2008-09-16 Ramesh|500|2008-07-20 Charles|500|2008-07-13 Srinivas|500|2008-06-20 Avinash|500|2008-06-24 Cook|2500|2008-07-20 Stuart|2500|2008-07-20 Ramesh|3500|2008-07-20 Ramanathan|2000|2008-09-16 Steven|900|2008-08-26 Ramesh|1200|2008-06-29 |
| --- |

# OPTIMIZATION

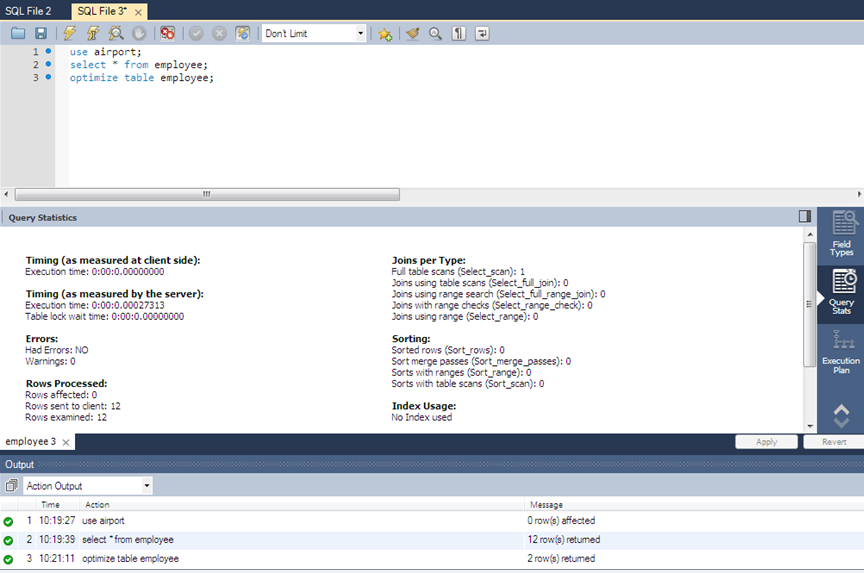
**TABLE LEVEL OPTIMIZATION**

**Normal Query : select \* from employee;**



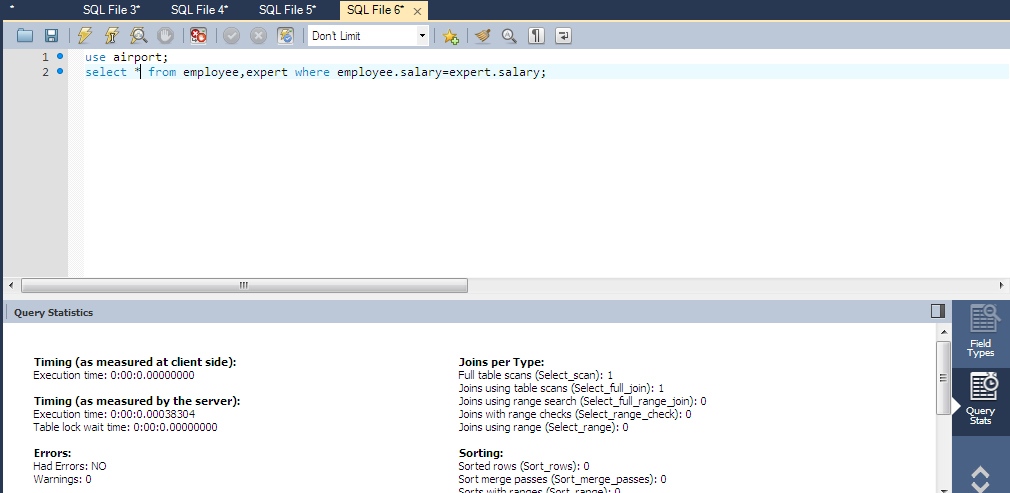
**Without Optimization**

**After Optimization**

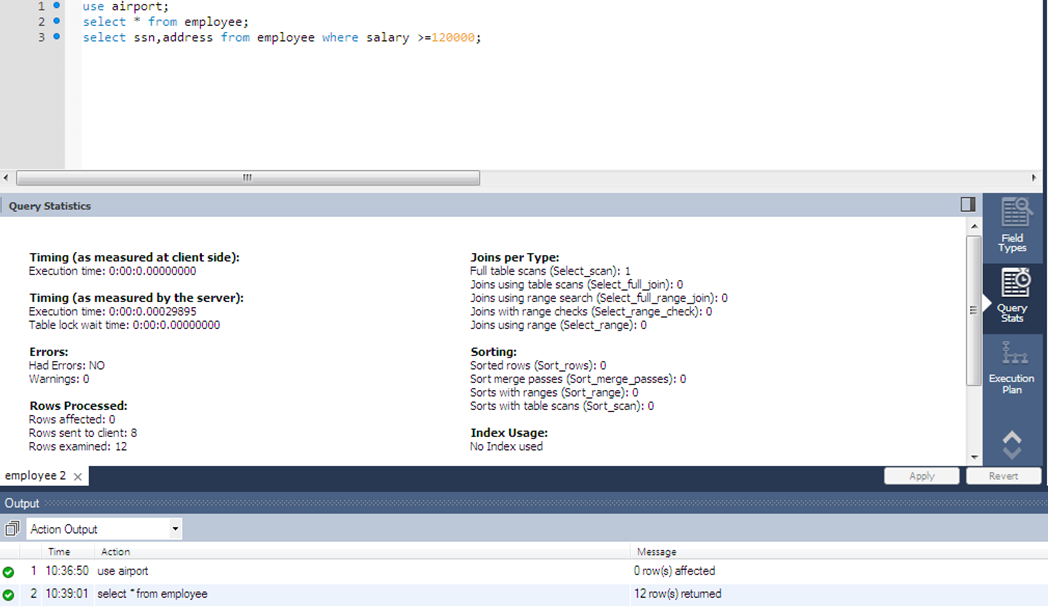


**INDEX LEVEL OPTIMIZATION**

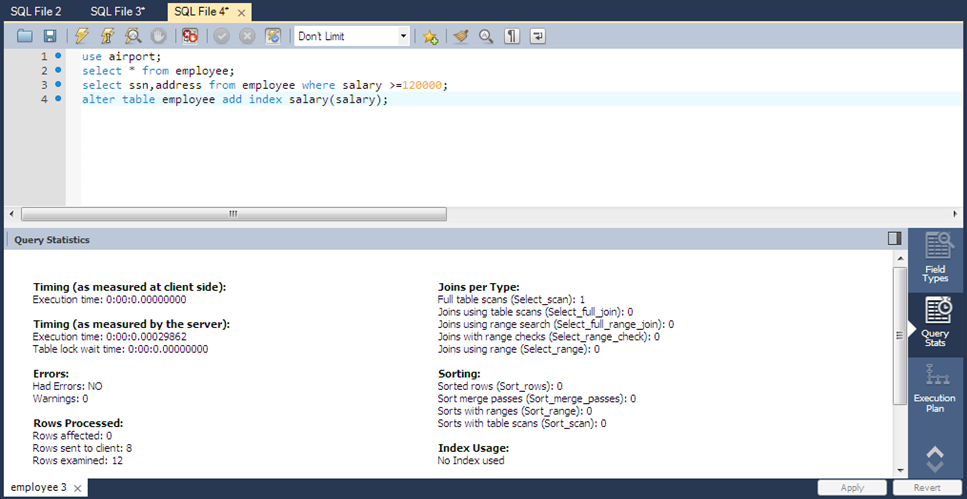
**Before adding index join query**

****

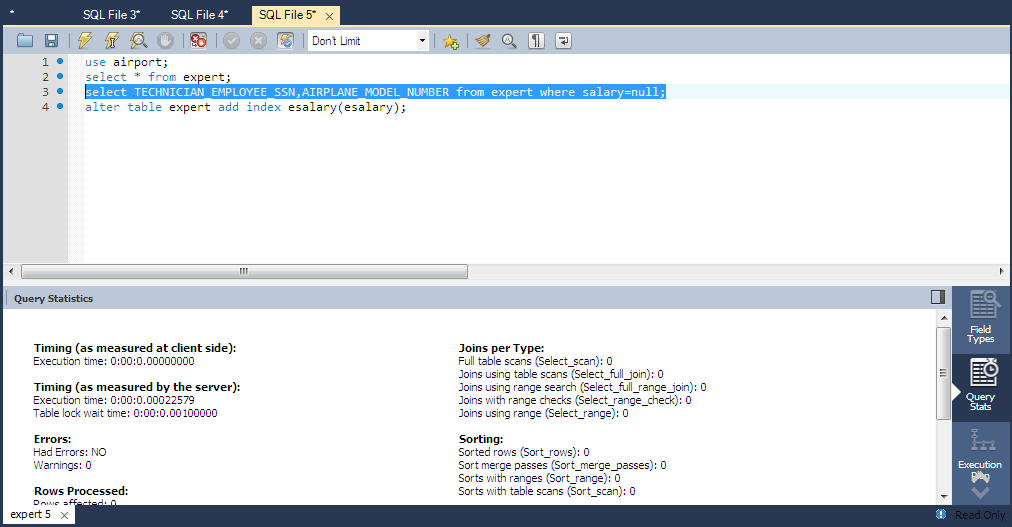
**Normal query on table employee**

****

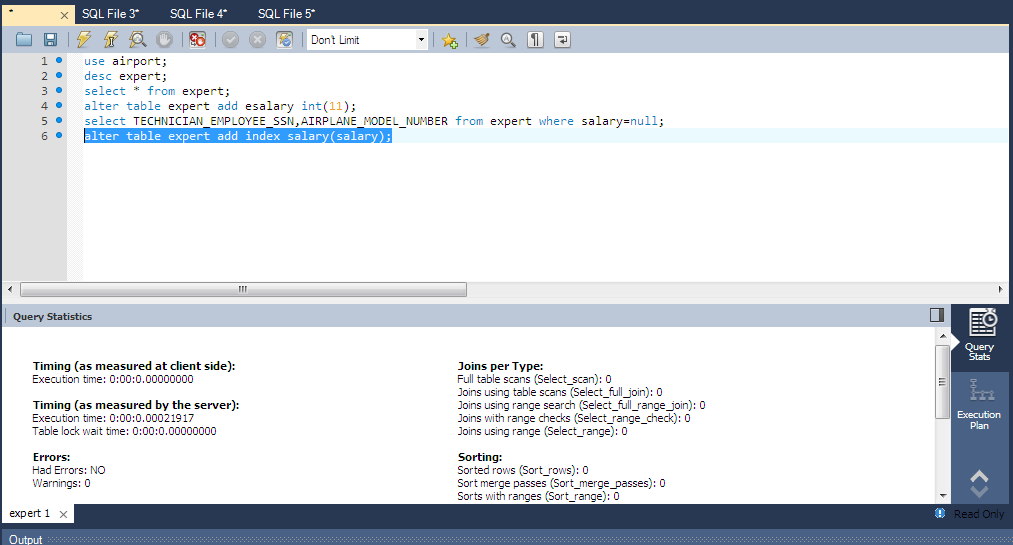
**After adding index**



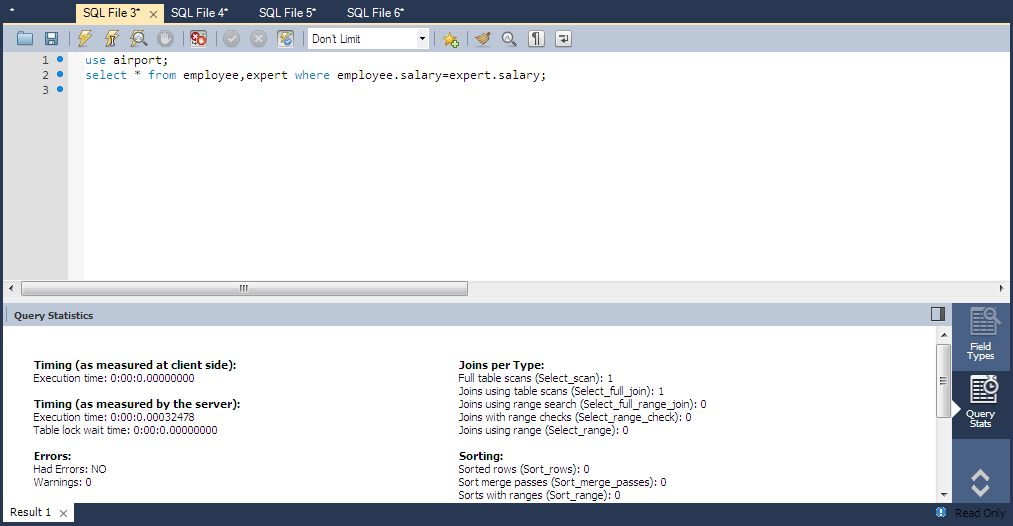
**Normal query on table expert**



**After adding index**



**After performing join**



# INDEXING

A database index is a data structure that improves the speed of operations in a table. Indexes can be created using one or more columns, providing the basis for both rapid random lookups and efficient ordering of access to records.

While creating an index, it should be taken into consideration which all columns will be used to make SQL queries and create one or more indexes on those columns.

Practically, indexes are also a type of tables, which keep the primary key or index field and a pointer to each record into the actual table.

The users cannot see the indexes, they are just used to speed up queries and will be used by the Database Search Engine to locate records very fast.

The INSERT and UPDATE statements take more time on tables having indexes, whereas the SELECT statements become fast on those tables. The reason is that while doing insert or update, a database needs to insert or update the index values as well.

By default, MySQL allows index type **BTREE** if we have not specified the type of index. The following table shows the different types of an index based on the storage engine of the table.

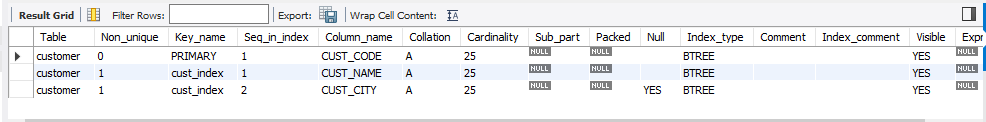
SYNTAX:

**CREATE** **INDEX** [index\_name] **ON** [table\_name] (**column** names)

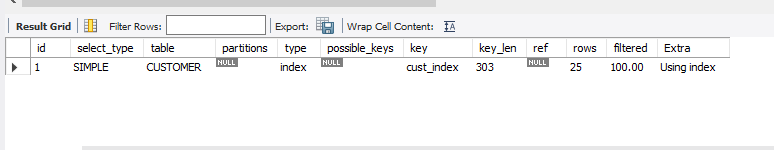
CODE:

| CREATE INDEX cust\_index ON CUSTOMER (CUST\_NAME, CUST\_CITY); |
| --- |

| SHOW INDEXES FROM CUSTOMER; |
| --- |



| EXPLAIN SELECT CUST\_NAME FROM CUSTOMER; |
| --- |



# **Conclusion**: We learnt about SQL queries and how to optimise it. We then learnt about the different optimization queries and indexing.

# 

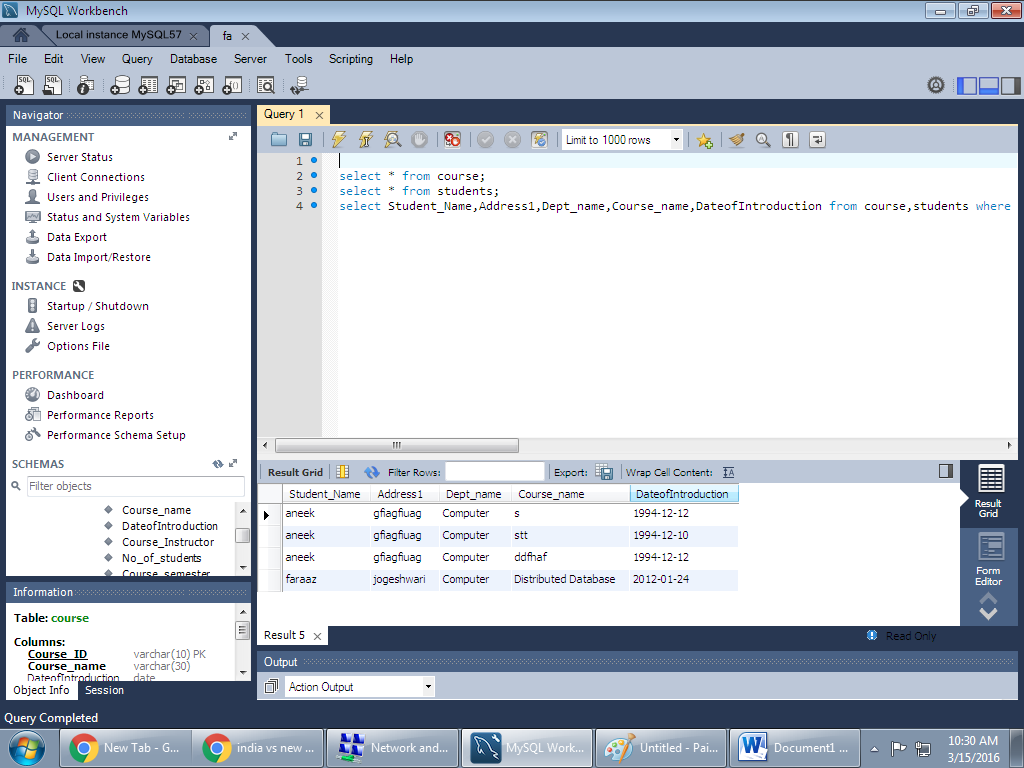
# EXPERIMENT 3

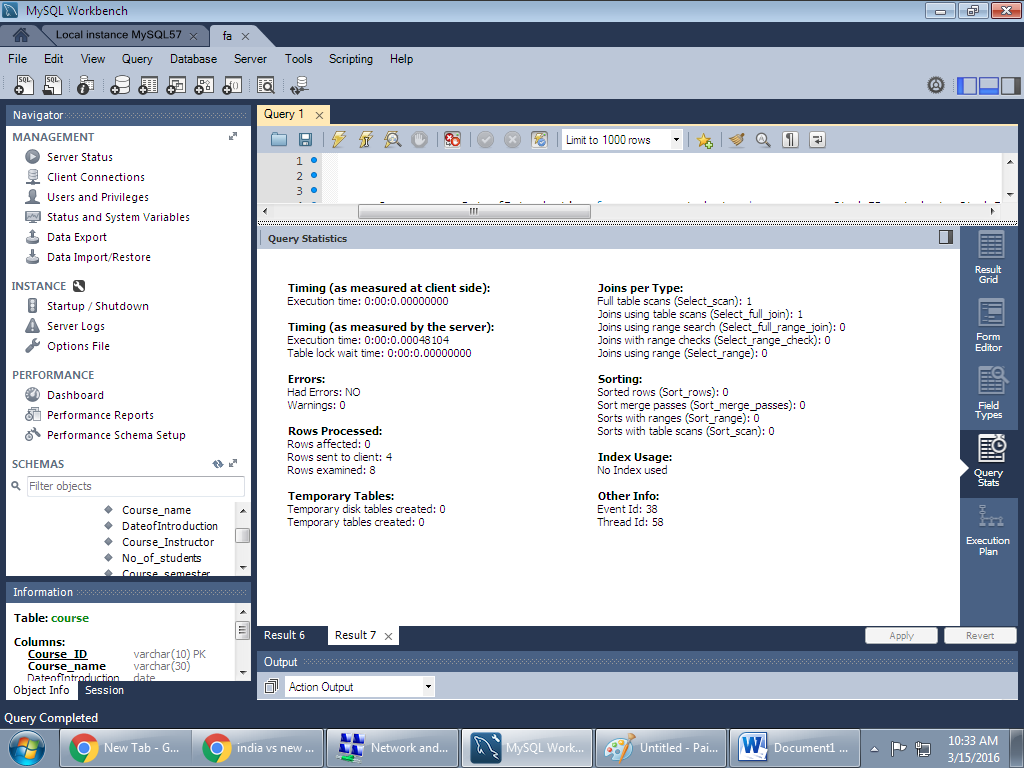
Aim: Implementation of query monitor.

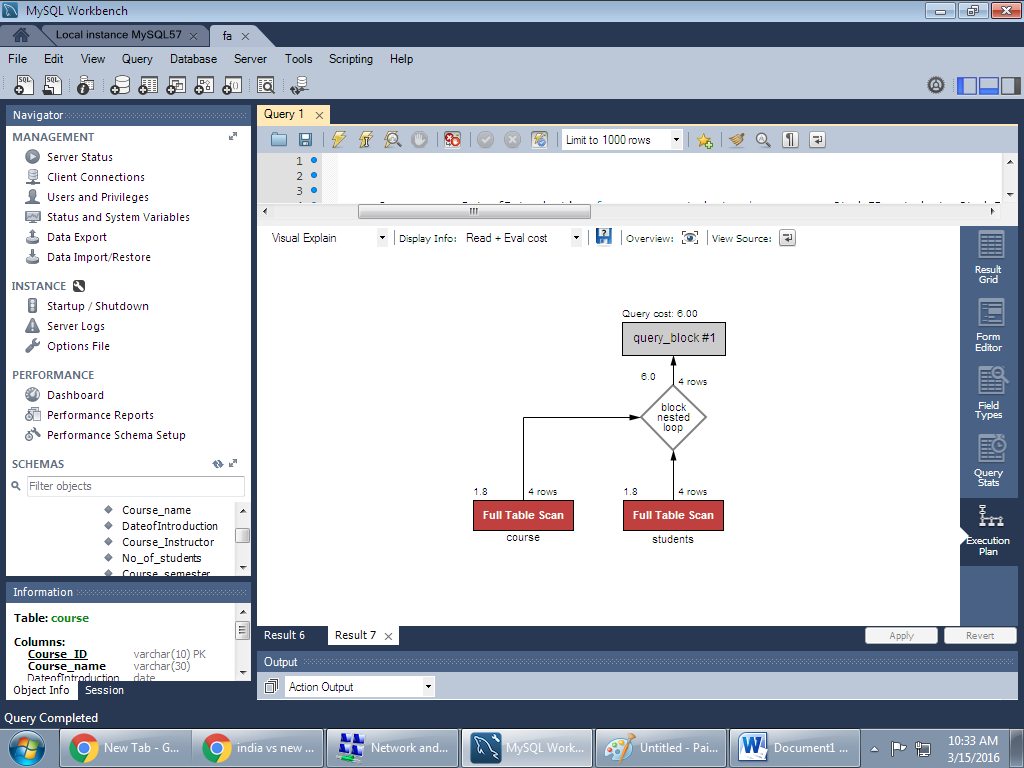
Labwork:

Join query on course and students tables

select Student\_Name,Address1,Dept\_name,Course\_name,DateofIntroduction from course,students where course.Stud\_ID= students.Stud\_ID;

1. 





**What is an execution plan?**

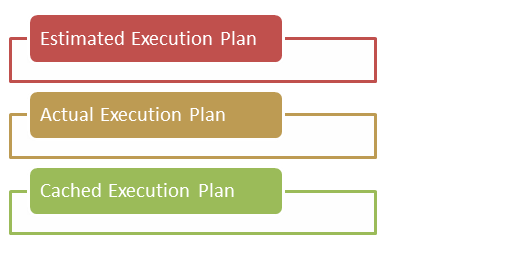
It is a graphical representation of the operation performed by the SQL server database engine. Execution plan is generated by query optimizer. It tells us the flow of the query.

Execution plan lets us know how a query will execute on the database engine to return some results.

**Types of execution plan**

Most people think that there are only two types of execution plans, estimated execution plan and actual execution plan, but that’s not true, there are three types of execution plans as mentioned below,

**SQL Query Execution Plan**

****

**Estimated execution plan**

This execution plan shows us SQL optimizer view. Basically it provides us an estimation of query execution process to get the query results. It is a compiled plan, which means query does not get executed to get execution plan.

To get a query’s estimated execution plan we have to do the following steps,

* Switch to SSMS (Sql Server Management Studio)
* highlight the query
* click on Query
* click Display Estimated Execution Plan

Conclusion: In this experiment we have learned what is an execution, how many types of execution plans sql has and the display options of the plans.

# 

# EXPERIMENT 4

# PARTITIONING

**THEORY**:

Partitioning in MySQL is used to split or partition the rows of a table into separate tables in different locations, but still, it is treated as a single table. It distributes the portions of the table's data across a file system based on the rules we have set as our requirement. The rule that we have set to accomplish the division of table data is called a partitioning function (modulus, a linear or internal hashing function, etc.). The selected function is based on the partitioning type we have specified and takes a user-supplied expression as its parameter. The user- expression can be a column value or a function acting on column values, depending on the type of partitioning used.

MySQL has mainly two forms of partitioning:

**1. Horizontal Partitioning**

This partitioning split the rows of a table into multiple tables based on our logic. In horizontal partitioning, the number of columns is the same in each table, but no need to keep the same number of rows. It physically divides the table but logically treated as a whole. Currently, MySQL supports this partitioning only.

**2. Vertical Partitioning**

This partitioning splits the table into multiple tables with fewer columns from the original table. It uses an additional table to store the remaining columns. Currently, MySQL does not provide supports for this partitioning.

**Benefits of Partitioning**

The following are the benefits of partitioning in MySQL:

* It optimizes the query performance. When we query on the table, it scans only the portion of a table that will satisfy the particular statement.
* It is possible to store extensive data in one table that can be held on a single disk or file system partition.
* It provides more control to manage the data in your database.

**How can we partition the table in MySQL?**

We can create a partition in MySQL using the CREATE TABLE or ALTER TABLE statement. Below is the syntax of creating partition using CREATE TABLE command:

| CREATE TABLE [IF NOT EXISTS] table\_name  (column\_definitions)  [table\_options]  [partition\_options] |
| --- |

The below is the syntax of creating partition using ALTER TABLE command:

| ALTER TABLE [IF EXISTS] tab\_name  (colm\_definitions)  [tab\_options]  [partition\_options] |
| --- |

# **Types of MySQL Partitioning**

# MySQL has mainly six types of partitioning, which are given below:

# RANGE Partitioning

# LIST Partitioning

# COLUMNS Partitioning

# HASH Partitioning

# KEY Partitioning

# Subpartitioning

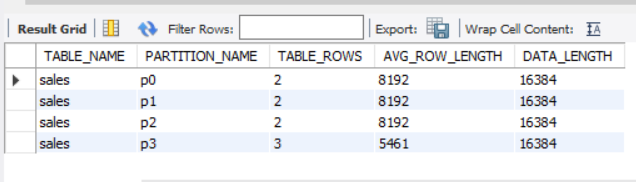
# **MySQL RANGE Partitioning**

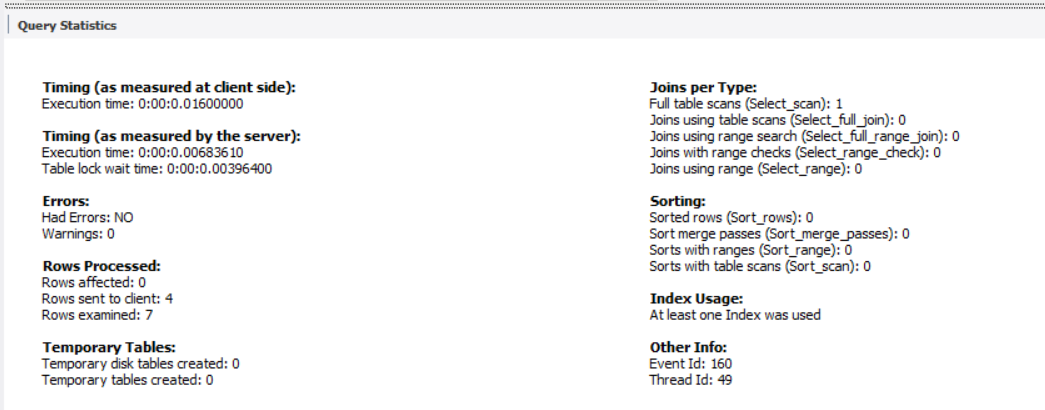
# This partitioning allows us to partition the rows of a table based on column values that fall within a specified range. The given range is always in a contiguous form but should not overlap each other, and also uses the VALUES LESS THAN operator to define the ranges.

CODE:

| CREATE TABLE Sales ( cust\_id INT NOT NULL, name VARCHAR(40),  store\_id VARCHAR(20) NOT NULL, bill\_no INT NOT NULL,  bill\_date DATE PRIMARY KEY NOT NULL, amount DECIMAL(8,2) NOT NULL)  PARTITION BY RANGE (year(bill\_date))(  PARTITION p0 VALUES LESS THAN (2016),  PARTITION p1 VALUES LESS THAN (2017),  PARTITION p2 VALUES LESS THAN (2018),  PARTITION p3 VALUES LESS THAN (2020));   INSERT INTO Sales VALUES  (1, 'Mike', 'S001', 101, '2015-01-02', 125.56),  (2, 'Robert', 'S003', 103, '2015-01-25', 476.50),  (3, 'Peter', 'S012', 122, '2016-02-15', 335.00),  (4, 'Joseph', 'S345', 121, '2016-03-26', 787.00),  (5, 'Harry', 'S234', 132, '2017-04-19', 678.00),  (6, 'Stephen', 'S743', 111, '2017-05-31', 864.00),  (7, 'Jacson', 'S234', 115, '2018-06-11', 762.00),  (8, 'Smith', 'S012', 125, '2019-07-24', 300.00),  (9, 'Adam', 'S456', 119, '2019-08-02', 492.20);   SELECT TABLE\_NAME, PARTITION\_NAME, TABLE\_ROWS, AVG\_ROW\_LENGTH, DATA\_LENGTH  FROM INFORMATION\_SCHEMA.PARTITIONS  WHERE TABLE\_SCHEMA = 'partitioning' AND TABLE\_NAME = 'Sales'; |
| --- |

OUTPUT:





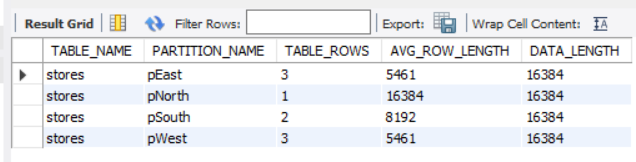
**MySQL LIST Partitioning**

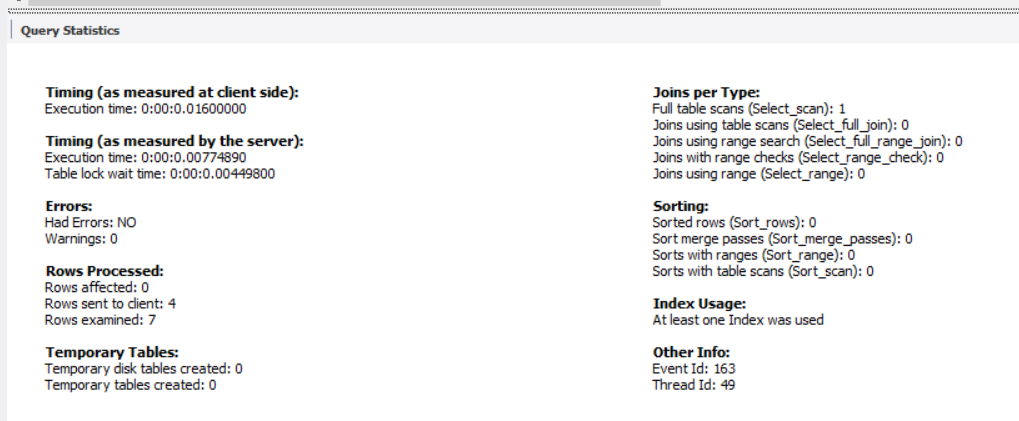
It is the same as Range Partitioning. Here, the partition is defined and selected based on columns matching one of a set of discrete value lists rather than a set of a contiguous range of values. It is performed by the PARTITION BY LIST(exp) clause. The exp is an expression or column value that returns an integer value. The VALUES IN(value\_lists) statement will be used to define each partition.

CODE:

| CREATE TABLE Stores (   cust\_name VARCHAR(40),   bill\_no VARCHAR(20) NOT NULL,   store\_id INT PRIMARY KEY NOT NULL,   bill\_date DATE NOT NULL,   amount DECIMAL(8,2) NOT NULL  )  PARTITION BY LIST(store\_id) (  PARTITION pEast VALUES IN (101, 103, 105),  PARTITION pWest VALUES IN (102, 104, 106),  PARTITION pNorth VALUES IN (107, 109, 111),  PARTITION pSouth VALUES IN (108, 110, 112));   INSERT INTO Stores VALUES ("Mike", "1", 101, "2015-01-25", 100.00), ("Joseph", "2", 102, "2015-01-25", 100.00), ("Robert", "3", 103, "2015-01-25", 100.00), ("Peter", "4", 104, "2015-01-25", 100.00), ("Joseph", "5", 105, "2015-01-25",100.00), ("Harry", "6", 106, "2015-01-25", 100.00), ("Jacson", "7", 107, "2015-01-25", 100.00), ("Smith", "8", 108, "2015-01-25", 100.00), ("Adam", "9", 110, "2015-01-25", 100.00);  SELECT TABLE\_NAME, PARTITION\_NAME, TABLE\_ROWS, AVG\_ROW\_LENGTH, DATA\_LENGTH  FROM INFORMATION\_SCHEMA.PARTITIONS  WHERE TABLE\_SCHEMA = 'partitioning' AND TABLE\_NAME = 'Stores'; |
| --- |

OUTPUT:





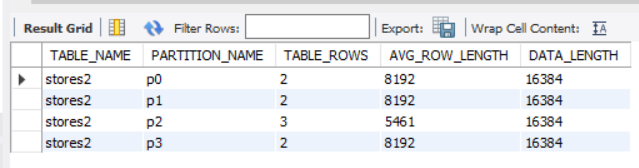
**MySQL HASH Partitioning**

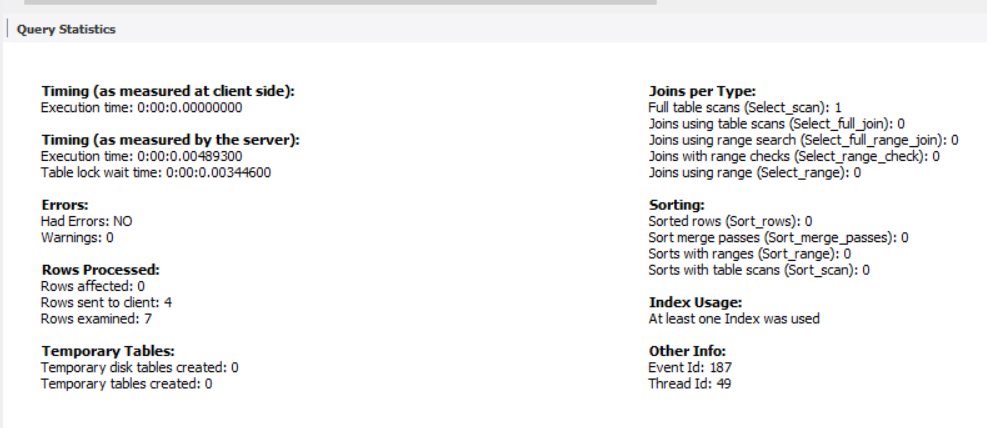
This partitioning is used to distribute data based on a predefined number of partitions. In other words, it splits the table as of the value returned by the user-defined expression. It is mainly used to distribute data evenly into the partition. It is performed with the PARTITION BY HASH(expr) clause. Here, we can specify a column value based on the column\_name to be hashed and the number of partitions into which the table is divided.

CODE:

| CREATE TABLE Stores2 (   cust\_name VARCHAR(40),   bill\_no VARCHAR(20) NOT NULL,   store\_id INT PRIMARY KEY NOT NULL,   bill\_date DATE NOT NULL,   amount DECIMAL(8,2) NOT NULL  )  PARTITION BY HASH(store\_id)  PARTITIONS 4;  INSERT INTO Stores2 VALUES ("Mike", "1", 101, "2015-01-25", 100.00), ("Joseph", "2", 102, "2015-01-25", 100.00), ("Robert", "3", 103, "2015-01-25", 100.00), ("Peter", "4", 104, "2015-01-25", 100.00), ("Joseph", "5", 105, "2015-01-25",100.00), ("Harry", "6", 106, "2015-01-25", 100.00), ("Jacson", "7", 107, "2015-01-25", 100.00), ("Smith", "8", 108, "2015-01-25", 100.00), ("Adam", "9", 110, "2015-01-25", 100.00);  SELECT TABLE\_NAME, PARTITION\_NAME, TABLE\_ROWS, AVG\_ROW\_LENGTH, DATA\_LENGTH  FROM INFORMATION\_SCHEMA.PARTITIONS  WHERE TABLE\_SCHEMA = 'partitioning' AND TABLE\_NAME = 'Stores2'; |
| --- |

OUTPUT:





**MySQL COLUMN Partitioning**

This partitioning allows us to use the multiple columns in partitioning keys. The purpose of these columns is to place the rows in partitions and determine which partition will be validated for matching rows. It is mainly divided into two types:

**RANGE Columns Partitioning**

LIST Columns Partitioning

They provide supports for the use of non-integer columns to define the ranges or value lists. They support the following data types:

All Integer Types: TINYINT, SMALLINT, MEDIUMINT, INT (INTEGER), and BIGINT.

String Types: CHAR, VARCHAR, BINARY, and VARBINARY.

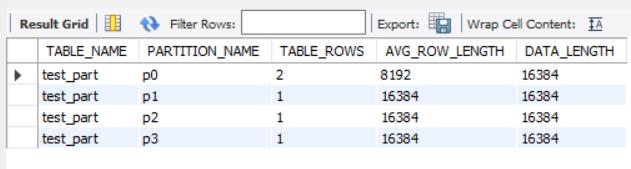
DATE and DATETIME data types.

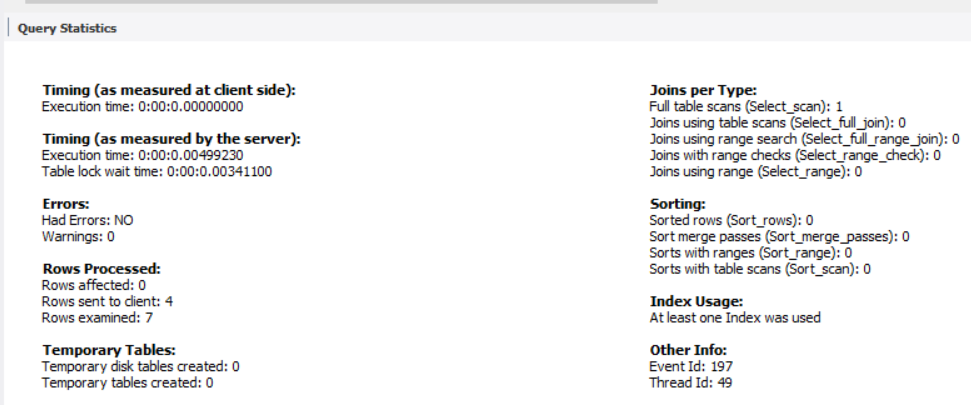
Range Column Partitioning: It is similar to the range partitioning with one difference. It defines partitions using ranges based on various columns as partition keys. The defined ranges are of column types other than an integer type.

CODE:

| CREATE TABLE test\_part (A INT, B CHAR(5), C INT, D INT)  PARTITION BY RANGE COLUMNS(A, B, C)   (PARTITION p0 VALUES LESS THAN (50, 'test1', 100),   PARTITION p1 VALUES LESS THAN (100, 'test2', 200),   PARTITION p2 VALUES LESS THAN (150, 'test3', 300),   PARTITION p3 VALUES LESS THAN (MAXVALUE, MAXVALUE, MAXVALUE));     INSERT INTO test\_part VALUES  (10, 'a', 50, 3),  (30, 'test1', 150, 3),  (55, 'test1', 175, 3),  (123, 'test2', 233, 3),  (160, 'test4', 333, 3);   SELECT TABLE\_NAME, PARTITION\_NAME, TABLE\_ROWS, AVG\_ROW\_LENGTH, DATA\_LENGTH  FROM INFORMATION\_SCHEMA.PARTITIONS  WHERE TABLE\_SCHEMA = 'partitioning' AND TABLE\_NAME = 'test\_part'; |
| --- |

OUTPUT:





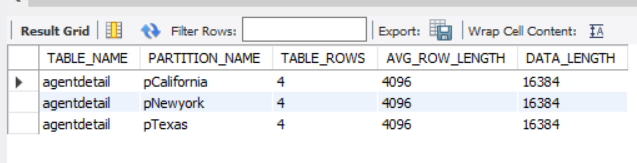
**LIST Columns Partitioning**

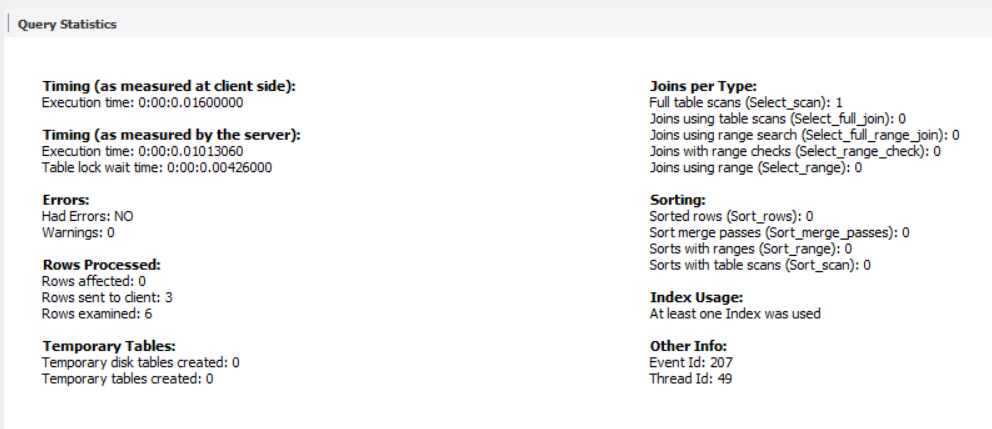
It takes a list of single or multiple columns as partition keys. It enables us to use various columns of types other than integer types as partitioning columns. In this partitioning, we can use String data types, DATE, and DATETIME columns.

CODE:

| CREATE TABLE AgentDetail (  agent\_id VARCHAR(10),  agent\_name VARCHAR(40),  city VARCHAR(10))  PARTITION BY LIST COLUMNS(agent\_id) (  PARTITION pNewyork VALUES IN('A1', 'A2', 'A3'),  PARTITION pTexas VALUES IN('B1', 'B2', 'B3'),  PARTITION pCalifornia VALUES IN ('C1', 'C2', 'C3'));   INSERT INTO AgentDetail VALUES ('A1', 'DummyName', 'CityName'), ('A2', 'DummyName', 'CityName'), ('A3', 'DummyName', 'CityName'), ('B1', 'DummyName', 'CityName'), ('B2', 'DummyName', 'CityName'), ('B3', 'DummyName', 'CityName'), ('C1', 'DummyName', 'CityName'), ('C2', 'DummyName', 'CityName'), ('C3', 'DummyName', 'CityName'), ('A1', 'DummyName', 'CityName'), ('C2', 'DummyName', 'CityName'), ('B1', 'DummyName', 'CityName');  SELECT TABLE\_NAME, PARTITION\_NAME, TABLE\_ROWS, AVG\_ROW\_LENGTH, DATA\_LENGTH  FROM INFORMATION\_SCHEMA.PARTITIONS  WHERE TABLE\_SCHEMA = 'partitioning' AND TABLE\_NAME = 'AgentDetail'; |
| --- |

OUTPUT:





**MySQL KEY Partitioning**

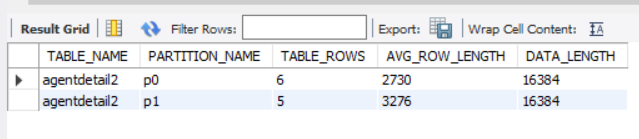
It is similar to the HASH partitioning where the hash partitioning uses the user-specified expression, and MySQL server supplied the hashing function for key. If we use other storage engines, the MySQL server employs its own internal hashing function that is performed by using the PARTITION BY KEY clause. Here, we will use KEY rather than HASH that can accept only a list of zero or more column names.

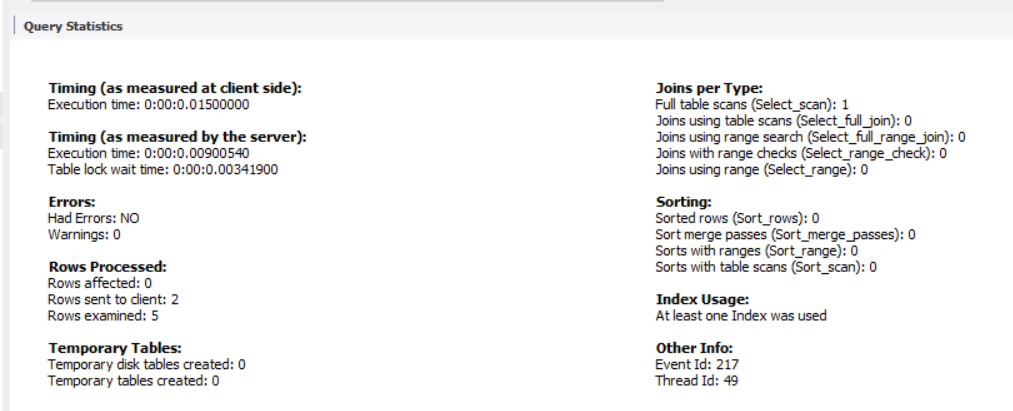
If the table contains a PRIMARY KEY and we have not specified any column for partition, then the primary key is used as partitioning key.

CODE:

| CREATE TABLE AgentDetail2 (   agent\_id INT NOT NULL PRIMARY KEY,   agent\_name VARCHAR(40)  )  PARTITION BY KEY()  PARTITIONS 2;   INSERT INTO AgentDetail2 VALUES (1, "Name"), (2, "Name"), (3, "Name"), (4, "Name"), (5, "Name"), (6, "Name"), (7, "Name"), (8, "Name"), (9, "Name"), (10, "Name"), (11, "Name");  SELECT TABLE\_NAME, PARTITION\_NAME, TABLE\_ROWS, AVG\_ROW\_LENGTH, DATA\_LENGTH  FROM INFORMATION\_SCHEMA.PARTITIONS  WHERE TABLE\_SCHEMA = 'partitioning' AND TABLE\_NAME = 'AgentDetail2'; |
| --- |

OUTPUT:





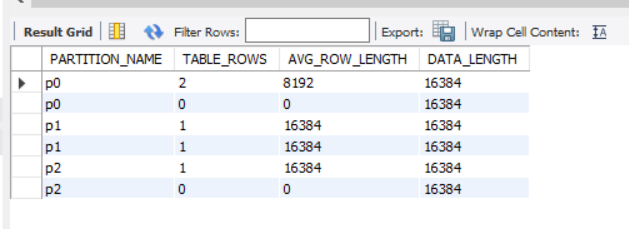
**SUBPARTITIONING**

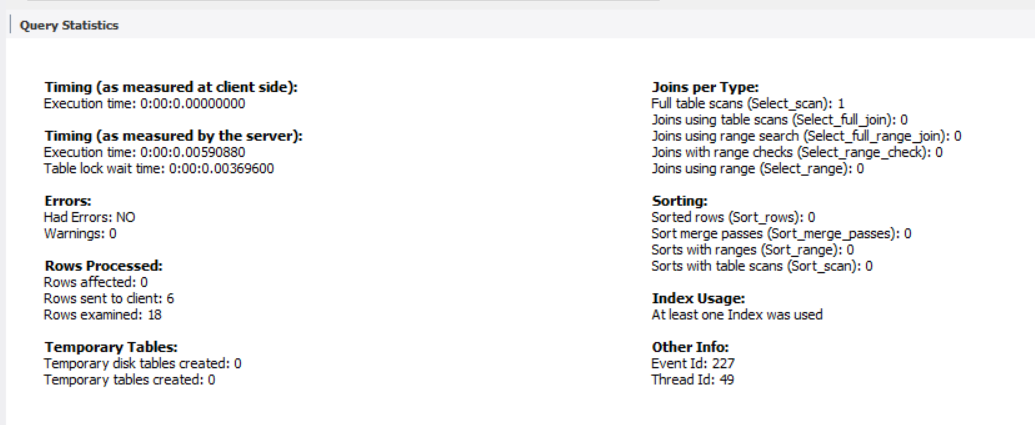
It is a composite partitioning that further splits each partition in a partition table.

CODE:

| CREATE TABLE Person (   id INT NOT NULL,   name VARCHAR(40),   purchased DATE,  PRIMARY KEY (`id`, `purchased`) )   PARTITION BY RANGE( YEAR(purchased) )   SUBPARTITION BY HASH( TO\_DAYS(purchased) )   SUBPARTITIONS 2 (   PARTITION p0 VALUES LESS THAN (2015),   PARTITION p1 VALUES LESS THAN (2020),   PARTITION p2 VALUES LESS THAN MAXVALUE   );    INSERT INTO Person VALUES (1, "Name", "2013-01-13"), (2, "Name2", "2014-04-22"), (3, "Name3", "2015-02-25"), (4, "Name4", "2018-05-05"), (5, "Name5", "2020-02-04");  SELECT PARTITION\_NAME, TABLE\_ROWS  FROM INFORMATION\_SCHEMA.PARTITIONS  WHERE TABLE\_SCHEMA = 'partitioning' AND TABLE\_NAME = 'Person'; |
| --- |

OUTPUT:





# 

# **Conclusion**: We learnt about partitioning and the different types of partitions available in MySQL. We then learnt about its uses and applications in real life where the main of goal of partitioning is to aid in maintenance of large tables and to reduce the overall response time to read and load data for particular SQL operations.

# 