

SUMMARY: VIEWS, INDEXES, AND DATA PARTITIONING

SESSION OVERVIEW:

By the end of this session students will be able to:

- Understand the concepts of Views and their use cases in MySQL.
- Understand the concepts of indexes.
- Understand the concepts of data partitioning along with list, range, and hash.

KEY TOPICS AND EXAMPLES:

Understanding the concepts of Views and their use cases in MySQL:

1. What is a view?

A view is a database object that has no values. Its contents are based on the base table. It contains rows and columns similar to the real table. In MySQL, the View is a virtual table created by a query that joins one or more tables. It is operated similarly to the base table but does not contain any data of its own. The View and table have one main difference the views are definitions built on top of other tables (or views). If any changes occur in the underlying table, the same changes are reflected in the View also.

2. Uses of views in MySQL:

- **Simplify complex query:** It allows the user to simplify complex queries. If we are using the complex query, we can create a view based on it to use a simple SELECT statement instead of typing the complex query again.
- **Increases the Re-usability:** We know that View simplifies complex queries and converts them into a single line of code to use VIEWS. Such a type of code makes it easier to integrate with our application. This will eliminate the chances of repeatedly writing the same formula in every query, making the code reusable and more readable.
- **Help in Data Security:** It also allows us to show only authorized information to the users and hide essential data like personal and banking information. We can limit which information users can access by authoring only the necessary data to them.
- **Enable Backward Compatibility:** A view can also enable backward compatibility in legacy systems. Suppose we want to split a large table into many smaller ones without affecting the current applications that reference the table. In this case, we will create a view with the same name as the real table so that the current applications can reference the view as if it were a table.

Basic Syntax:

```
CREATE [OR REPLACE] VIEW view_name AS
SELECT columns
FROM tables
[WHERE conditions];
```

NOTE:

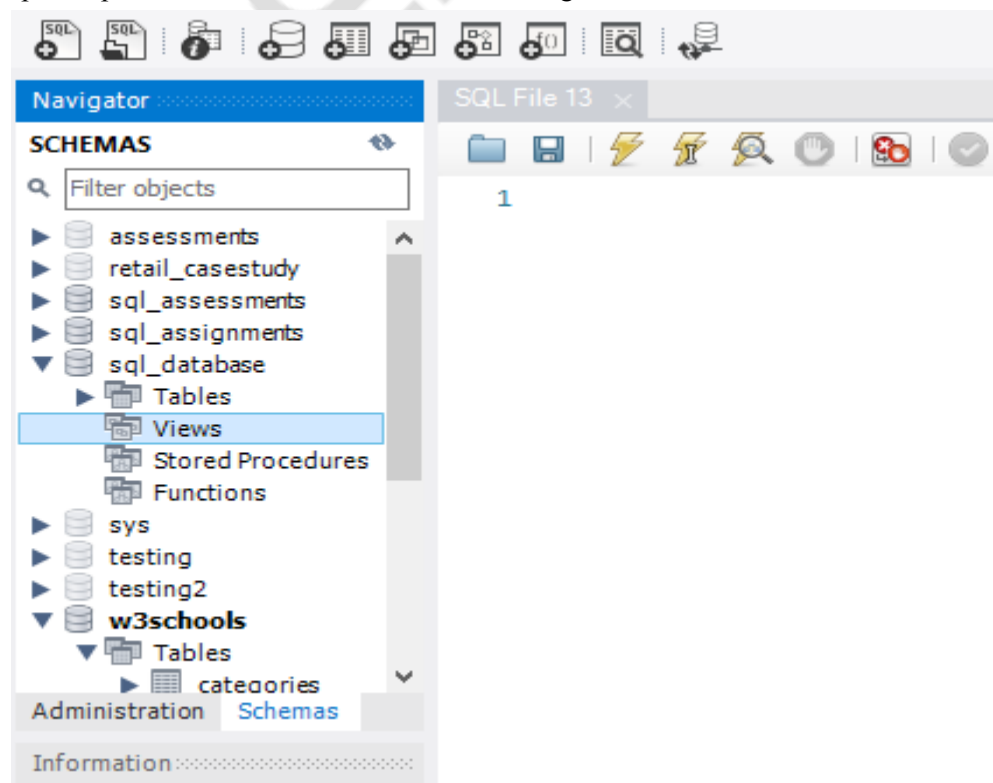
- **OR REPLACE:** It is optional. It is used when a VIEW already exists. If you do not specify this clause and the VIEW already exists, the CREATE VIEW statement will return an error.
- **view_name:** It specifies the name of the VIEW that you want to create in MySQL.
- **WHERE conditions:** It is also optional. It specifies the conditions that must be met for the records to be included in the VIEW.

3. How to create a view using MySQL Workbench:

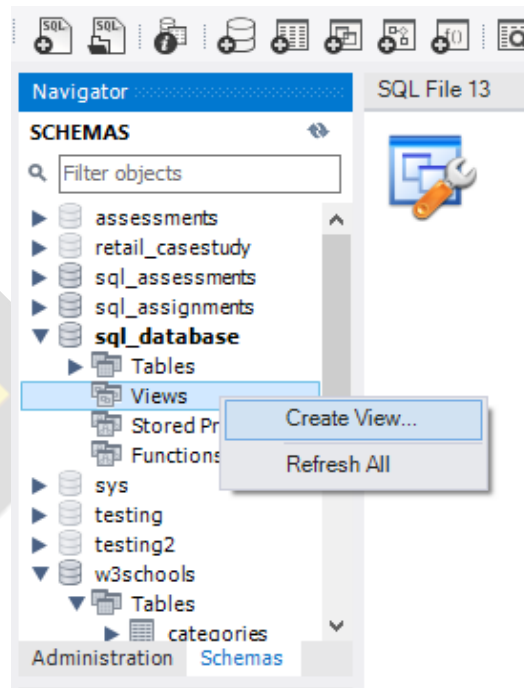
To create a view in the database using this tool, we first need to launch the MySQL Workbench and log in with the username and password to the MySQL server.

Steps:

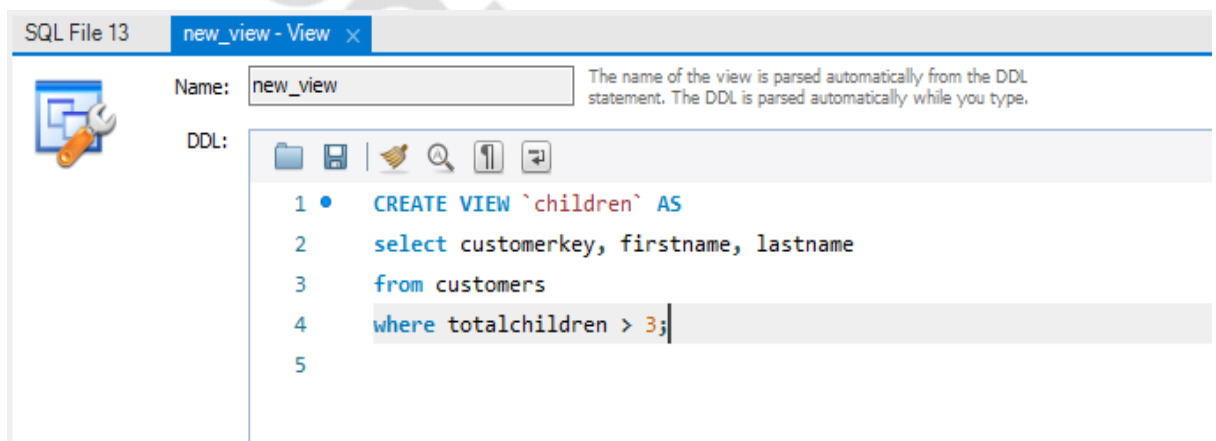
- Go to the Navigation tab and click on the Schema menu. Here, we can see all the previously created databases. Select any database under the Schema menu. It will pop up the option that can be shown in the following screen.



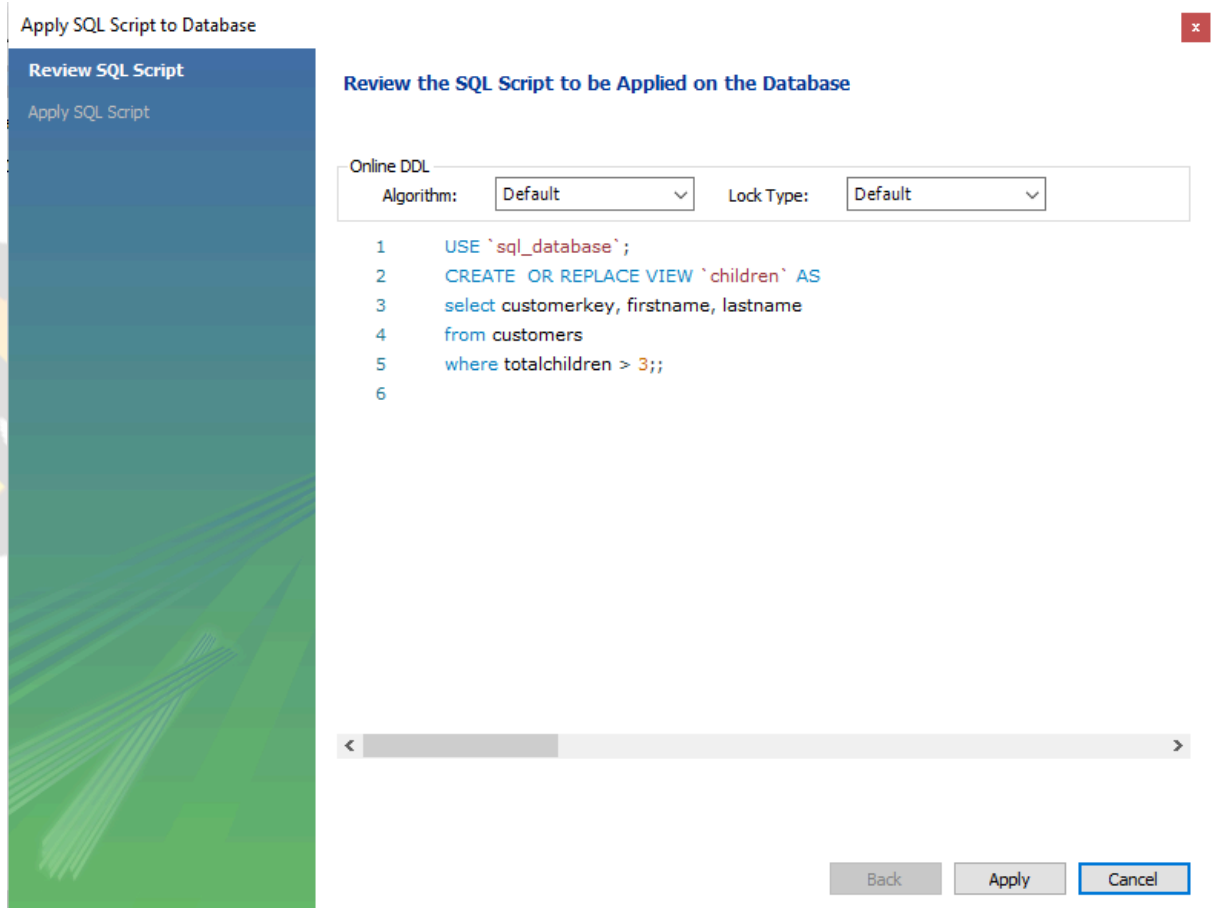
- Next, we need to right-click on the view option, and a new pop up screen will come:



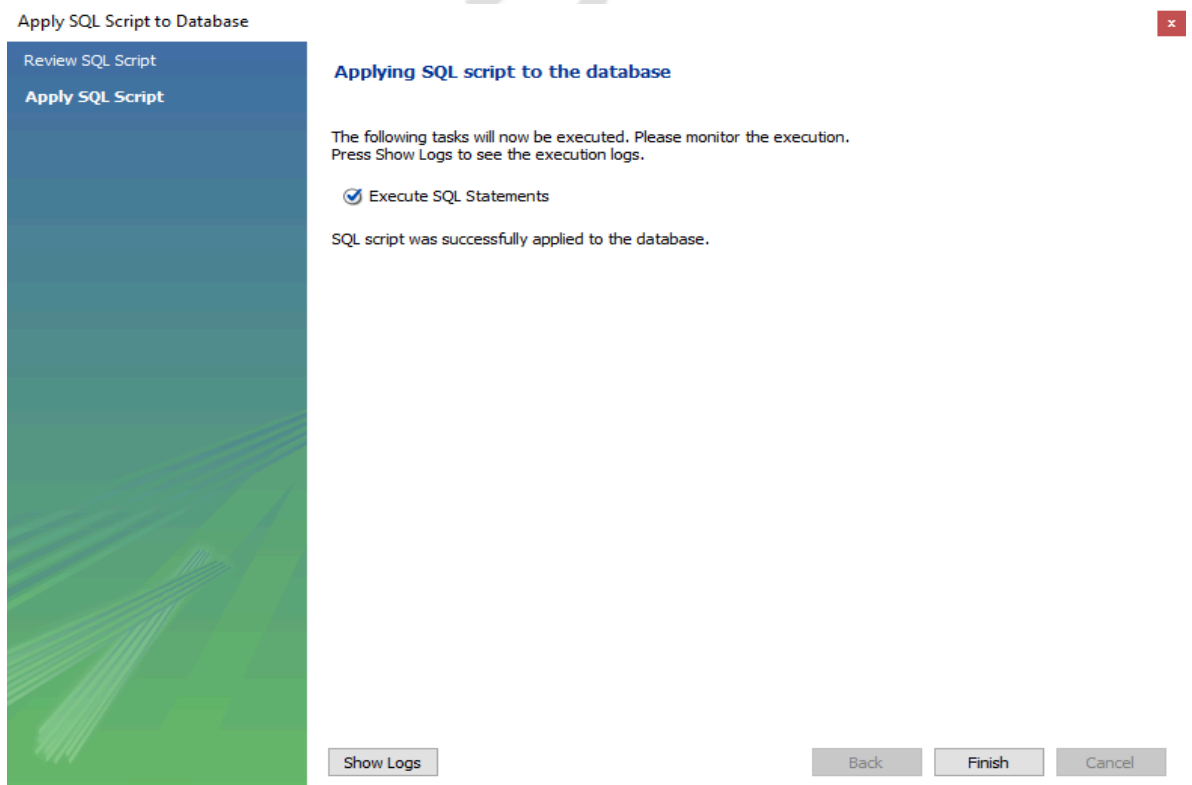
- c. As soon as we select the "Create View" option, it will give the below screen where we can write our own view.



- d. After completing the script's writing, click on the Apply button which is present below the screen and the following screen will appear:




e. In this screen, we will review the script and click the Apply button on the database



f. Finally, click on the Finish button to complete the view creation. Now, we can verify the view. On a new SQL tab for query creation write the following query:

```
select * from children;
```

Result Grid  Filter Rows: <input type="text"/>			
	customerkey	firstname	lastname
▶	11004	ELIZABETH	JOHNSON
	11008	ROBIN	VERHOFF
	11011	CURTIS	LU
	11017	SHANNON	WANG
	11031	THERESA	RAMOS
	11032	DENISE	STONE
	11033	JAIME	NATH
	11034	EBONY	GONZALEZ
	11102	JULIA	NELSON
	11113	MICHEAL	BLANCO
	11114	LESLIE	MORENO
	11115	ALVIN	CAI
	11116	CLINTON	CARLSON
	11117	APRIL	DENG

children 1 x

4. How to update a view in MySQL Workbench:

In MYSQL, the ALTER VIEW statement is used to modify or update the already created VIEW without dropping it.

Basic Syntax:

```
ALTER VIEW view_name AS
SELECT columns
FROM table
WHERE conditions;
```

Let's say I want to modify the above view that we have created. Now I want to add the total children column to the view.

```
ALTER VIEW children AS
SELECT customerkey, firstname, lastname, totalchildren
FROM customers
where totalchildren > 3;
```

Result Grid Filter Rows: Export:				
	customerkey	firstname	lastname	totalchildren
▶	11004	ELIZABETH	JOHNSON	5
	11008	ROBIN	VERHOFF	4
	11011	CURTIS	LU	4
	11017	SHANNON	WANG	4
	11031	THERESA	RAMOS	4
	11032	DENISE	STONE	4
	11033	JAIME	NATH	4
	11034	EBONY	GONZALEZ	4
	11102	JULIA	NELSON	5

children 3 x

The view has been updated and we can observe that a new column has been added in the view.

5. Methods to drop a view in MySQL Workbench:

We can drop a view in two ways in MySQL Workbench. Here is how we can drop a view.

a. Using SQL Command:

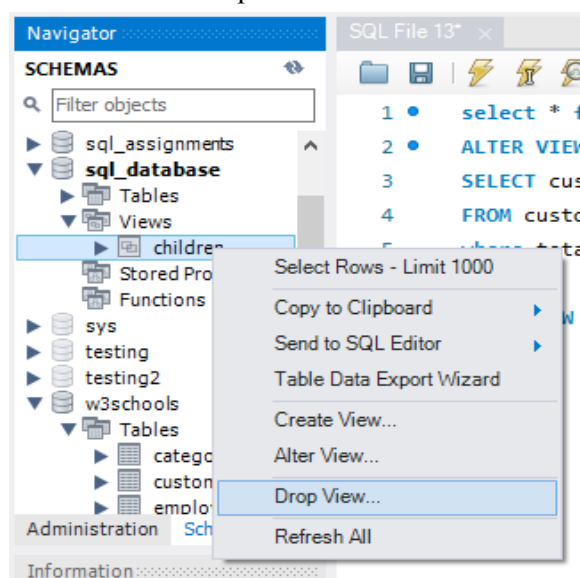
- Open MySQL Workbench and connect to your database.
- Open a new SQL query tab by clicking the "SQL" icon or pressing Ctrl+T.
- Type the DROP VIEW command and execute it:

```
DROP VIEW IF EXISTS children;
```

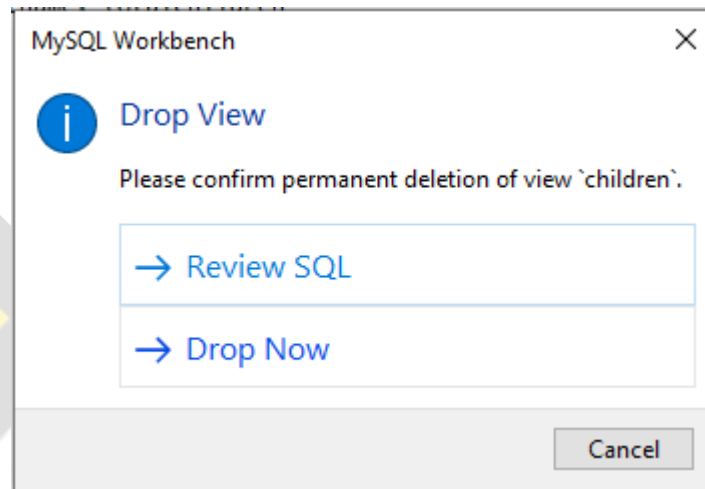
This command will drop the view if it exists.

b. Using the Graphical Interface

- Open MySQL Workbench and connect to your database.
- In the left navigation panel, expand the database schema containing the view you want to drop.
- Find the "Views" section and expand it to see the list of views in the schema.
- Right-click on the view you want to drop.
- Select "Drop View..." from the context menu.



- Confirm the action in the dialog that appears. This will execute the DROP VIEW command and remove the view from the database.



6. Comparing CTEs vs Views:

CTEs:

- **Temporary:** Defined within the execution scope of a single query. After the query runs, the CTE is no longer available.
- **Use-case Specific:** Typically used to simplify complex queries by breaking them down into more manageable parts within a particular query.

Views:

- **Permanent:** Stored in the database schema and persists beyond the execution of a query. It can be queried repeatedly like a regular table.
- **Reusable:** Serves as a virtual table that encapsulates a complex SQL query. Useful when the same result set is needed across multiple queries or applications.

CTEs:

- **Non-materialized:** Not stored on disk; the SQL engine may recompute the CTE each time it's referenced in the query. This can be a drawback if the CTE is complex and used multiple times within the same query.
- **Execution Plan Integration:** This can be optimized as part of the overall query execution plan, which can sometimes lead to more efficient querying than with views, especially if the CTE logic is closely integrated with the rest of the query.

Views:

- **Stored Query:** Though not stored as data, the query definition of a view is stored, meaning the data it returns is computed on-demand unless it's a materialized view.
- **Potential Performance Overhead:** Every time a view is queried, the database must run the underlying SQL query, which can be less efficient than querying a regular table unless indexed views are used.

Disadvantages of views:

Performance Degradation: Views can lead to performance issues, particularly when they involve complex queries with multiple joins across large datasets. These views can become resource-intensive, causing slower query execution times.

Increased Complexity: Over-reliance on views can lead to increased complexity in the database schema. This can make it more difficult to understand the underlying data model, especially if views are nested within other views.

Understanding the concepts of indexes:

1. What is indexing in SQL:

An index is a data structure that allows us to add indexes to the existing table. It enables you to improve the faster retrieval of records on a database table. It creates an entry for each value of the indexed columns. We use it to quickly find the record without searching each row in a database table whenever the table is accessed. We can create an index by using one or more columns of the table for efficient access to the records. An index creates a data structure that improves the speed of data retrieval operations at the cost of additional writes and storage space to maintain the index data structure.

2. Benefits of Indexes:

- **Faster data retrieval:** Indexes allow the database engine to locate specific rows quickly, especially for large tables.
- **Improved query performance:** Queries that involve indexed columns can be processed more efficiently.
- **Supports constraints:** Indexes can enforce uniqueness and primary key constraints.

3. Analogy:

Consider this with respect to reading a book and not having the index page. You open a random page (or in case of binary search you open the middle page of the book) and turn the pages left and right accordingly to go to the page that you want. This certainly takes time.

But if the book had contained the index page, this would certainly make the search even more efficient. You could have gone to the page just by looking at the index.

Maintain a block pointer for each block along with the ordered values used for the previous organization.

This will result in a smaller number of blocks being used to store the data files. Now, to search for a particular record, you will just have to search through this new organization consisting of a lesser number of blocks and eventually, you will get your record (if at all it is present) in much less time. Let's visualize this thing.

4. Basic Syntax:


```
CREATE TABLE t_index(  
  col1 INT PRIMARY KEY,  
  col2 INT NOT NULL,  
  col3 INT NOT NULL,  
  col4 VARCHAR(20),  
  INDEX (col2,col3)  
);
```

If we want to add an index to the table, we will use the **CREATE INDEX** statement as follows:

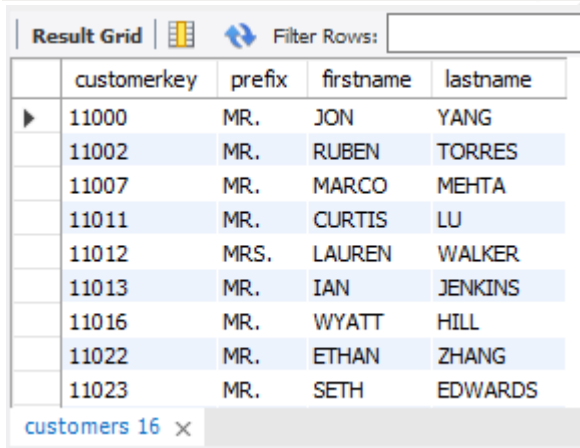
```
CREATE INDEX [index_name] ON [table_name] (column names)
```

In this statement, `index_name` is the name of the index, `table_name` is the name of the table to which the index belongs, and `column_names` is the list of columns.

Let's understand this using a simple example of the dataset that we have been using throughout the module. In this example, we will create an index on the customers table that already exists.

Now, execute the following statement to return the result of the customers whose marital status is married:

```
SELECT customerkey, prefix, firstname, lastname  
FROM customers  
WHERE maritalstatus = 'M';
```



Result Grid | Filter Rows:

	customerkey	prefix	firstname	lastname
▶	11000	MR.	JON	YANG
	11002	MR.	RUBEN	TORRES
	11007	MR.	MARCO	MEHTA
	11011	MR.	CURTIS	LU
	11012	MRS.	LAUREN	WALKER
	11013	MR.	IAN	JENKINS
	11016	MR.	WYATT	HILL
	11022	MR.	ETHAN	ZHANG
	11023	MR.	SETH	EDWARDS

customers 16 x

If you want to see how MySQL performs this query internally, execute the following statement:

```
EXPLAIN SELECT customerkey, prefix, firstname, lastname  
FROM customers  
WHERE maritalstatus = 'M';
```

You will get the output below. Here, MySQL scans the whole table to find the customers whose maritalstatus is M which represents married.

Result Grid

Filter Rows:

Export:

Wrap Cell Content:








	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
	1	SIMPLE	customers	NULL	ALL	NULL	NULL	NULL	NULL	2018	10.00	Using where

Now, let us create an index for a customerkey column using the following statement.

```
CREATE INDEX customerkey ON customers (customerkey);
```

After executing the above statement, the index is created successfully. Now, run the below statement to see how MySQL internally performs this query.

```
show index from customers;
```

Result Grid														
Filter Rows:		Export:		Wrap Cell Content: 										
	Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible
▶	customers	0	uq_firstname_lastname	1	FirstName	A	545	255		YES	BTREE			YES
	customers	0	uq_firstname_lastname	2	LastName	A	2018	255		YES	BTREE			YES
	customers	0	FirstName	1	FirstName	A	545	255		YES	BTREE			YES
	customers	0	FirstName	2	LastName	A	2018	255		YES	BTREE			YES
	customers	1	customerkey	1	customerkey	A	2018				BTREE			YES

A Case Study to understand these concepts more clearly:

A case study of what happens with & without index.

```
SELECT first_name, last_name, email
FROM customers
WHERE city = 'New York';
```

What Happens Without an Index:

- The database scans every row in customers.
- For each row, it checks if the city column matches "New York".
- This results in slow performance, especially with a large number of records

Adding an Index:

```
CREATE INDEX idx_city ON customers (city);
```

SQL Query (same as before) - no change in SQL query.

What Happens With an Index:

- The database uses the index idx_city to quickly locate all rows where the city is "New York".
- Instead of scanning the entire table, it scans the smaller, more efficient index to find matching rows.
- The database then retrieves the corresponding rows based on the index.

Without an Index:

- **Full table scan:** Checks all 100,000 rows to find matches. Time-consuming, especially with complex queries or additional joins.

With an Index:

- **Index scan:** Quickly locates entries in the city index for "New York". Significantly faster because it avoids scanning rows that do not match. Greatly reduces the number of rows the database needs to check.

Additional Benefits of Indexes:

- **Improved Query Performance:** As demonstrated, indexes drastically reduce the amount of data scanned.
- **Efficiency in Sorting and Grouping:** Indexes can also improve the performance of ORDER BY and GROUP BY clauses.
- **Support for Unique Constraints:** Indexes are used to enforce unique constraints and primary keys, ensuring data integrity.

Potential Drawbacks:

- **Increased Storage:** Indexes require additional disk space.
- **Performance Overhead on Writes:** Inserting, updating, or deleting rows in an indexed table takes longer because the index also needs to be updated.

The above case study can help solidify understanding and also shows how query doesn't change.

Concepts on drop index in SQL:

MySQL allows a DROP INDEX statement to remove the existing index from the table. To delete an index from a table, we can use the following query:

```
DROP INDEX index_name ON table_name
```

From the above example where we have created an index, now we will drop the index using the DROP keyword.

```
DROP INDEX customerkey ON customers;
```

Concepts of primary indexing:

Primary indexing in SQL refers to the indexing mechanism associated with the primary key of a table. A primary key is a unique identifier for a table's records, and primary indexing ensures that this key is used to quickly locate records within the table.

Concepts on Unique Index:

MySQL allows another constraint called the UNIQUE INDEX to enforce the uniqueness of values in one or more columns. We can create more than one UNIQUE index in a single table, which is not

possible with the primary key constraint.

Basic syntax:

```
CREATE UNIQUE INDEX index_name
ON table_name (index_column1, index_column2,...);
```

MySQL allows another approach to enforcing the uniqueness value in one or more columns using the UNIQUE Key statement.

```
CREATE TABLE table_name(
    col1 col_definition,
    col2 col_definition,
    ...
    [CONSTRAINT constraint_name]
    UNIQUE Key (column_name(s))
);
```

Let us understand it with the help of an example. Suppose we want to manage the employee details in a database application where we need email columns unique. Execute the following statement that creates a table "Employee_Detail" with a UNIQUE constraint:

```
CREATE TABLE Employee_Detail(
    ID int AUTO_INCREMENT PRIMARY KEY,
    Name varchar(45),
    Email varchar(45),
    Phone varchar(15),
    City varchar(25),
    UNIQUE KEY unique_email (Email)
);
```

If we execute the below statement, we can see that MySQL created a UNIQUE index for the Email column of the Employee_Detail table:

```
SHOW INDEXES FROM Employee_Detail;
```

Next, we are going to insert records into the table using the following statements:

```
INSERT INTO Employee_Detail(ID, Name, Email, Phone, City)
VALUES (1, 'Peter', 'peter@javatpoint.com', '49562959223', 'Texas'),
(2, 'Suzi', 'suzi@javatpoint.com', '70679834522', 'California'),
(3, 'Joseph', 'joseph@javatpoint.com', '09896765374', 'Alaska');
```

Suppose we want the Name and Phone of the Employee_Detail table is also unique. In this case, we will use the below statement to create a UNIQUE index for those columns:

```
CREATE UNIQUE INDEX index_name_phone  
ON Employee_Detail (Name, Phone);
```

Now the indexes will be created.

At a high level, using indexes might be counterproductive in the following scenarios:

- **Tables with Frequent Write Operations:** Indexes need to be updated whenever a row is inserted, updated, or deleted. For tables with frequent write operations, this maintenance overhead can significantly slow down the performance of these operations.
- **Small Tables:** For small tables, the overhead of maintaining indexes may outweigh the benefits. Full table scans can be faster than using an index, especially if the table fits entirely in memory.

Concepts on the clustered index:

A clustered index is actually a table where the data for the rows are stored. It defines the order of the table data based on the key values that can be sorted in only one way. In the database, each table can have only one clustered index. In a relational database, if the table column contains a primary key or unique key, MySQL allows you to create a clustered index named PRIMARY based on that specific column.

1. Advantages:

- It helps us to maximize the cache hits and minimizes the page transfer.
- It is an ideal option for a range or group with max, min, and count queries.
- At the start of the range, it uses a location mechanism for finding an index entry.

2. Disadvantages:

- **Maintenance Overhead:** Every time a row is inserted or deleted, the index needs to be updated, and the table may need to be reorganized to maintain order.
- **Single Clustered Index Limit:** Because data can only be sorted in one way physically, each table can only have one clustered index.

3. Clustered Index on InnoDB Tables:

MySQL InnoDB table must have a clustered index. The InnoDB table uses a clustered index for optimizing the speed of most common lookups and DML (Data Manipulation Language) operations like INSERT, UPDATE, and DELETE commands.

When the primary key is defined in an InnoDB table, MySQL always uses it as a clustered index named PRIMARY. If the table does not contain a primary key column, MySQL searches for the unique key. In the unique key, all columns are NOT NULL, and use it as a clustered index. Sometimes, the table does not have a primary key or unique key, then MySQL internally creates a hidden clustered index GEN_CLUST_INDEX that contains the values of row id. Thus, there is only one clustered index in the InnoDB table.

4. Basic Syntax:

```
CREATE TABLE `student_info` (
  `studentid` int NOT NULL AUTO_INCREMENT,
  `name` varchar(45) DEFAULT NULL,
  `age` varchar(3) DEFAULT NULL,
  `mobile` varchar(20) DEFAULT NULL,
  `email` varchar(25) DEFAULT NULL,
  PRIMARY KEY (`studentid`), //clustered index
  UNIQUE KEY `email_UNIQUE` (`email`)
)
```

Example:

```
CREATE CLUSTERED INDEX idx_studentid ON student_info (studentid);
```

Concepts on non-clustered indexes:

The indexes other than PRIMARY indexes (clustered indexes) are called a non-clustered index. The non-clustered indexes are also known as secondary indexes. The non-clustered index and table data are both stored in different places. It is not able to sort (order) the table data. The non-clustered indexing is the same as a book where the content is written in one place, and the index is at a different place. MySQL allows a table to store one or more than one non-clustered index. The non-clustered indexing improves the performance of the queries which uses keys without assigning primary keys.

Basic syntax:

```
CREATE NonClustered INDEX index_name ON table_name (column_name ASC);
```

Clustered vs Non-Clustered Indexes:

Clustered Index	Non-Clustered Index
A clustered index is a table where the data for the rows are stored. In a relational database, if the table column contains a primary key, MySQL automatically creates a clustered index named PRIMARY.	The indexes other than PRIMARY indexes (clustered indexes) are called non-clustered indexes. The non-clustered indexes are also known as secondary indexes.
It can be used to sort the record and store the index in physical memory.	It creates a logical ordering of data rows and uses pointers for accessing the physical data files.
Its size is large.	Its size is small in comparison to a clustered index.
It accesses the data very fast.	It has slower accessing power in comparison to the clustered index.
It stores records in the leaf node of an index.	It does not store records in the leaf node of an index which means it takes extra space for data.

It does not require additional reports.	It requires an additional space to store the index separately.
It uses the primary key as a clustered index.	It can work with unique constraints that act as a composite key.
A clustered index always contains an index id of 0.	A non-clustered index always contains an index id>0

Understand the concepts of data partitioning (list, range, and hash):

1. What is Data Partitioning?

Data partitioning in MySQL Workbench involves dividing a large table into smaller, more manageable pieces while maintaining the overall structure and schema of the original table. This technique is particularly useful for improving performance, simplifying maintenance, and managing large datasets efficiently.

2. Uses of Data Partitioning:

a. Performance Optimization:

- Enhances query performance by scanning only relevant partitions.
- Optimizes access to large datasets by segregating current and historical data.

b. Maintenance and Management:

- Facilitates efficient data archiving and purging without affecting the entire table.
- Simplifies backup and restore processes by handling individual partitions.

c. Improved Availability:

- It allows for the repair of individual partitions, minimizing downtime.
- Enhances system availability through focused maintenance tasks.

d. Scalability:

- Distributes data across multiple servers or storage devices, improving scalability.
- Supports sharding strategies to distribute load and manage data more effectively.

e. Load Balancing:

- Distributes the workload evenly across the database system, preventing hotspots.
- Ensures balanced load distribution for improved system performance.

Overview of date partitioning:

Before diving into the details of this topic let's understand it in brief.

Types of Partitioning:

1. **Range Partitioning:** This method involves splitting the table into partitions according to ranges of values in a specified column. Commonly used for date fields or any numerical range.

Example: Partitioning sales data into yearly segments, where each partition contains data for one year.

```
CREATE TABLE sales (  
  sale_id INT AUTO_INCREMENT PRIMARY KEY,  
  product_id INT NOT NULL,  
  sale_date DATE NOT NULL,  
  amount DECIMAL(10, 2)  
)  
PARTITION BY RANGE (YEAR(sale_date)) (  
  PARTITION p0 VALUES LESS THAN (2018),  
  PARTITION p1 VALUES LESS THAN (2019),  
  PARTITION p2 VALUES LESS THAN (2020),  
  PARTITION p3 VALUES LESS THAN MAXVALUE  
);
```

2. **List Partitioning:** Data is divided into partitions based on column values matching a predefined list. It's useful when the column has a limited set of possible values.

Example: Partitioning a products table into different partitions for each category, such as Electronics, Apparel, Furniture, etc.

```
CREATE TABLE products (  
  product_id INT AUTO_INCREMENT PRIMARY KEY,  
  category VARCHAR(255)  
)  
PARTITION BY LIST (category) (  
  PARTITION electronics VALUES IN ('Electronics', 'Gadgets'),  
  PARTITION apparel VALUES IN ('Apparel', 'Clothing'),  
  PARTITION furniture VALUES IN ('Furniture')  
);
```

3. **Hash Partitioning:** This type employs a hash function on one or more columns to determine the partition where each row should be placed. It's designed to ensure even distribution of data across partitions.

Example: Partitioning user data across several partitions by applying a hash function to user IDs.

```
CREATE TABLE users (  

```



```
user_id INT AUTO_INCREMENT PRIMARY KEY,  
username VARCHAR(255)  
)  
PARTITION BY HASH (user_id) PARTITIONS 4;
```

3. Concepts on range partitioning:

In MySQL, RANGE partitioning mode allows us to specify various ranges for which data is assigned. Ranges should be contiguous but not overlapping and are defined using the VALUES LESS THAN operator. In the following example, the sale_mast table contains four columns bill_no, bill_date, cust_code, and amount. This table can be partitioned by range in various of ways, depending on your requirements. Here we have used the bill_date column and decided to partition the table 4 ways by adding a PARTITION BY RANGE clause. In these partitions, the range of the sale date (sale_date) is as follows:

- partition p0 (sale between 01-01-2013 to 31-03-2013)
- partition p1 (sale between 01-04-2013 to 30-06-2013)
- partition p2 (sale between 01-07-2013 to 30-09-2013)
- partition p3 (sale between 01-10-2013 to 30-12-2013)

Adding partitions before creating the table that will structure the data beforehand and will make things easier. Here we have created a separate table to showcase how we partition the raw data before creating the table.

```
CREATE TABLE sale_mast (  
    bill_no INT NOT NULL,  
    bill_date TIMESTAMP NOT NULL,  
    cust_code VARCHAR(15) NOT NULL,  
    amount DECIMAL(8,2) NOT NULL  
)  
PARTITION BY RANGE (UNIX_TIMESTAMP(bill_date)) (  
    PARTITION p0 VALUES LESS THAN (UNIX_TIMESTAMP('2013-04-01')),  
    PARTITION p1 VALUES LESS THAN (UNIX_TIMESTAMP('2013-07-01')),  
    PARTITION p2 VALUES LESS THAN (UNIX_TIMESTAMP('2013-10-01')),  
    PARTITION p3 VALUES LESS THAN (UNIX_TIMESTAMP('2014-01-01'))  
);
```

NOTE:

- This SQL statement creates a new table named "sale_mast" with four columns and defines range-based partitioning based on the UNIX timestamp of the "bill_date" column.
- sale_mast is the name of the new table being created.
- **PARTITION BY RANGE (UNIX_TIMESTAMP(bill_date))**: Specifies that the table will be partitioned by range based on the UNIX timestamp of the "bill_date" column.
- The partitions are defined as follows:
 - **PARTITION p0 VALUES LESS THAN (UNIX_TIMESTAMP('2013-04-01'))**: Partition "p0" contains rows with bill dates before April 1, 2013.


- **PARTITION p1 VALUES LESS THAN (UNIX_TIMESTAMP('2013-07-01')):**
Partition "p1" contains rows with bill dates before July 1, 2013.
- **PARTITION p2 VALUES LESS THAN (UNIX_TIMESTAMP('2013-10-01')):**
Partition "p2" contains rows with bill dates before October 1, 2013.
- **PARTITION p3 VALUES LESS THAN (UNIX_TIMESTAMP('2014-01-01')):**
Partition "p3" contains rows with bill dates before January 1, 2014.

Now let's insert some of the data in the table and notice how the data gets impacted after partitioning.

```
INSERT INTO sale_mast VALUES (1, '2013-01-02', 'C001', 125.56),
(2, '2013-01-25', 'C003', 456.50),
(3, '2013-02-15', 'C012', 365.00),
(4, '2013-03-26', 'C345', 785.00),
(5, '2013-04-19', 'C234', 656.00),
(6, '2013-05-31', 'C743', 854.00),
(7, '2013-06-11', 'C234', 542.00),
(8, '2013-07-24', 'C003', 300.00),
(8, '2013-08-02', 'C456', 475.20);
```

```
SELECT * FROM sale_mast
```

Result Grid



Filter Rows:

Export:

	bill_no	bill_date	cust_code	amount
▶	1	2013-01-02 00:00:00	C001	125.56
	2	2013-01-25 00:00:00	C003	456.50
	3	2013-02-15 00:00:00	C012	365.00
	4	2013-03-26 00:00:00	C345	785.00
	5	2013-04-19 00:00:00	C234	656.00
	6	2013-05-31 00:00:00	C743	854.00
	7	2013-06-11 00:00:00	C234	542.00
	8	2013-07-24 00:00:00	C003	300.00
	8	2013-08-02 00:00:00	C456	475.20

sale_mast 4

×

This seems very normal in terms of output and we can't notice the partitioning of the data in the above output. To notice the partitioning of the data we will run the following query:

```
SELECT PARTITION_NAME, TABLE_ROWS FROM INFORMATION_SCHEMA.PARTITIONS
WHERE TABLE_NAME='sale_mast';
```


Result Grid	Filter Rows:
PARTITION_NAME	TABLE_ROWS
p0	4
p1	3
p2	2
p3	0

In this output, we can notice how the data has been partitioned.

Let's understand how we can alter the table and its partitions.

If you feel some data are useless in a partitioned table you can drop one or more partition(s). To delete all rows from partition p0 of sale_mast, you can use the following statement :

```
ALTER TABLE sale_mast TRUNCATE PARTITION p0;
```

Result Grid |  Filter Rows:

	PARTITION_NAME	TABLE_ROWS
▶	p0	0
	p1	6
	p2	4
	p3	0

Let's see another example:

```
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));
```

Now let's insert data into the table that we created.

```
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);
```

We can see the partition created by the CREATE TABLE statement using the below query:

```
SELECT TABLE_NAME, PARTITION_NAME, TABLE_ROWS, AVG_ROW_LENGTH,
DATA_LENGTH
FROM INFORMATION_SCHEMA.PARTITIONS
WHERE TABLE_NAME = 'Sales';
```

Result Grid					
		Filter Rows:		Export:	Wrap Cell Content:
	TABLE_NAME	PARTITION_NAME	TABLE_ROWS	AVG_ROW_LENGTH	DATA_LENGTH
▶	sales	p0	2	8192	16384
	sales	p1	2	8192	16384
	sales	p2	2	8192	16384
	sales	p3	3	5461	16384

4. Concepts on 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.

In the below example, suppose we have 12 stores distributed among four franchises based on their region.

We can partition the regions according to the rows for stores belonging to the same region will be stored in the same partition. The following statement arranges the stores in the same region using LIST partitioning, as shown below:

```
CREATE TABLE sales1 (
  sale_id INT,
  product_id INT,
  sale_date DATE,
  category VARCHAR(20),
  amount DECIMAL(10, 2)
)
PARTITION BY LIST COLUMNS (category) (
  PARTITION p_electronics VALUES IN ('Electronics'),
  PARTITION p_clothing VALUES IN ('Clothing'),
  PARTITION p_furniture VALUES IN ('Furniture'),
  PARTITION p_books VALUES IN ('Books')
);
```

```
INSERT INTO sales1 (sale_id, product_id, sale_date, category, amount)
VALUES
(1, 101, '2024-01-01', 'Electronics', 199.99),
(2, 102, '2024-01-02', 'Clothing', 49.99),
(3, 103, '2024-01-03', 'Furniture', 299.99),
(4, 104, '2024-01-04', 'Books', 19.99),
(5, 105, '2024-01-05', 'Electronics', 499.99),
(6, 106, '2024-01-06', 'Clothing', 89.99),
```




```
(7, 107, '2024-01-07', 'Furniture', 1299.99),
(8, 108, '2024-01-08', 'Books', 9.99),
(9, 109, '2024-01-09', 'Electronics', 299.99),
(10, 110, '2024-01-10', 'Clothing', 59.99),
(11, 111, '2024-01-11', 'Furniture', 799.99),
(12, 112, '2024-01-12', 'Books', 14.99),
(13, 113, '2024-01-13', 'Electronics', 399.99),
(14, 114, '2024-01-14', 'Clothing', 109.99),
(15, 115, '2024-01-15', 'Furniture', 499.99),
(16, 116, '2024-01-16', 'Books', 24.99),
(17, 117, '2024-01-17', 'Electronics', 599.99),
(18, 118, '2024-01-18', 'Clothing', 79.99),
(19, 119, '2024-01-19', 'Furniture', 699.99),
(20, 120, '2024-01-20', 'Books', 29.99);
```

If we want to check the partitions we can use the following query to check all the criteria that we have mentioned while creating partitions.

NOTE: Here the table_schema will be the name of the schema that you have created and it's not necessary to keep the name of the schema as 'Testing' or anything as such.

```
SELECT PARTITION_NAME, SUBPARTITION_NAME, PARTITION_ORDINAL_POSITION,
SUBPARTITION_ORDINAL_POSITION, PARTITION_METHOD, SUBPARTITION_METHOD,
PARTITION_EXPRESSION, SUBPARTITION_EXPRESSION, PARTITION_DESCRIPTION
FROM
    INFORMATION_SCHEMA.PARTITIONS
WHERE
    TABLE_SCHEMA = 'testing2'
    AND TABLE_NAME = 'sales1';
```

Output:

Result Grid  Filter Rows: <input type="text"/> Export:  Wrap Cell Content: 								
PARTITION_NAME	SUBPARTITION_NAME	PARTITION_ORDINAL_POSITION	SUBPARTITION_ORDINAL_POSITION	PARTITION_METHOD	SUBPARTITION_METHOD	PARTITION_EXPRESSION	SUBPARTITION_EXPRESSION	SUBPARTITION_DESCRIPTION
p_books	NULL	4	NULL	LIST COLUMNS	NULL	'category'	NULL	NULL
p_clothing	NULL	2	NULL	LIST COLUMNS	NULL	'category'	NULL	NULL
p_electronics	NULL	1	NULL	LIST COLUMNS	NULL	'category'	NULL	NULL
p_furniture	NULL	3	NULL	LIST COLUMNS	NULL	'category'	NULL	NULL

5. Concepts on 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.

This statement is used to create a table Store using the CREATE TABLE command and uses hashing on the store_id column that divides it into four partitions:




```
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 HASH(store_id)  
PARTITIONS 4;
```

NOTE: If you do not use the PARTITIONS clause, the number of partitions will be one by default. If you do not specify the number with the PARTITIONS keyword, it will throw an error.

```
INSERT INTO Stores (cust_name, bill_no, store_id, bill_date, amount)  
VALUES  
( 'Alice', 'B001', 1, '2024-01-01', 150.75),  
( 'Bob', 'B002', 2, '2024-01-02', 200.00),  
( 'Charlie', 'B003', 3, '2024-01-03', 99.99),  
( 'David', 'B004', 4, '2024-01-04', 175.50),  
( 'Eva', 'B005', 5, '2024-01-05', 250.00),  
( 'Frank', 'B006', 6, '2024-01-06', 300.75),  
( 'Grace', 'B007', 7, '2024-01-07', 80.25),  
( 'Hannah', 'B008', 8, '2024-01-08', 120.50),  
( 'Ivan', 'B009', 9, '2024-01-09', 450.00),  
( 'Jack', 'B010', 10, '2024-01-10', 60.00),  
( 'Karen', 'B011', 11, '2024-01-11', 110.75),  
( 'Leo', 'B012', 12, '2024-01-12', 220.00),  
( 'Mia', 'B013', 13, '2024-01-13', 330.50),  
( 'Nathan', 'B014', 14, '2024-01-14', 55.00),  
( 'Olivia', 'B015', 15, '2024-01-15', 95.25),  
( 'Paul', 'B016', 16, '2024-01-16', 500.00);
```

```
SELECT TABLE_NAME, PARTITION_NAME, TABLE_ROWS, AVG_ROW_LENGTH,  
DATA_LENGTH  
FROM INFORMATION_SCHEMA.PARTITIONS  
WHERE TABLE_NAME = 'Stores';
```

Output:

Result Grid  Filter Rows: <input type="text"/> Export:  Wrap Cell Content: 					
	TABLE_NAME	PARTITION_NAME	TABLE_ROWS	AVG_ROW_LENGTH	DATA_LENGTH
▶	stores	p0	4	4096	16384
	stores	p1	4	4096	16384
	stores	p2	4	4096	16384
	stores	p3	4	4096	16384

6. Altering data partitioning:

Let's consider that a table exists named stores and the table has been partitioned by store_id. Now you realize that the table has certain entries where store_id is more than 16 and you want to make another bucket of less than 20.

```
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 RANGE (store_id) (
  PARTITION p0 VALUES LESS THAN (4),
  PARTITION p1 VALUES LESS THAN (8),
  PARTITION p2 VALUES LESS THAN (12),
  PARTITION p3 VALUES LESS THAN (16)
);
```

Using the ALTER TABLE function we can add a partition.

Note: Whenever we have to add a new partition we have to add it after the range that is already created and cannot add a new partition in between the previously created partitions. If we want to add a new partition in between the previously created partition we have to drop the previous partition and create the new partition.

```
ALTER TABLE Stores
ADD PARTITION (
  PARTITION p4 VALUES LESS THAN (20)
);
```

Important Considerations:

- **Ensure No Overlap:** Make sure the new partition does not overlap with the existing partitions.
- **Data Redistribution:** Adding a partition does not automatically redistribute existing data.

You may need to manually manage the data if required.

- **Performance Impact:** Adding partitions can impact performance. Test the impact on your queries and update accordingly.



codingninjas