



# SQL Constraints & Functions

# Agenda

- SQL Constraints
- Constraint with ALTER
- Where clause
  - Predicates
  - Wildcard Filtering
- Set Operations
- Duplicate rows
- Operator precedence
- SQL Built-in function

# SQL Constraints

- SQL constraints are used to specify rules for data in a table
- Constraints can be specified when the table is created with the CREATE TABLE statement

Syntax:

```
CREATE TABLE table_name (  
    column1 datatype constraint,  
    column2 datatype constraint,  
    column3 datatype constraint,  
    ...);
```

# SQL Constraints

The following constraints are commonly used in SQL

Constraints	Meaning
NOT NULL	Ensures that a column cannot have a NULL value
UNIQUE	Ensures that all values in a column are different
PRIMARY KEY	A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table

# SQL Constraints

Constraints	Meaning
FOREIGN KEY	Uniquely identifies a row/record in another table
CHECK	Ensures that all values in a column satisfies a specific condition
DEFAULT	Sets a default value for a column when no value is specified
INDEX	Used to create and retrieve data from the database very quickly



# Not Null-Constraint

# SQL NOT NULL Constraint

- By default, a column can hold NULL values
- The NOT NULL constraint enforces a column to NOT accept NULL values
- This enforces a field to always contain a value
- You cannot insert a new record, or update a record without adding a value to this

# SQL NOT NULL on CREATE TABLE

- The following SQL ensures that the "id", "first\_name", and "last\_name" columns will NOT accept NULL values:

```
CREATE TABLE customers ( id int Not Null
                        first_name varchar(20) Not
                        Null
                        last_name varchar(20) Not Null
                        country varchar(20)
) ;
```





# Unique - Constraint

# SQL UNIQUE Constraint

- The UNIQUE constraint ensures that all values in a column are different.
- A PRIMARY KEY constraint automatically has a UNIQUE constraint.
- You can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table

# SQL UNIQUE Constraint on CREATE TABLE

- The following SQL creates a UNIQUE constraint on the "ID" column when the "company" table is created:

```
CREATE TABLE customers ( id int Not Null UNIQUE,  
                           first_name varchar(20) Not  
                           Null,  
                           last_name varchar(20) Not  
                           Null,  
                           country varchar(20)  
                           ) ;
```



# Primary Key-Constraint

# SQL PRIMARY KEY Constraint

- The PRIMARY KEY constraint uniquely identifies each record in a table
- Primary keys must contain UNIQUE values, and cannot contain NULL values
- A table can have only ONE primary key

# SQL PRIMARY KEY Constraint on CREATE TABLE

- The following SQL creates a PRIMARY KEY on the "id" column when the "customers" table is created:

```
CREATE TABLE customers ( id int Not Null,  
                           first_name varchar(20) Not  
                           Null, last_name varchar(20)  
                           Not  
                           Null,  
                           country varchar(20),  
                           PRIMARY KEY (id)  
                           );
```



# Foreign Key - Constraint

# SQL FOREIGN KEY Constraint

- A FOREIGN KEY is a key used to link two tables together
- It is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table
- The table containing the foreign key is called the child table
- The table containing the candidate key is called the referenced or parent table



# SQL FOREIGN KEY Constraint - Data

CustomerID	First name	Last name	country
------------	------------	-----------	---------

1	Mike	Christensen	USA
---	------	-------------	-----

2	Andy	Hollands	Australia
---	------	----------	-----------

3	Ravi	Vedantam	India
---	------	----------	-------

→ Customer table

OrderID	OrderNumber	CustomerID
---------	-------------	------------

1	56675	3
---	-------	---

2	74498	3
---	-------	---

3	85467	2
---	-------	---

4	42986	1
---	-------	---

→ Orders table

# SQL FOREIGN KEY Constraint

- Notice that the "CustomerID" column in the "Orders" table points to the "CustomerID" column in the "Customer" table
- The "CustomerID" column in the "Customer" table is the PRIMARY KEY in the "customer" table
- The "CustomerID" column in the "Orders" table is a FOREIGN KEY in the "Orders" table

# SQL FOREIGN KEY Constraint

- The FOREIGN KEY constraint is used to prevent:
  - actions that would destroy links between tables
  - invalid data being inserted into the foreign key column, because it has to be one of the values contained in the table it points to

# SQL FOREIGN KEY on CREATE TABLE

- The following SQL creates a FOREIGN KEY on the "CustomerID" column when the "Orders" table is created:

```
CREATE TABLE Orders ( OrderID int NOT NULL,  
                        OrderNumber int NOT NULL,  
                        customerID int,  
                        PRIMARY KEY (OrderID) ,  
                        FOREIGN KEY (CustomerID)  
                        REFERENCES Customer (CustomerID)  
                        ) ;
```



# Constraint using ALTER

# The Alter Query - Add Clause

## Assigning Primary Key Constraint to an Existing Field

- To create a PRIMARY KEY constraint on a column when the table is already created, we use the *ADD* clause with the *ALTER* command

Syntax:

```
ALTER TABLE table_name ADD PRIMARY KEY (column_name) ;
```

# The Alter Query - Add Clause

- Consider the previously created table *Customer*:

Field	Type	Null	Key	Default	Extra
customer_id	int(11)	YES		NULL	
First_name	varchar(10)	YES		NULL	
last_name	varchar(10)	YES		NULL	
City	varchar(20)	YES		NULL	
Total_exp	varchar(10)	YES		NULL	
Salary	int(11)	YES		NULL	

- Here, we need to declare "customer\_id" as primary key

# The Alter Query - Add Clause

- Use below *alter* query to add primary key to an existing field 'customer\_id':

```
ALTER TABLE Customer ADD PRIMARY KEY(customer_id) ;
```

- Use **describe Customer** to check if the primary key has been assigned to the field

Field	Type	Null	Key	Default	Extra
customer_id	int(11)	NO	PRI	NULL	
First_name	varchar(10)	YES		NULL	
last_name	varchar(10)	YES		NULL	
City	varchar(20)	YES		NULL	
Total_exp	varchar(10)	YES		NULL	
Salary	int(11)	YES		NULL	



# The Alter Query - Drop Clause

- The *Drop* clause with the alter command allows you to:
  - Delete a column from the table
  - Removing the constraint from a column

# The Alter Query - Drop Clause

## Dropping the Primary Key Constraint from a Field

- To drop a PRIMARY KEY constraint from an already created table, use below syntax

Syntax:

```
ALTER TABLE table_name DROP PRIMARY KEY _
```



*There is only one Primary Key in a table. Hence, we don't need to specify the name of the column while dropping the Primary Key using the ALTER command*

# The Alter Query - Drop Clause

- Consider a table *Customer* with below fields:

Field	Type	Null	Key	Default	Extra
customer_id	int(11)	YES		NULL	
First_name	varchar(10)	NO	PRI	NULL	
last_name	varchar(10)	YES		NULL	
City	varchar(20)	YES		NULL	
Total_exp	varchar(10)	YES		NULL	
Salary	int(11)	YES		NULL	

- It would be not be a good practise to declare 'First\_name' as Primary Key

# The Alter Query - Drop Clause

- Use below *alter* query to drop primary key from an already defined primary key field 'First\_name':

```
ALTER TABLE Customer DROP PRIMARY KEY;
```

- Use **describe Customer** to check if the primary key has been drop from the field

Field	Type	Null	Key	Default	Extra
customer_id	int(11)	YES		NULL	
First_name	varchar(10)	NO		NULL	
last_name	varchar(10)	YES		NULL	
City	varchar(20)	YES		NULL	
Total_exp	varchar(10)	YES		NULL	
Salary	int(11)	YES		NULL	



# Row Selection

# Where Clause Predicates

- A Where clause predicate is an expression which evaluates to a Boolean value to determine which row or a set of rows are relevant to a particular query
- By using predicates in the where clause we can filter out unwanted rows

# Where Clause Predicates

- The Where Clause supports the following types of predicates :
  - Comparison
    - = Equal
    - <> Not Equal
    - < Less Than
    - <= Less Than Or Equal
  - Pattern Matching
    - LIKE ( Used for Wildcard Filtering)
  - BETWEEN
  - IN
  - IS NULL



# Where Clause Predicates

- Consider the table, tblUser, and let us build some query examples using it

UserID	Name	City	Salary
1	Atanu Chakraborty	Kolkata	15000
2	Hriday Bharadwaj	Lucknow	27000
3	Birendra Maity	Hyderabad	68000
4	Raj Patil	Bangalore	71000
22	Bikram Ganguly	Kolkata	51000
31	Tapan Chatterjee	Kolkata	25000
33	Jai Prakash Bhatt	Jaipur	81000
34	Keith Kosta	Mumbai	77000
35	Mark Tailor	Bangalore	52000
36	Christopher Swan	Null	98000
39	Paul Grant	Mumbai	67000

# Where Clause Predicates - Comparison Operator

- Let's say you want to retrieve UserID, Name and Salary of users where user's salary is greater than 50000

```
SELECT UserID, Name, Salary FROM tblUser Where Salary > 50000
```

Output:



# Where Clause Predicates - Comparison Operator

- We will use the '=' operator to retrieve records where users belong to Kolkata

```
SELECT * FROM tblUser WHERE city = 'Kolkata';
```

Output:

UserID	Name	City	Salary
1	Atanu Chakraborty	Kolkata	15000
22	Bikram Ganguly	Kolkata	51000
31	Tapan Chatterjee	Kolkata	25000

# Where Clause Predicates - *Between* Predicate

- BETWEEN predicate is used to select rows within a specified range of values
- The below query will display the users whose userID's are in between 33 and 36:

```
SELECT UserID, Name FROM tblUser WHERE UserID  
        BETWEEN 33 AND 36;
```

Output:

UserID	Name
33	Jai Prakash Bhatt
34	Keith Kosta
35	Mark Tailor
36	Christopher Swan

# Where Clause Predicates - *IN* Predicate

- IN predicate is used to select rows within a set of specified values
- The below query will display the users whose UserIDs are 4, 22 and 36:

```
SELECT UserID,Name FROM tblUser WHERE UserID IN (4,22,36)
```

Output:

UserID	Name
4	Raj Patil
22	Bikram Ganguly
36	Christopher Swan

# Where Clause Predicates - *IS NULL* Predicate

- *IS NULL* is used to select rows if a specified column value contains a null value
- For Example:

```
SELECT * FROM tblUser WHERE City  
IS NULL
```

Output:

UserID	Name	City	Salary
36	Christopher Swan	NULL	98000



Where Clause Predicate - *Like* Predicate

# Wildcard Filtering

- Wildcard filtering is a method for comparing and displaying texts in a column against a search pattern
- Wildcard filtering is generally used in cases when you:
  - Do not know how to spell someone's last name, but you know what letter it starts with
  - Do not know the name of a song but you know the lyrics
  - Want to search a product based on its description in case you don't remember the product name



# Why Use Wildcard Filtering?

- Using wildcard in MySQL can increase the performance of an application
- It can reduce the time to filter the record from the database
- Complex SQL queries can be converted into simple one using wildcards
- Using wildcards we can develop powerful search engines in a large data-driven application. Searching in the data-driven application are much more dependent on the use of wildcards

# Wildcards

- Wildcards are characters that are used to filter/search data from the database on the basis of certain patterns
- They are often used with the operators like *LIKE* and *NOT LIKE* in conjunction with the *WHERE* clause

# Wildcards

- Types of Wildcards:

Type	Description
%	The percent character indicates any character with any number of counts
-	The hyphen character indicates a range of the character within the braces [a-c]
_	The underscore character indicates any one character
[]	The square bracket indicates any one value with the brackets
^	The caret character indicates all the character except available in the brackets: [^xyz]
#	The hash sign indicates a single numeric character

# Wildcards

- To carry out the wildcard filtering exercise we use movie lens dataset
- Follow the below instructions to load the data in a mysql table:
  - Download the data using the link:  
<http://files.grouplens.org/datasets/movielens/ml-latest.zip>
  - Run the query:

```
SHOW VARIABLES LIKE "secure_file_priv";
```

Variable_name	Value
secure_file_priv	C:\ProgramData\MySQL\MySQL Server 8.0\Uploads\

copy movies.csv to this  
folder

# Wildcards

- Create tables movies:

```
CREATE TABLE MOVIES (  
    movieId int,  
    title varchar(200),  
    genres varchar(200));
```

- Load the data in table using:

```
LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server
```

# Wildcard Filtering - '%' Wildcard

- % wildcard is used in searching or filtering a record by matching any string of zero or more character
- It has the following syntax:

```
SELECT statement...WHERE column_name LIKE 'xxx%'
```

Here,

- "**SELECT statement...**" is the standard SQL SELECT command.
- "**WHERE**" is the key word used to apply the filter.
- "**LIKE**" is the comparison operator that is used in conjunction with wildcards
- '**xxx**' is any specified starting pattern such as a single character or more and "%" matches any number of characters starting from zero (0).

# Wildcard Filtering - '%' Wildcard

- % can be used either in the first place, in the last or at both side of the string
- If used in the first place ("%xxx"), all those rows will be extracted from the database where the column will contain strings ending with 'xxx'
- If used at both the ends ("%xxx%"), all those rows will be extracted from the database where the column will contain the string 'xxx' anywhere in the column string
- If used at the last ("xxx%"), all those rows will be extracted from the database where the column will contain string starting with 'xxx'

# Wildcard Filtering - '%' Wildcard

- Let us understand these usage one-by-one with examples

## Scenario - 1

- Suppose you forgot the complete name of the movie that you finished watching a few days ago, say, "Eternal Sunshine of the Spotless Mind". You only remember that the name started with the word "Eternal"
- Searching each and every movie in the database will not be a good option



# Wildcard Filtering - '%' Wildcard

- Below is the SQL statement that can help you filter records

```
SELECT * FROM movies WHERE title LIKE 'eternal%';
```

movieId	title	genres
7361	Eternal Sunshine of the Spotless Mind (2004)	Drama Romance Sci-Fi
88493	Eternally Yours (1939)	Comedy Drama
101986	Eternal Return, The (L'éternel retour) (1943)	Drama Romance
159241	Eternal Summer (2015)	Drama Romance
169016	Eternal Summer (2006)	Drama Romance
181443	Eternal Family (1997)	Animation Comedy
184785	Eternal Homecoming (2013)	Drama
186335	Eternal Winter (2018)	Drama Romance
191987	Eternal Fire (Fuego eterno) (2012)	Documentary

- You can now easily recall the name of the movie by looking at the above output

# Wildcard Filtering - '%' Wildcard

- The query translates to:
  - Select all the columns from the *Movies* table
  - Return the rows whose title begins with the string 'eternal' and is followed by zero or more occurrences of any character

# Wildcard Filtering - '%' Wildcard

## Scenario - 2

- You are a *Harry Potter* fan and want to know the name of all the *Harry Potter* movies that are not based famous book series
- The only information that you have is that the name of all the *Harry Potter* movies based on the books started with the string *Harry Potter* . But is there any other movie based on *Harry Potter*?

# Wildcard Filtering - '%' Wildcard

- Below is the SQL statement that can help you take a look at all the movie titles based-on *Harry Potter*

```
SELECT * FROM movies WHERE title LIKE '%harry potter%';
```

movieId	title	genres
4896	Harry Potter and the Sorcerer's Stone (a.k.a. Harry Potter and the Philosopher's Stone) (2001)	Adventure Children Fantasy
5816	Harry Potter and the Chamber of Secrets (2002)	Adventure Fantasy
8368	Harry Potter and the Prisoner of Azkaban (2004)	Adventure Fantasy IMAX
40815	Harry Potter and the Goblet of Fire (2005)	Adventure Fantasy Thriller IMAX
54001	Harry Potter and the Order of the Phoenix (2007)	Adventure Drama Fantasy IMAX
69844	Harry Potter and the Half-Blood Prince (2009)	Adventure Fantasy Mystery Romance IMAX
81834	Harry Potter and the Deathly Hallows: Part 1 (2010)	Action Adventure Fantasy IMAX
88125	Harry Potter and the Deathly Hallows: Part 2 (2011)	Action Adventure Drama Fantasy Mystery IMAX
186777	The Greater Good - Harry Potter Fan Film (2013)	Action Adventure Fantasy

- There is one movie in the list, 'The Greater Good - Harry Potter Fan Film', which is not based on the *Harry Potter* book

# Wildcard Filtering - '%' Wildcard

## Scenario - 3

- You are a kind of person who loves to watch movies across all genres. Among all the genres, you find the 'film noir' genre intriguing
- You want to watch all the movies that carries a 'film noir' theme from the year 1990
- The challenge is we do not have a *year* column in the movie table. How do we retrieve those movies that belong to the 'film noir' category from the year 1990?

# Wildcard Filtering - '%' Wildcard

- Below is the SQL statement that can help you take a look at all film-noir movies from the year 1990

```
SELECT * FROM movies WHERE genres LIKE '%film-noir%' AND title LIKE "%1990%"
```

movieId	title	genres
1179	Grifters, The (1990)	Crime Drama Film-Noir
1245	Miller's Crossing (1990)	Crime Drama Film-Noir Thriller
26664	Moon 44 (1990)	Action Film-Noir Sci-Fi Thriller
42794	Buried Alive (1990)	Film-Noir Horror Thriller
66289	Singapore Sling (Singapore sling: O anthropos pou agapise ena ptoma) (1990)	Crime Film-Noir Horror Romance Thriller
113698	Alligator Eyes (1990)	Drama Film-Noir Mystery

# Wildcard Filtering - '\_' (underscore) Wildcard

- The underscore(\_) wildcard is used to match *exactly one character*
- It has the following syntax:

```
SELECT statement...WHERE column_name LIKE 'xxx_'
```

Here,

- "**SELECT statement...**" is the standard SQL SELECT command
- "**WHERE**" is the key word used to apply the filter
- "**LIKE**" is the comparison operator that is used in conjunction with wildcards
- '**xxx**' is any specified starting pattern such as a single character or more and \_ matches the string with only one characters

# Wildcard Filtering - '\_' (underscore) Wildcard

- \_ can be used either in the first place, in the last or at both side of the string
- If used in the first place ("\_xxx"), all those rows will be extracted from the database where the instance of column will contain string that starts with any single character and ends with 'xxx'
- If used at both the ends ("\_xxx\_"), all those rows will be extracted from the database where the column will contain the string 'xxx' that starts and ends with any single character
- If used at the last ("xxx\_"), all those rows will be extracted from the database where the column will contain string starting with 'xxx' followed by any single character
- You can use  $n$  number of underscore character for matching  $n$  number of character



# Wildcard Filtering - '\_' (underscore) Wildcard

- Let us understand these usage one-by-one with examples

## Scenario - 1

- Consider the movies table

movieId	title	genres
1	Toy Story (1995)	Adventure Animation Children Comedy Fantasy
2	Jumanji (1995)	Adventure Children Fantasy
3	Grumpier Old Men (1995)	Comedy Romance
4	Waiting to Exhale (1995)	Comedy Drama Romance
5	Father of the Bride Part II (1995)	Comedy

Most of the movie in the table are cross-genre (blended genres) movies

# Wildcard Filtering - '\_' (underscore) Wildcard

- You want to retrieve the name of all the movies that are not cross genre movies but are pure genre movie, say just the children category, from the year 2010
- Below is the SQL statement that can help you take a look at all the movie names from the year 2010 that belongs only to the children category

```
SELECT * FROM movies WHERE genres LIKE "children_" AND title LIKE '%2010%';
```

movieId	title	genres
120098	First Dog (2010)	Children
120416	Fuchsia the Mini-Witch (2010)	Children
128904	Runner from Ravenshead, The (2010)	Children
144958	Škola princů (2010)	Children
152161	What Would Jesus Do? (2010)	Children
164837	A Heartland Christmas (2010)	Children
191833	Den Brother (2010)	Children

# Wildcard Filtering - *NOT* Logical Operator

- The *NOT* logical operator can be used together with the wildcards to return rows that do not match the specified pattern
- Suppose we want to retrieve movies that were not released in the year 1900s
- We would use the NOT logical operator together with the underscore as well as the percent wildcard to get our results ( Yes!! You can use combination of wildcards)

# Wildcard Filtering - *NOT* Logical Operator

- Below is the script that retrieves movie name not belonging to 1900s

```
SELECT * FROM movies WHERE title NOT LIKE '%19__%';
```

movieId	title	genres
2769	Yards, The (2000)	Crime Drama
3177	Next Friday (2000)	Comedy
3190	Supernova (2000)	Adventure Sci-Fi Thriller
3225	Down to You (2000)	Comedy Romance
3228	Wirey Spindell (2000)	Comedy
3239	Isn't She Great? (2000)	Comedy
3273	Scream 3 (2000)	Comedy Horror Mystery Thriller
3275	Boondock Saints, The (2000)	Action Crime Drama Thriller
3276	Gun Shy (2000)	Comedy
3279	Knockout (2000)	Action Drama
3285	Beach, The (2000)	Adventure Drama
3286	Snow Day (2000)	Comedy
3287	Tigger Movie, The (2000)	Animation Children
3291	Trois (2000)	Thriller

- Notice that we have used two percent operator at both the ends and two underscore operators right after '19'

# Wildcard Filtering - *Escape* operator

- The *ESCAPE* keyword is used to escape pattern matching characters such as the (%) percentage and underscore (\_) if they form part of the data.
- For example, movies like *100% Love* has the % symbol in its title, if we try to search the name of such movies using wildcard filtering the % in the title would be treated as wildcard
- To make such characters treated as string, we use the *ESCAPE* keyword

# Wildcard Filtering - *NOT* Logical Operator

- We use below query to retrieve movie name that has the % character in its name:

```
SELECT * FROM movies WHERE title LIKE '%#%' ESCAPE '#';
```

movieId	title	genres
53498	Who the #\$&% is Jackson Pollock? (2006)	Documentary
91423	99 and 44/100% Dead (1974)	Action Adventure Comedy Crime
110973	Ultimate Accessory,The (100% cachemire) (2013)	Comedy
148801	10%: What Makes a Hero? (2013)	Adventure Documentary
152041	100% Love (2011)	Drama Romance
154838	Wool 100% (2006)	Animation Comedy Fantasy Sci-Fi
171641	Richard Pryor: I Ain't Dead Yet, #*%\$#@!! (2003)	Comedy
175557	Portrait of a '60% Perfect Man': Billy Wilder (1982)	Documentary
183567	F*&% the Prom (2017)	Comedy
189393	97% Owned (2012)	Documentary

- Here note the double "%%" in the LIKE predicate, the first one in red "%" is treated as part of the string to be searched for. The other one is used to match any number of characters that follow

# Wildcard Filtering - Summary

- The *LIKE* predicate & *WILDCARDS* are powerful tools that help search data matching complex patterns
- *The percentage ( % )* wildcard is used to match any number of characters starting from zero (0) and more characters
- *The underscore ( \_ )* wildcard is used to match exactly one character



# Set Operations



# Set Operations

- Set operators are used to join the results of two (or more) SELECT statements
- Types of SET operations:
  - UNION
  - UNION ALL
  - INTERSECT
  - MINUS

# Set Operations - Union

- The Union clause/operator is used to combine the results of two or more SELECT Statements with Identical columns without returning any duplicate rows

Syntax:

```
SELECT col1, col2, col3 FROM tables [WHERE  
conditions]  
UNION  
SELECT col1, col2, col3 FROM tables [WHERE  
conditions];
```

- There should be same number of columns in both SELECT statements
- The column names from the first SELECT statement in the UNION operator are used as the column names for the result set

# Set Operations - Union

- Consider a query to find the names of all users that live in Kolkata and all the users of any city who are getting salary of 50000 and above
- Let us design the query and examine the result.

```
SELECT UserID, Name, City, Salary FROM tblUser WHERE City = 'Kolkata'  
UNION  
SELECT UserID, Name, City, Salary FROM tblUser WHERE Salary >= 50000;
```

Output:

UserID	Name	City	Salary
1	Atanu Chakraborty	Kolkata	15000
22	Bikram Ganguly	Kolkata	51000
31	Tapan Chatterjee	Kolkata	25000
3	Birendra Maity	Hyderabad	68000
4	Raj Patil	Bangalore	71000
33	Jai Prakash Bhatt	Jaipur	81000
34	Keith Kosta	Mumbai	77000
35	Mark Tailor	Bangalore	52000
36	Christophar Swan	NULL	98000
39	Paul Grant	Mumbai	67000

# Set Operations - Union

- Let us run both the queries and check the results

First Query:

```
SELECT UserID, Name, City, Salary
FROM tblUser
WHERE City = 'Kolkata';
```

Output:

UserID	Name	City	Salary
1	Atanu Chakraborty	Kolkata	15000
22	Bikram Ganguly	Kolkata	51000
31	Tapan Chatterjee	Kolkata	25000

Second Query:

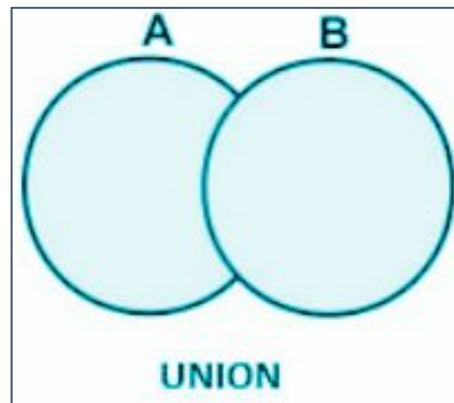
```
SELECT UserID, Name, City, Salary
FROM tblUser
WHERE Salary >= 50000;
```

Output:

UserID	Name	City	Salary
3	Birendra Maity	Hyderabad	68000
4	Raj Patil	Bangalore	71000
22	Bikram Ganguly	Kolkata	51000
33	Jai Prakash Bhatt	Jaipur	81000
34	Keith Kosta	Mumbai	77000
35	Mark Tailor	Bangalore	52000
36	Christophar Swan	NULL	98000
39	Paul Grant	Mumbai	67000

# Set Operations - Union

- In the result we can see UserID 22 appeared once but if we look into the query both query returns the same row if they run individually but while in the UNION operation they returned only once
- This is because the Union clause works just like the Union operation in Set theory where the duplicate members appear only once in the resultant set.



# Set Operations - Union All

- The Union All clause/operator is used to combine the results of two or more SELECT Statements into a single set of rows and columns without the removal of any duplicates

Syntax:

```
SELECT col1, col2, col3 FROM tables [WHERE  
conditions]  
UNION ALL  
SELECT col1, col2, col3 FROM tables [WHERE  
conditions];
```

# Set Operations - Union All

- Consider a query to find the names of all users that live in Kolkata and all the users of any city who are getting salary of 50000 and above
- Let us design the query and examine the result.

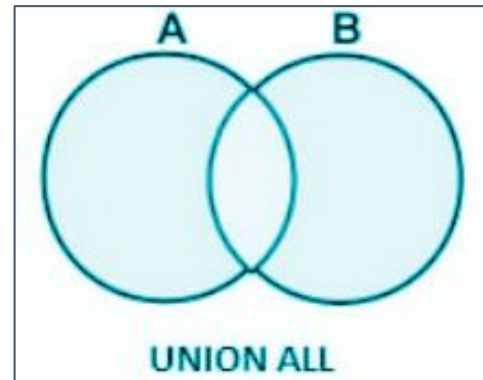
```
SELECT UserID, Name, City, Salary FROM tblUser WHERE City = 'Kolkata'  
UNION ALL  
SELECT UserID, Name, City, Salary FROM tblUser WHERE Salary >= 50000;
```

Output:

UserID	Name	City	Salary
1	Atanu Chakraborty	Kolkata	15000
22	Bikram Ganguly	Kolkata	51000
31	Tapan Chatterjee	Kolkata	25000
3	Birendra Maity	Hyderabad	68000
4	Raj Patil	Bangalore	71000
22	Bikram Ganguly	Kolkata	51000
33	Jai Prakash Bhatt	Jaipur	81000
34	Keith Kosta	Mumbai	77000
35	Mark Tailor	Bangalore	52000
36	Christophar Swan	NULL	98000
39	Paul Grant	Mumbai	67000

# Set Operations - Union All

- The result of both the select statements are combined into single set of rows and columns WITHOUT the removal of duplicates
- UserID 22 appears twice the view





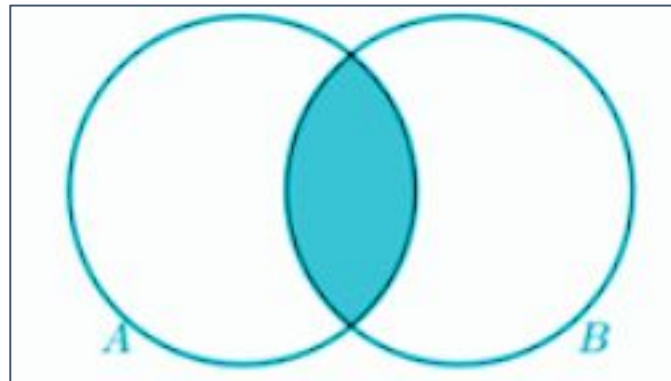
# Set Operations - Intersect

- The INTERSECT OPERATION is used to combine two SELECT statements with identical columns and returns rows only common rows returned by the two select statements
- We can use the intersect operations to display records where city is Kolkata and salary is greater than or equal to 50000

UserID	Name	City	Salary
22	Bikram Ganguly	Kolkata	51000

# Set Operations - Intersect

- In the result we can see UserID 22 is the common observation in the given two tables
- Intersect operates similar to intersect operation from set theory

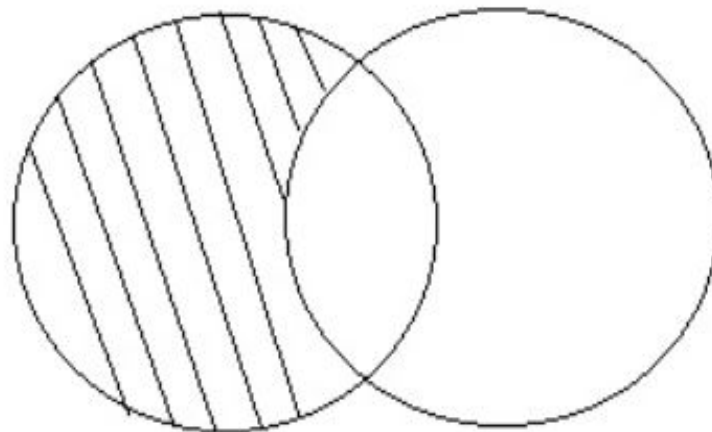




*MySQL DOES NOT support INTERSECT operator. However, you can emulate the INTERSECT operator using JOINS. We will study joins in the upcoming sessions.*

# Set Operations - Minus

- The MINUS OPERATION combines results of two SELECT statements and return only those in the final result, which belongs to the first set of the result





*MySQL DOES NOT support MINUS operator. However, you can emulate the MINUS operator using JOINS. We will study joins in the upcoming sessions.*



# Duplicate Rows

# Duplicate Rows

- Duplicate rows, or duplicate records, are identical rows in a table.
- It is not desired that a relation or a table have duplicate records
- Several constraints (e.g. Primary Key, Candidate Key, Unique Key etc.) during the database design phase can be used to avoid duplicate records
- There are still unavoidable situations where duplicate records get stored in the table
- For example, when a staging table is used and data is loaded from different sources, duplicate rows are possible

# Duplicate Rows

- The DISTINCT keyword in the SELECT statement eliminates duplicate rows and display a unique list of values

Syntax:

```
SELECT DISTINCT column_list FROM table_name
```



# Duplicate Rows

- Consider the table below:

Table : Store			
ItemID	ItemName	ItemQty	UnitPrice
1	Item A 1	20	151
2	Item A 1	20	151
3	Item C51	8	39
4	Item R36	31	75
5	Item B11	77	281
6	Item B11	77	281
7	Item R55	91	45
8	Item NB36	4	21
9	Item CR47	73	200
10	Item CR47	73	200
11	Item CR47	73	200
12	Item CR47	73	200

- Here the column ItemID is considered as a primary key and is a auto generated column which means the field values are not supplied from any outside source, rather they are generated automatically and the values are stored in a sequentially incremental fashion
- Hence we do not have any control over that column values but other column values are supplied from outside sources

# Duplicate Rows

- Use a below SELECT statement to get all the unique observations

Query:

```
SELECT DISTINCT * FROM  
Store
```

Output

:

itemID	ItemName	ItemQty	UnitPrice
2	Item A 1	20	151
3	Item C51	8	39
4	Item R36	31	75
6	Item B11	77	281
7	Item R55	91	45
8	Item NB36	4	21
12	Item CR47	73	200

The above view has no duplicate rows



# Operator Precedence

# Operator Precedence

- Consider the below Employee table:

ID	NAME	AGE	ADDRESS	SALARY
1	Kellie	32	California	2000
2	Pete	25	Texas	1500
3	Popy	23	Boston	2000
4	Sam	25	Florida	6500
5	Jhon	27	Hawaii	10000

- Write a query to display employees having salary greater than 1800 and whose name starts with either 's' or 'p'

```
SELECT * FROM Employee  
WHERE Salary > 1800 and Name like 's%' or  
Name like 'p%';
```

Output:

Id	Name	Age	Address	Salary
2	Pete	25	Texas	1500
3	Popy	23	Boston	2000
4	Sam	25	Florida	6500

- This query displays employee whose salary is less than 1800 in the result. This is due to the higher precedence of AND as compared to OR

# Operator Precedence

- Operator precedence specifies the order in which operators are evaluated when two or more operators with different precedence are adjacent in an expression
- These operator precedence rule greatly affects MySQL queries

# Operator Precedence

- To display employees having salary greater than 1800 and whose name starts with either 's' or 'p', we write the below query:

```
SELECT * FROM Employee  
WHERE Salary > 1800 and (Name like 's%' or Name  
like 'p%');
```

Output:

Id	Name	Age	Address	Salary
3	Popy	23	Boston	2000
4	Sam	25	Florida	6500

# Operator Precedence

- Operator precedences are shown in the following list, from highest precedence to the lowest. Operators that are shown together on a line have the same precedence

#	Operator	Description
1	INTERVAL	Return the index of the argument that is less than the first argument
2	BINARY, COLLATE	Binary: This is a type that stores binary byte strings Collate: This clause override whatever the default collation is for comparison
3	!	Negate values
4	- (unary minus), ~ (unary bit inversion)	Unary Minus: It change the sign of the operand Unary Bit Inversion: It inverts the bits of operand



# Operator Precedence

#	Operator	Description
5	<code>^</code>	Bitwise XOR
6	<code>*</code> , <code>/</code> , <code>%</code>	<code>*</code> : Multiplication operator <code>/</code> : Division operator <code>%</code> : Modulo Operator
7	<code>-</code> , <code>+</code>	<code>-</code> : Minus Operator <code>+</code> : Plus Operator
8	<code>&lt;&lt;</code> , <code>&gt;&gt;</code>	<code>&lt;&lt;</code> : Shift a (BIGINT) number or binary string to left  <code>&gt;&gt;</code> : Shift a (BIGINT) number or binary string to right

# Operator Precedence

#	Operator	Description
9	&	Bitwise AND
10		Bitwise OR
11	=, <=>, >=, >, <=, <, <> , !=, IS, LIKE, REGEXP, IN	= : Comparison Operator <=> : NULL-safe equal to Operator >= : Greater than or equal to > : Greater than <= : Less than or equal to < : Less than <> , != : Not equal to IS : Test value against a boolean value LIKE : Pattern matching operator REGEXP: Matches the string expression with the regular expression IN : Check whether a value is present in list or not

# Operator Precedence

#	Operator	Description
12	BETWEEN, CASE WHEN, THEN ELSE	BETWEEN: Check whether a value is within a range of values CASE WHEN, THEN ELSE: Case operator
13	NOT	Negates Value
14	AND, &&	Logical AND
15	XOR	Logical XOR
16	OR,	Logical OR
17	= (assignment), :=	Assign a value



# SQL Built-in Functions

# SQL Built-in Function

- Built in functions are functions that are shipped with MySQL
- They can be categorized according to the data types that they operate on i.e. strings, date and numeric built in functions

# SQL Built-in Function

- There are different types of SQL built - in functions, which are mentioned below:
  - String
  - Numeric
  - Date
  - Bin
  - Cast

# SQL Built-in Function

- We use the following Employee table to understand the built-in function

ID	NAME	AGE	ADDRESS	SALARY
1	Kellie	32	California	2000
2	Pete	25	Texas	1500
3	Popy	23	Boston	2000
4	Sam	25	Florida	6500
5	Jhon	27	Hawaii	10000



# NUMERIC Functions



# Numeric Function - Syntax

- The following syntax is used to perform any mathematical operation in any query

Syntax:

```
SELECT numerical_expression as OPERATION_NAME  
[FROM table_name WHERE CONDITION] ;
```

# Numeric Function - Example - 1

The numerical\_expression is used for a mathematical expression or any formula. Following is a simple example showing the usage of SQL Numeric Expressions

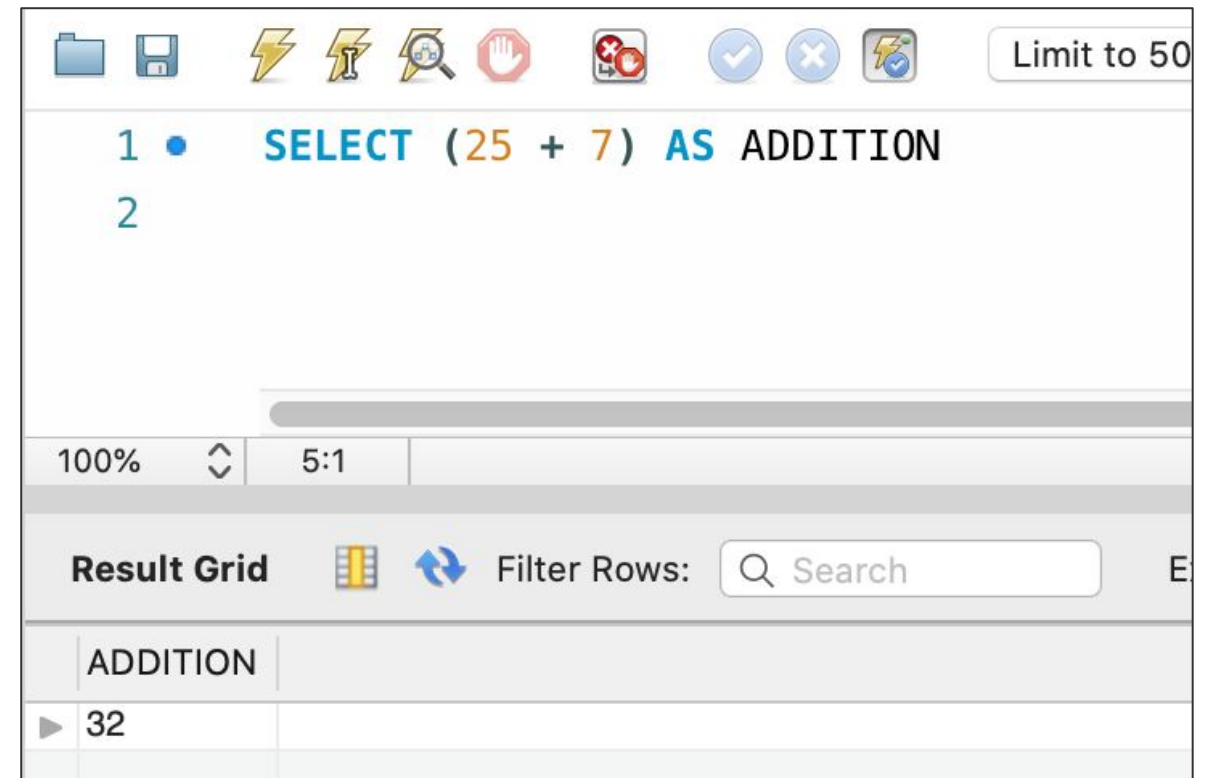
Syntax:

```
SELECT (25 + 7) AS ADDITION
```

Output:

ADDITION

32



# Numeric Function - Example - 2

The numerical\_expression is used for a mathematical expression or any formula. Following is a simple example showing the usage of SQL Numeric Expressions

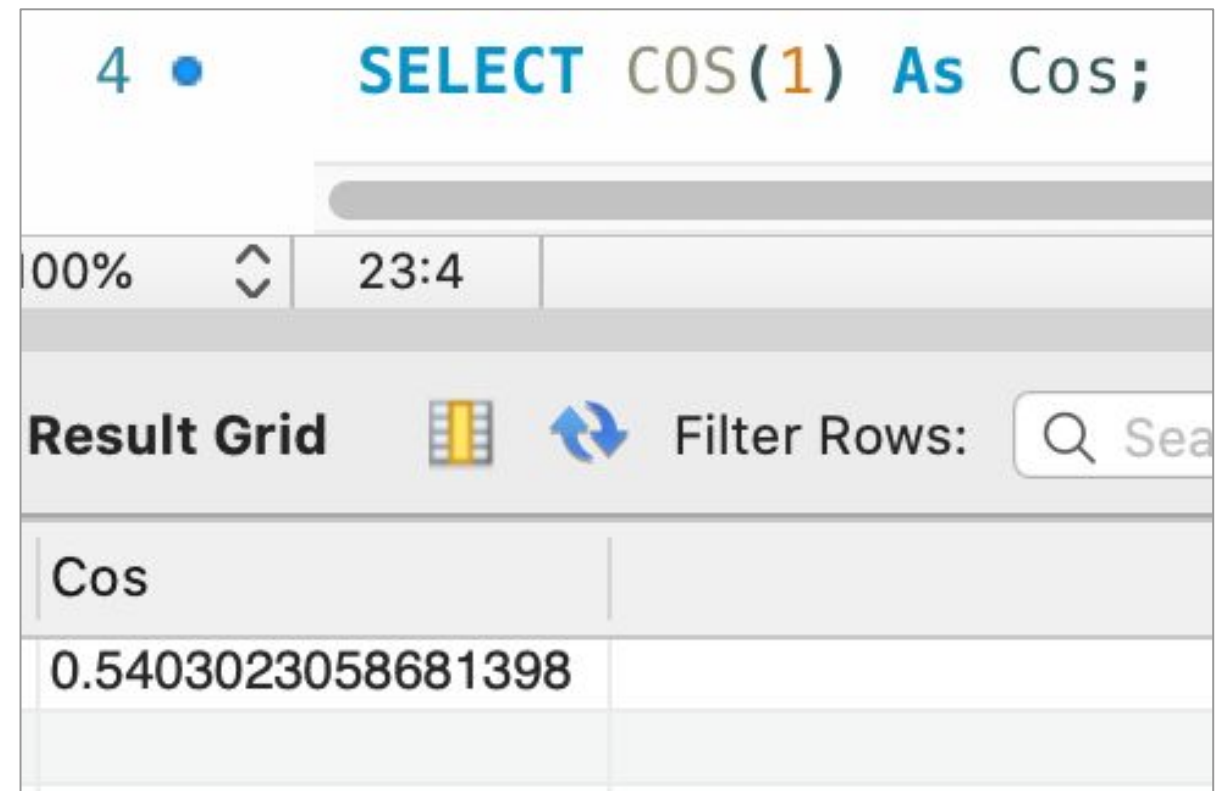
Syntax:

```
SELECT COS(1) As Cos;
```

Output:

Cos

0.5403023058681398



The screenshot shows a SQL query editor with the query `SELECT COS(1) As Cos;` and a result grid below it. The query editor has a line number 4 and a blue dot. The result grid has a header row with 'Cos' and a data row with the value '0.5403023058681398'. The interface includes a progress bar at 100%, a refresh icon, a search bar, and a 'Filter Rows' section.

Cos
0.5403023058681398

# Numeric Function - Example - 3

The numerical\_expression is used for a mathematical expression or any formula. Following is a simple example showing the usage of SQL Numeric Expressions

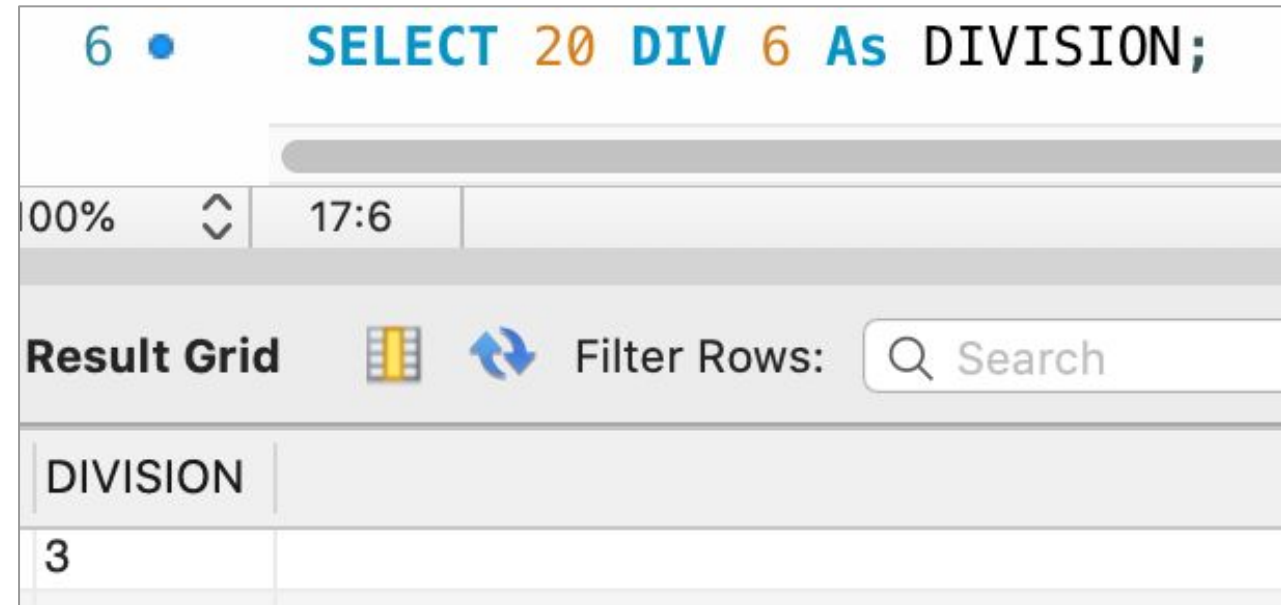
Syntax:

```
SELECT 20 DIV 6 As DIVISION;
```

Output:

DIVISION

3



The screenshot shows a SQL query editor with the query `SELECT 20 DIV 6 As DIVISION;` and a result grid below it. The result grid has a header row with the column name `DIVISION` and a data row with the value `3`. The interface includes a progress bar at 00%, a timer at 17:6, and a search bar.

DIVISION
3

# More Numeric Functions

- Following are the numeric functions defined in SQL

Functions	Meaning
<b>ABS</b>	It returns the absolute value of a number
<b>ACOS</b>	It returns the cosine of a number
<b>ASIN</b>	It returns the arc sine of a number
<b>ATAN</b>	It returns the arc sine of a number
<b>CEIL</b>	It returns the smallest integer value that is greater than or equal to a number
<b>CEILING</b>	It returns the smallest integer value that is greater than or equal to a number

# More Numeric Functions

Functions	Meaning
COS	It returns the cosine of a number
COT	It returns the cotangent of a number
DEGREES	It converts a radian value into degrees
DIV	It is used for integer division
EXP	It returns e raised to the power of number
FLOOR	It returns the largest integer value that is less than or equal to a number

# More Numeric Functions

Functions	Meaning
<b>GREATEST</b>	It returns the greatest value in a list of expressions
<b>LEAST</b>	It returns the smallest value in a list of expressions
<b>LN</b>	It returns the natural logarithm of a number
<b>LOG10</b>	It returns the base-10 logarithm of a number
<b>LOG2</b>	It returns the base-2 logarithm of a number
<b>MOD</b>	It returns the remainder of n divided by m

# More Numeric Functions

Functions	Meaning
PI	It returns the value of PI displayed with 6 decimal places
POW	It returns m raised to the nth power
RADIANS	It converts a value in degrees to radians
RAND	It returns a random number
ROUND	It returns a number rounded to a certain number of decimal places
SIGN	It returns a value indicating the sign of a number



# More Numeric Functions

Functions	Meaning
SIN	It returns the sine of a number
SQRT	It returns the square root of a number
TAN	It returns the tangent of a number
ATAN2	It returns the arctangent of the x and y coordinates, as an angle and expressed in radians
TRUNCATE	This doesn't work for SQL Server. It returns 7.53635 truncated to 2 places right of the decimal point



# String Functions

# More Numeric Functions

- String functions are used to perform an operation on input string and return an output string
- Following are the string functions defined in SQL

Functions	Meaning
ASCII	Return the ASCII code value of a character
CHAR	Convert an ASCII value to a character
CHARINDEX	Search for a substring inside a string starting from a specified location and return the position of the substring

# More Numeric Functions

Functions	Meaning
CONCAT	Join two or more strings into one string
CONCAT_WS	Concatenate multiple strings with a separator into a single string
DIFFERENCE	Compare the SOUNDEX() values of two strings
FORMAT	Return a value formatted with the specified format and optional culture
LEFT	Extract a given a number of characters from a character string starting from the left

# More Numeric Functions

Functions	Meaning
LEN	Return a number of characters of a character string
LOWER	Convert a string to lowercase
LTRIM	Return a new string from a specified string after removing all leading blanks
NCHAR	Return the Unicode character with the specified integer code, as defined by the Unicode standard
PATINDEX	Returns the starting position of the first occurrence of a pattern in a string

# More Numeric Functions

Functions	Meaning
QUOTENAME	Returns a Unicode string with the delimiters added to make the input string a valid delimited identifier
REPLACE	Replace all occurrences of a substring, within a string, with another substring
REPLICATE	Return a string repeated a specified number of times
REVERSE	Return the reverse order of a character string
RIGHT	Extract a given a number of characters from a character string starting from the right

# More Numeric Functions

Functions	Meaning
RTRIM	Return a new string from a specified string after removing all trailing blanks
REPLACE	Replace all occurrences of a substring, within a string, with another substring
SOUNDEX	Return a four-character (SOUNDEX) code of a string based on how it is spoken
SPACE	Returns a string of repeated spaces
STR	Returns character data converted from numeric data

# More Numeric Functions

Functions	Meaning
<code>STRING_AGG</code>	Concatenate rows of strings with a specified separator into a new string
<code>STRING_ESCAPE</code>	Escapes special characters in a string and returns a new string with escaped characters
<code>STRING_SPLIT</code>	A table-valued function that splits a string into rows of substrings based on a specified separator
<code>STUFF</code>	Delete a part of a string and then insert another substring into the string starting at a specified position



# More Numeric Functions

Functions	Meaning
SUBSTRING	Extract a substring within a string starting from a specified location with a specified length
TRANSLATE	Replace several single-characters, one-to-one translation in one operation
TRIM	Return a new string from a specified string after removing all leading and trailing blanks
UPPER	Convert a string to uppercase

# String Function - Example - 1

CHAR\_LENGTH(): Doesn't work for SQL Server. Use LEN() for SQL Server. This function is used to find the length of a word.

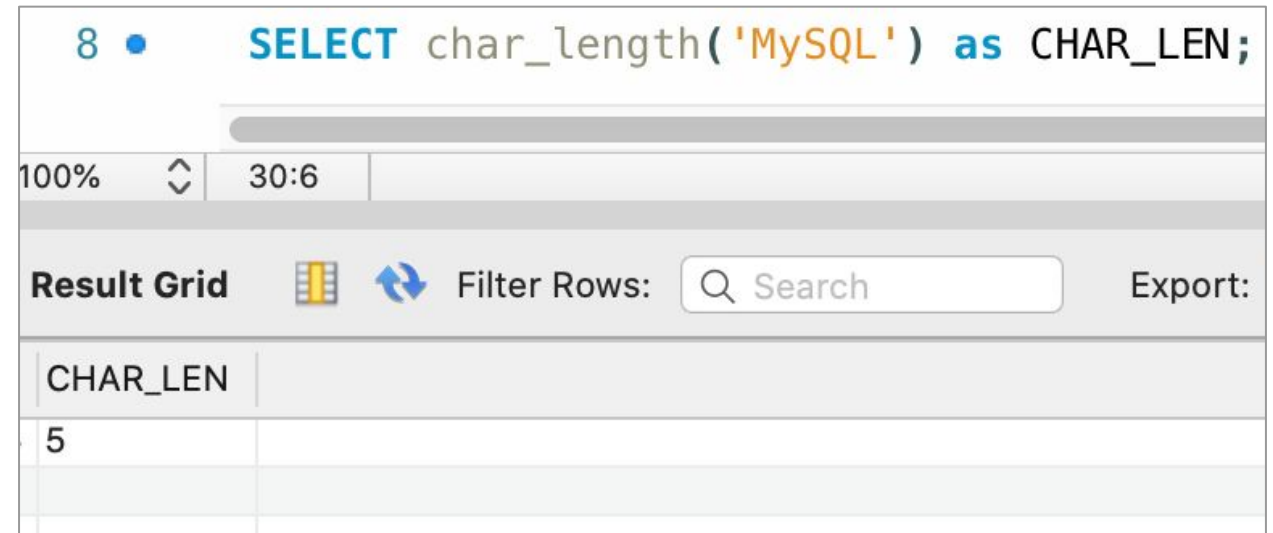
Syntax:

```
SELECT char_length('MySQL')  
as CHAR_LEN;
```

Output:

CHAR\_LEN

5



The screenshot shows a SQL query editor with the following query: `SELECT char_length('MySQL') as CHAR_LEN;`. The query is executed, and the results are displayed in a table. The table has one column named `CHAR_LEN` and one row with the value `5`.

CHAR_LEN
5

# String Function - Example - 2

**CONCAT\_WS():** This function is used to add two words or strings with a symbol as concatenating symbol

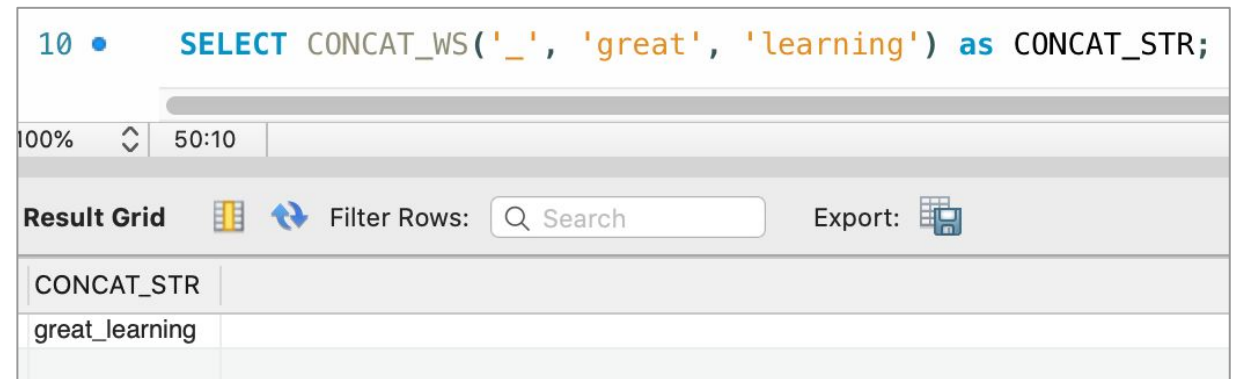
Syntax:

```
SELECT CONCAT_WS('_',  
'great', 'learning') as  
CONCAT_STR;
```

Output:

CONCAT\_STR

great\_learning



The screenshot shows a SQL query editor with a query window at the top containing the SQL statement: `SELECT CONCAT_WS('_', 'great', 'learning') as CONCAT_STR;`. Below the query window, there is a toolbar with icons for zooming (100%), a refresh icon, and a filter icon. To the right of the toolbar is a search bar labeled "Filter Rows:" and an "Export:" button with a download icon. Below the toolbar is a table titled "Result Grid" which contains the output of the query.

CONCAT_STR
great_learning

# String Function - Example - 3

LCASE(): This function is used to convert the given string into lower case

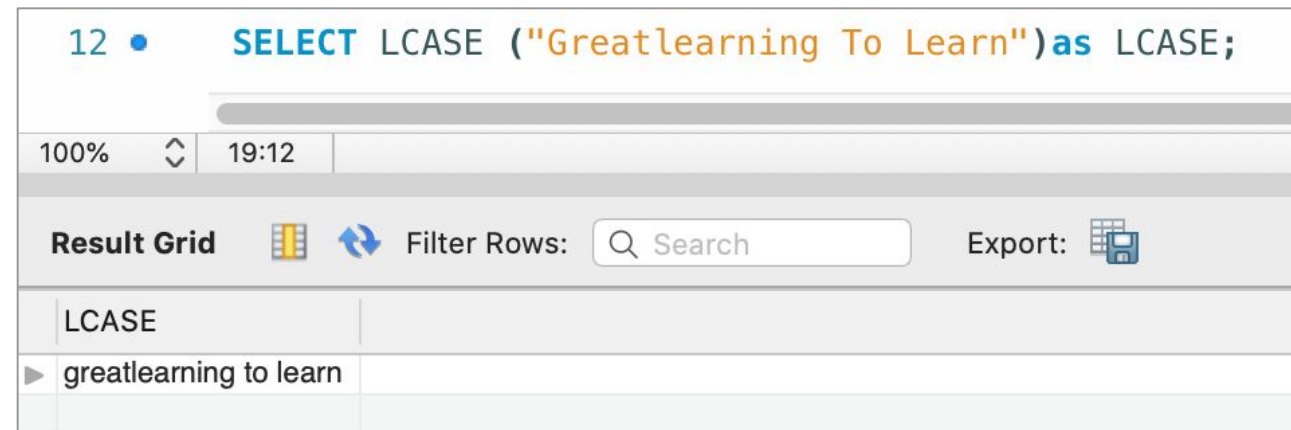
Syntax:

```
SELECT LCASE ("Greatlearning  
To Learn") as LCASE;
```

Output:

LCASE

greatlearning to learn



The screenshot shows a SQL query editor with the following query: `SELECT LCASE ("Greatlearning To Learn")as LCASE;`. Below the query, there is a result grid with the following data:

LCASE
greatlearning to learn

The interface includes a toolbar with icons for zooming (100%), undo, redo, and a search bar. The result grid also has an 'Export' button.

# String Function - Example - 4

**REPLACE():** This function replaces all occurrences of a substring within a string, with a new substring

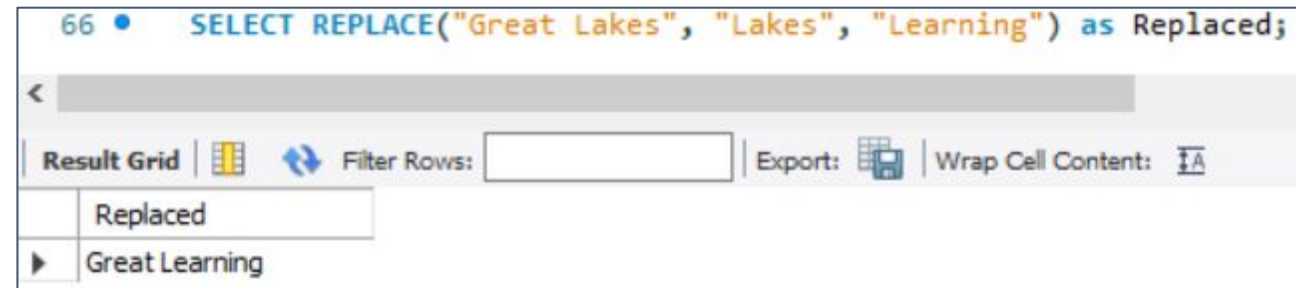
Syntax:

```
SELECT REPLACE("Great Lakes",  
"Lakes", "Learning") as  
Replaced;
```

Output:

Replaced

Great Learning



The screenshot shows a SQL query editor with a query window at the top containing the SQL statement: `SELECT REPLACE("Great Lakes", "Lakes", "Learning") as Replaced;`. Below the query window is a toolbar with options like "Result Grid", "Filter Rows", "Export", and "Wrap Cell Content". The "Result Grid" is active, displaying a table with two rows: a header row with the column name "Replaced" and a data row with the value "Great Learning".

Replaced
Great Learning

# String Function - Example - 5

**TRIM():** This function removes unwanted characters from a string

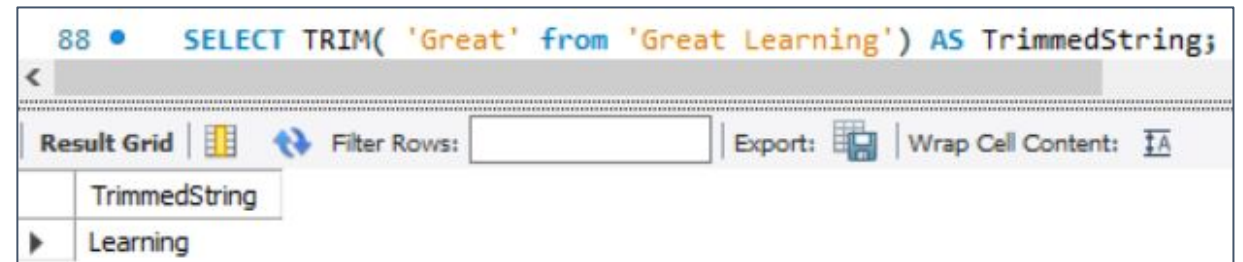
Syntax:

```
SELECT TRIM( 'Great' from  
'Great Learning') AS  
TrimmedString;
```

Output:

TrimmedString

Learning



The screenshot shows a SQL query editor with the following query: `SELECT TRIM( 'Great' from 'Great Learning') AS TrimmedString;`. Below the query editor, there is a toolbar with options like 'Result Grid', 'Filter Rows', 'Export', and 'Wrap Cell Content'. The 'Result Grid' is selected, and it displays a table with two columns: 'TrimmedString' and 'Learning'.

	TrimmedString
▶	Learning

# String Function - Example - 6

SUBSTR(): This function extracts a substring from a string

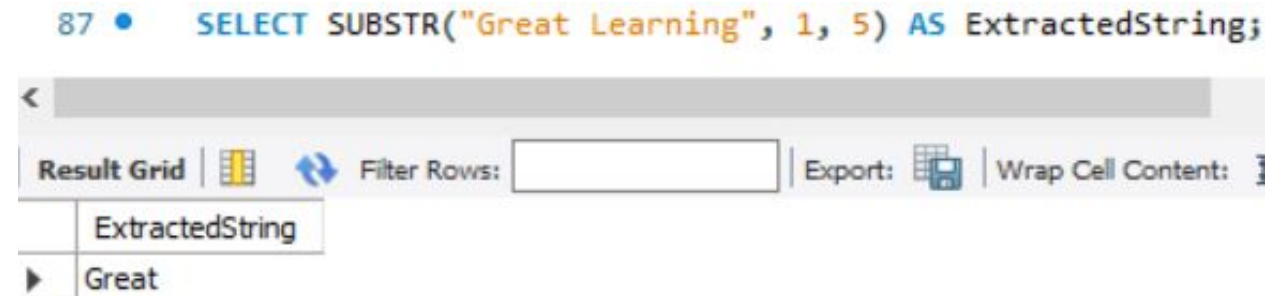
Syntax:

```
SELECT SUBSTR("Great  
Learning", 1, 5) AS  
ExtractedString;
```

Output:

ExtractedString

Great





# DATE Function



# Date Functions

- The format of the date in table must be matched with the input date in order to insert. In various scenarios instead of date, datetime is used

In MySQL the default date functions are:

Functions	Meaning
NOW	Returns the current date and time
CURDATE	Returns the current date
CURTIME	Returns the current time

# Date Functions

Functions	Meaning
DATE	Extracts the date part of a date or date/time expression
EXTRACT	Returns a single part of a date/time
DATE_ADD	Adds a specified time interval to a date
DATE_SUB	Subtracts a specified time interval from a date
DATEDIFF	Returns the number of days between two dates
DATE_FORMAT	Displays date/time data in different formats

# Date Function - Example - 1

**NOW():** Returns the current date and time.

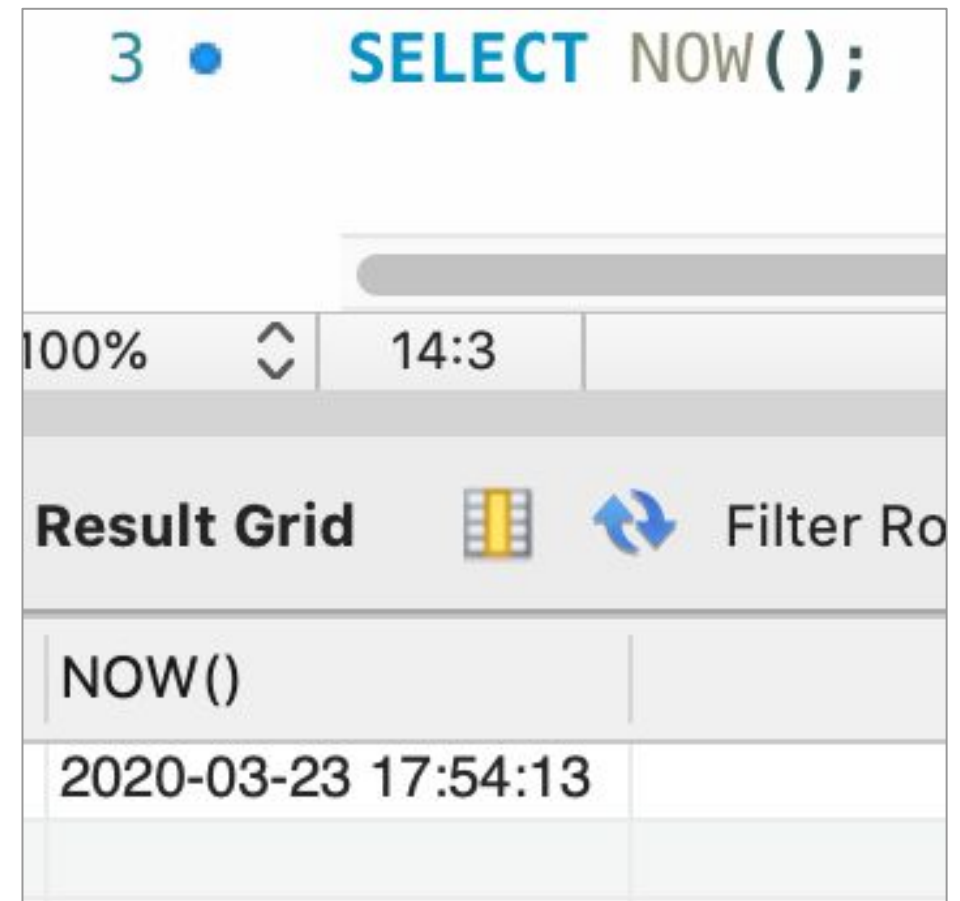
Syntax

```
SELECT NOW ( ) ;
```

Output:

NOW ( )

2020-03-23 17:54:13



The screenshot shows a database query interface. At the top, the SQL command `SELECT NOW();` is entered. Below the command bar, there are controls for zoom (100%) and a refresh button. The main area is labeled "Result Grid" and contains a single row of results. The first column is labeled "NOW()" and the second column is empty. The result value is "2020-03-23 17:54:13".

Result Grid	
NOW()	
2020-03-23 17:54:13	

# Date Function - Example - 2

**CURDATE()**: Returns the current date.

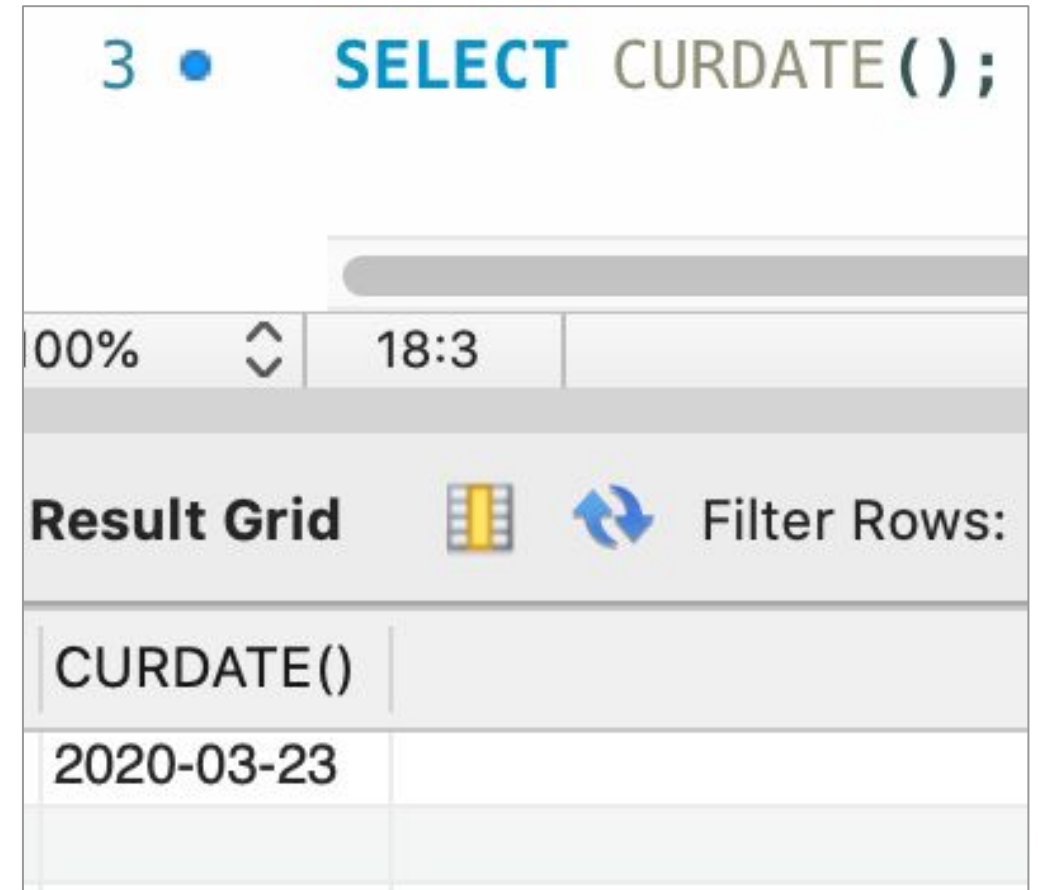
Syntax

```
SELECT CURDATE ( ) ;
```

Output:

**CURDATE** ( )

2020-03-23



The screenshot shows a SQL query editor with the query `SELECT CURDATE();` and a result grid below it. The result grid has a header row with `CURDATE()` and a data row with the value `2020-03-23`. The interface includes a query editor with a line number 3, a query bar, a result grid, and a filter rows button.

3	<code>SELECT CURDATE();</code>
00%	18:3
<b>Result Grid</b>	<b>Filter Rows:</b>
<code>CURDATE()</code>	
2020-03-23	

# Date Function - Example - 3

**CURTIME():** Returns the current time

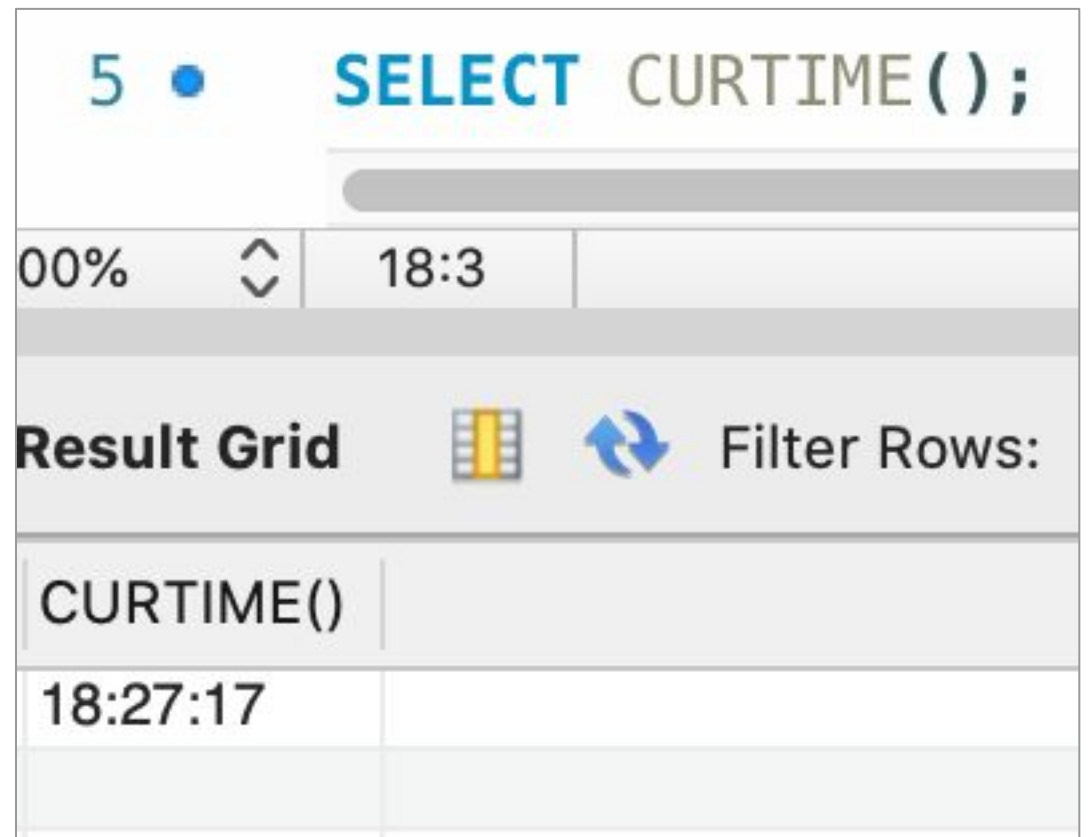
Syntax

```
SELECT CURTIME ( ) ;
```

Output:

```
CURTIME ( )
```

```
18:27:17
```



The screenshot shows a database query interface. At the top, a query editor displays the SQL statement `SELECT CURTIME();`. Below the editor, a progress bar indicates 00% completion, and a timer shows 18:3. The interface includes a 'Result Grid' section with a table icon and a 'Filter Rows' section with a refresh icon. The result grid contains a single row with the output of the CURTIME() function, which is 18:27:17.

CURTIME()
18:27:17



# BIN Function

# BIN Function - Syntax

- MySQL BIN() returns the corresponding string representation of the binary value of a BIGINT number

Syntax:

```
BIN (num1)
```

Argument:

Name	Description
num1	A number whose binary value is to be retrieved

# BIN Function - Example

- The following MySQL statement returns the string representation of 255, i.e. '11111111'

Syntax:

```
Select BIN(255) ;
```

Output:

	BIN(255)	
▶	11111111	





# CAST Function

# CAST Function - Syntax

- The CAST() function converts a value (of any type) into the specified datatype

Syntax:

```
CAST (value AS datatype)
```

Argument:

Parameter	Description
value	Required. The value to convert
datatype	Required. The datatype to convert to

# CAST Function - Example - 1

- Converting a value to a CHAR data type

Syntax:

```
SELECT CAST (150 AS CHAR) ;
```

Output:

Number of Records: 1	
<b>CAST(150 AS CHAR)</b>	
	150

# CAST Function - Example - 2

- Converting a value to a TIME datatype

Syntax:

```
SELECT CAST ("14:06:10" AS TIME) ;
```

Output:

Number of Records: 1
<b>CAST("14:06:10" AS TIME)</b>
14:06:10



# COALESCE Function

# COALESCE Function - Syntax

- The COALESCE() function returns the first non-null value in a list

Syntax:

```
COALESCE(val1, val2, ..., val_n)
```

Parameter Values:

Parameter	Description
<i>val1</i> , <i>val2</i> , <i>val_n</i>	Required. The values to test

# COALESCE Function - Example

- Return the first non-null value in a list

```
SELECT COALESCE(NULL, 1, 2, 'greatlearning.in');
```

Output:

Number of Records: 1
<b>COALESCE(NULL, 1, 2, 'greatlearning.in')</b>
1

The first value after the NULL value is '1', so the output is 1



# Sorting Query Result



# Sorting Query Results

Consider the following table for the study:

UserID	Name	City	Salary
1	Atanu Chakraborty	Kolkata	15000
2	Hriday Bharadwaj	Lucknow	27000
3	Birendra Maity	Hyderabad	68000
4	Raj Patil	Bangalore	71000
22	Bikram Ganguly	Kolkata	51000
31	Tapan Chatterjee	Kolkata	25000
33	Jai Prakash Bhatt	Jaipur	81000
34	Keith Kosta	Mumbai	77000
35	Mark Tailor	Bangalore	52000
36	Christopher Swan	Null	98000
39	Paul Grant	Mumbai	67000

# Sorting Query Results

- The *order by* clause causes the tuples in the result of a query to appear in sorted order
- Let us write a query which will show the users according to the ascending order of their salary.

```
SELECT UserID, Name, Salary FROM tblUser ORDER BY Salary
```

Output:

UserID	Name	Salary
1	Atanu Chakraborty	15000
31	Tapan Chatterjee	25000
2	Hriday Bharadwaj	27000
22	Bikram Ganguly	51000
35	Mark Tailor	52000
39	Paul Grant	67000
3	Birendra Maity	68000
4	Raj Patil	71000
34	Keith Kosta	77000
33	Jai Prakash Bhatt	81000
36	Christophar Swan	98000

By default, the order by clause lists items in ascending order.

# Sorting Query Results

- To specify the sort order we used: desc for descending order or asc for ascending order
- Now let us see an example to show the users according to the descending order of their salary

```
SELECT UserID, Name, Salary FROM tblUser ORDER BY  
Salary desc
```

Output:

UserID	Name	Salary
36	Christophar Swan	98000
33	Jai Prakash Bhatt	81000
34	Keith Kosta	77000
4	Raj Patil	71000
3	Birendra Maity	68000
39	Paul Grant	67000
35	Mark Tailor	52000
22	Bikram Ganguly	51000
2	Hriday Bharadwaj	27000
31	Tapan Chatterjee	25000
1	Atanu Chakraborty	15000

# Sorting Query Results

- Furthermore ordering can be performed on multiple attributes.
- Suppose we wish to order the entire tblUser relation in ascending order of City and then descending order of amount. Then we express this query in SQL as follows.

```
SELECT UserID, Name, Salary FROM tblUser ORDER BY City  
desc, Salary asc
```

Output:

UserID	Name	City	Salary
39	Paul Grant	Mumbai	67000
34	Keith Kosta	Mumbai	77000
2	Hriday Bharadwaj	Lucknow	27000
1	Atanu Chakraborty	Kolkata	15000
31	Tapan Chatterjee	Kolkata	25000
22	Bikram Ganguly	Kolkata	51000
33	Jai Prakash Bhatt	Jaipur	81000
3	Birendra Maity	Hyderabad	68000
35	Mark Tailor	Bangalore	52000
4	Raj Patil	Bangalore	71000
36	Christophar Swan	Null	98000



Thank You