

# SQL (2)

---

# Contents


- INSERT/ UPDATE/ DELETE
- Stored procedure
- Primary key and Foreign key
- MySQL shell
- Data encryption
- View
- Trigger
- SQL - Exist
- Query Execution Plan
- Types of Databases
- Normalizations

INSERT/ UPDATE/ DELETE

# INSERT

- Simple Syntax: **INSERT INTO** "tablename" ("column1", ...) **VALUES** ("value1", ...);

```
INSERT INTO Store_Information (Store_Name, Sales, Txn_Date)
VALUES ('Los Angeles', 900, '1999-01-10');
```



Store_Name	Sales	Txn_Date
Los Angeles	1500	1999-01-05
San Diego	250	1999-01-07
San Francisco	300	1999-01-08
Boston	700	1999-01-08

Store_Name	Sales	Txn_Date
Los Angeles	1500	1999-01-05
San Diego	250	1999-01-07
San Francisco	300	1999-01-08
Boston	700	1999-01-08
Los Angeles	900	1999-01-10

# Exercise: more than one rows inserted

```
INSERT INTO Store_Information_1
    (Store_Name, Sales, Txn_Date)
VALUES ('Lowell', 800, '1998-01-10'),
    ('Lowell', 700, '1998-01-11');
```



Store_Name	Sales	Txn_Date
Los Angeles	1500	1999-01-05
San Diego	250	1999-01-07
Los Angeles	300	1999-01-08
Boston	700	1999-01-08

Store_Name	Sales	Txn_Date
Los Angeles	1500	1999-01-05
San Diego	250	1999-01-07
Los Angeles	300	1999-01-08
Boston	700	1999-01-08
Lowell	800	1998-01-10
Lowell	700	1998-01-11

# INSERT Multiple Rows

```
INSERT INTO Store_Information (Store_Name, Sales, Txn_Date)
  SELECT Store_Name, Sales, Txn_Date
  FROM store_Information_1
  WHERE Year(Txn_Date) = 1998;
```

store\_information

Store_Name	Sales	Txn_Date
Los Angeles	1500	1999-01-05
San Diego	250	1999-01-07
San Francisco	300	1999-01-08
Boston	700	1999-01-08
Los Angeles	900	1999-01-10



Store_Name	Sales	Txn_Date
Los Angeles	1500	1999-01-05
San Diego	250	1999-01-07
San Francisco	300	1999-01-08
Boston	700	1999-01-08
Los Angeles	900	1999-01-10
Lowell	800	1998-01-10
Lowell	700	1998-01-11

# UPDATE

- Simple syntax: **UPDATE** "TableName"  
**SET** "column1" = [new value]  
**WHERE** "condition";

```
UPDATE Store_Information  
SET Sales = 555  
WHERE Store_Name = 'Los Angeles'
```

Store_Name	Sales	Txn_Date
Los Angeles	1500	1999-01-05
San Diego	250	1999-01-07
San Francisco	300	1999-01-08
Boston	700	1999-01-08
Los Angeles	900	1999-01-10
Lowell	800	1998-01-10
Lowell	700	1998-01-11



Store_Name	Sales	Txn_Date
Los Angeles	555	1999-01-05
San Diego	250	1999-01-07
San Francisco	300	1999-01-08
Boston	700	1999-01-08
Los Angeles	555	1999-01-10
Lowell	800	1998-01-10
Lowell	700	1998-01-11

# Exercise

```
UPDATE Store_Information
SET Sales = 777,
    store_name = 'SunnyVale'
WHERE Store_Name = 'Los Angeles'
```



Store_Name	Sales	Txn_Date
Los Angeles	555	1999-01-05
San Diego	250	1999-01-07
San Francisco	300	1999-01-08
Boston	700	1999-01-08
Los Angeles	555	1999-01-10
Lowell	800	1998-01-10
Lowell	700	1998-01-11

Store_Name	Sales	Txn_Date
SunnyVale	777	1999-01-05
San Diego	250	1999-01-07
San Francisco	300	1999-01-08
Boston	700	1999-01-08
SunnyVale	777	1999-01-10
Lowell	800	1998-01-10
Lowell	700	1998-01-11



# DELETE

- Simple syntax: **DELETE FROM** "Tablename"  
**WHERE** "condition";

```
DELETE FROM Store_Information  
WHERE sales=777;
```

Store_Name	Sales	Txn_Date
SunnyVale	777	1999-01-05
San Diego	250	1999-01-07
San Francisco	300	1999-01-08
Boston	700	1999-01-08
SunnyVale	777	1999-01-10
Lowell	800	1998-01-10
Lowell	700	1998-01-11

Store_Name	Sales	Txn_Date
San Diego	250	1999-01-07
San Francisco	300	1999-01-08
Boston	700	1999-01-08
Lowell	800	1998-01-10
Lowell	700	1998-01-11

# Stored Procedure

# Stored Procedure

- What?
  - A stored procedure is a set of Structured Query Language (SQL) statements with an assigned name, which are stored in a relational database management system (RDBMS) as a group, so it can be reused and shared by multiple programs, ex. `php`, `PL/SQL`, `Java`, `Python`, and so on.
- Purpose?
  - The main purpose of stored procedures to `hide` direct SQL queries from the code and `improve performance` of database operations such as `select`, `update`, `insert` and `delete` data.
- Why?
  - A stored procedure is a set of prepared SQL code that you can save, so the code `can be reused` over and over again.
  - a stored procedure allows them to be executed with a single call.

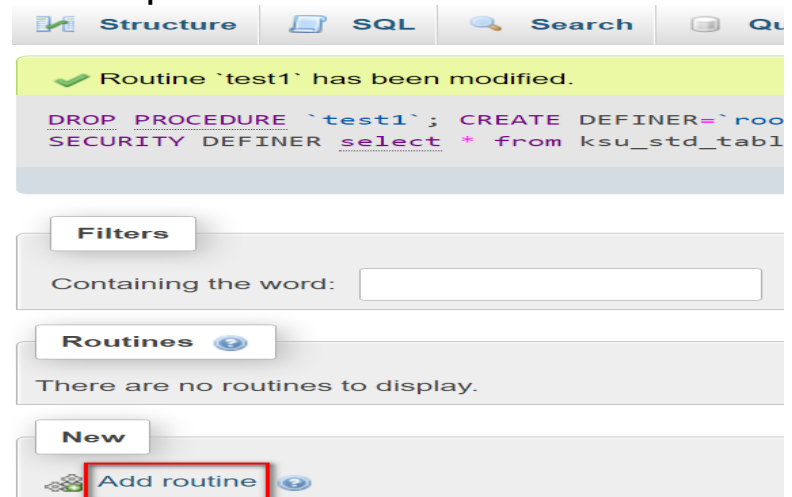
# Stored Procedure Creation & Execution

- Summarize the ideas:
  - Open PHP My Admin and select the database to create stored procedure
  - Go to Routines menu & Click on Add routine.
  - By Clicking on Add Routine Link, PHPMyAdmin will open Pop-up.
  - Follow the steps and create stored procedure. Create stored procedure to get data from users table (Without Parameters).

p.s. The name “routine” is the stored procedure we are talking about.

- Creation without parameters

- Step 1: click on the Database you want to operate it
- Step 2: click on the routines (預存程序) tab on the top bar
- Step 3: click on the add routine



- Step 4: follow the steps and fill out the form to finish the settings.

The 'Add routine' dialog box is shown with the 'Details' tab selected. It contains the following fields and controls:

- Routine name:** A text input field with a red rectangle around it.
- Type:** A dropdown menu currently set to 'PROCEDURE'.
- Parameters:** A table with columns 'Direction', 'Name', 'Type', 'Length/Values', and 'Options'. The first row has 'IN' in the 'Direction' column, an empty 'Name' field, 'IN' in the 'Type' column, an empty 'Length/Values' field, and a 'Drop' button. A red rectangle is around the 'Name' field.
- Add parameter:** A button below the parameters table.
- Footer:** 'Go' and 'Close' buttons.

- Step 5: click on the execute tab for the stored procedure invocation



✓ Routine 'test1' has been created.

```
CREATE PROCEDURE `test1`() NOT DETERMINISTIC CONTAINS SQL SQL SECURITY DEFINER select * from ksu_std_tabl
```

[ Edit inline ] [ Edit ] [ Create PHP code ]

Filters

Containing the word:

Routines

Name	Action	Type	Returns
<input type="checkbox"/> test1	<a href="#">Edit</a> <a href="#">Execute</a> <a href="#">Export</a> <a href="#">Drop</a>	PROCEDURE	

New

Add routine

```
CALL `test1`();
```

Execution results of routine 'test1'

ksu_std_id	ksu_std_name	ksu_std_age	ksu_std_department	ksu_std_signin	ksu_std_grade
2323E1	John1	33	QQ	2020-04-01	100
4040w1	John1	22	CS	2020-04-01	100
D01	John Sieg	22	CS	2019-12-05	100
D02	John Sieg	44	IE	2019-12-04	99
IE01	Canning	33	IE	2019-11-12	100
IE02	Mike Fire	32	IE	2019-12-11	77
IE03	Mary Wee	34	IM	2019-12-02	80
IM01	WuBer Eat	22	IM	2019-11-12	33
IM02	Foot Penny	27	CS	2019-10-10	44
IM05	John Sieg	24	CS	2019-12-16	55
ss	1John	22	CS	2020-04-01	100
33	33	0		0000-00-00	100
9898	Mike	0		0000-00-00	100
777	Taiwan	0		0000-00-00	100
s	sss	0		0000-00-00	100
ddd	dddd	0		0000-00-00	100

# Export the store procedure

9898	Mike	0
777	Taiwan	0
s	sss	0
ddd	dddd	0

Filters

Containing the word:

Routines

	Name	Action
<input type="checkbox"/>	test1	Edit  Execute  Export

New

Add routine

Export of routine `test1`

```
1 DELIMITER $$
2 CREATE DEFINER=`root`@`localhost` PROCEDURE `test1`()
3 select * from ksu_std_table$$
4 DELIMITER ;
```

Close

# • Creation with parameters

Routine name

test2

Type

PROCEDURE

Parameters

Direction	Name	Type	Length/Values	Options
IN	p1	IN		

Add parameter

Definition

```
1 IF p1 < 60 THEN
2     SELECT *
3     from ksu_std_table
4     where ksu_std_grade < 60;
5 ELSE
6     SELECT *
7     from ksu_std_table
8     where ksu_std_grade >= 60;
9 END IF
```

Execute routine `test2`

Routine parameters

Name	Type	Function	Value
p1	INT		100

Go Close

Execution results of routine `test2`

ksu_std_id	ksu_std_name	ksu_std_age	ksu_std_department	ksu_std_signin	ksu_std_grade
2323E1	John1	33	QQ	2020-04-01	100
4040w1	John1	22	CS	2020-04-01	100
D01	John Sieg	22	CS	2019-12-05	100
D02	John Sieg	44	IE	2019-12-04	99
IE01	Canning	33	IE	2019-11-12	100
IE02	Mike Fire	32	IE	2019-12-11	77
IE03	Mary Wee	34	IM	2019-12-02	80
ss	1John	22	CS	2020-04-01	100
33	33	0		0000-00-00	100
9898	Mike	0		0000-00-00	100
777	Taiwan	0		0000-00-00	100
s	sss	0		0000-00-00	100
ddd	dddd	0		0000-00-00	100

SET @p0='100'; CALL `test2` (@p0);



## Export of routine `test2`



```
1 DELIMITER $$
2 CREATE DEFINER=`root`@`localhost` PROCEDURE `test2`(IN
  `p1` INT)
3 IF p1 < 60 THEN
4     SELECT *
5     from ksu_std_table
6     where ksu_std_grade < 60;
7 ELSE
8     SELECT *
9     from ksu_std_table
10    where ksu_std_grade >= 60;
11 END IF $$
12 DELIMITER ;
```

//

Close

# Example 1

**Edit**

Routine name: test4

Type: PROCEDURE

Direction	Name	Type	Length/Values	Options
IN	p1	IN		

**Parameters**

**Add parameter**

**Definition**

```
1 BEGIN
2 DECLARE p1by INT;
3 SET p1by=p1 *0.6;
4
5 IF p1by < 6 THEN
6     SELECT *
7     from ksu_std_table
8     where ksu_std_grade < 60;
9 ELSE
10    SELECT *
11    from ksu_std_table
12    where ksu_std_grade >= 60;
13 END IF;
14 END
```

**Execute routine `test4`**

**Routine parameters**

Name	Type	Function	Value
p1	INT		100

**Go** **Close**

```
SET @p0='100'; CALL `test4` (@p0);
```

**Execution results of routine `test4`**

ksu_std_id	ksu_std_name	ksu_std_age	ksu_std_department	ksu_std_signin	ksu_std_grade
2323E1	John1	33	QQ	2020-04-01	100
4040w1	John1	22	CS	2020-04-01	100
D01	John Sieg	22	CS	2019-12-05	100
D02	John Sieg	44	IE	2019-12-04	99
IE01	Canning	33	IE	2019-11-12	100
IE02	Mike Fire	32	IE	2019-12-11	77
IE03	Mary Wee	34	IM	2019-12-02	80
ss	1John	22	CS	2020-04-01	100
33	33	0		0000-00-00	100
9898	Mike	0		0000-00-00	100
777	Taiwan	0		0000-00-00	100
s	sss	0		0000-00-00	100
ddd	dddd	0		0000-00-00	100

# Example 2

**Edit**

**Routine name** test3

**Type** PROCEDURE

**Parameters**

	Direction	Name	Type	Length/Values	Options
↑	IN	weight	INT		
↑	IN	height	INT		

Add parameter

**Definition**

```
1 BEGIN
2 DECLARE BMI int;
3 SET BMI=weight/height*height;
4
5 IF BMI>24 THEN
6 SELECT CONCAT('Your BMI is ',BMI,' over heavy!');
7 ELSEIF BMI>18 THEN
8 SELECT CONCAT('Your BMI is ',BMI,' Normal. ');
9 ELSE
10 SELECT CONCAT('Your BMI is ',BMI,' skinny!');
11 END IF;
12 END
```

Execute routine `test3`

**Routine parameters**

Name	Type	Function	Value
weight	INT		70
height	INT		175

Go Close

Execution results of routine `test3`

**CONCAT('Your BMI is ',BMI,' over heavy!')**  
Your BMI is 70 over heavy!

Go Close

# Example 3

Details

Routine name

test5

Type

PROCEDURE

Parameters

Direction

Name

Type

Length/Values

Options

Add parameter

Definition

```
1 BEGIN
2
3 SET AUTOCOMMIT=0;
4 drop table if exists test_table1, test_table2;
5
6 CREATE TABLE test_table1 (id int);
7 INSERT INTO test_table1 VALUES (100);
8 COMMIT;
9
10 CREATE TABLE test_table2 (id int);
11 INSERT INTO test_table2 VALUES (100);
12 ROLLBACK;
13 END
```

Go

Close

SET autocommit=0; After disabling autocommit mode by setting the autocommit variable to zero, **changes to transaction-safe tables** (such as those for InnoDB or NDB ) are not made permanent immediately. You must use COMMIT to store your changes to disk or ROLLBACK to ignore the changes.

✓ Showing rows 0 - 0 (1 total, Query t

```
SELECT * FROM `test_table1`
```

☐ Profiling [ Edit inline ] [ Edit ] [ Explai

☐ Show all | Number of rows:

+ Options

id

100

✓ MySQL returned an empty result

```
SELECT * FROM `test_table2`
```

☐ Profiling [ Edit inline ] [ Edit ] [ Ex

id

# Settings

- A procedure or function is considered “**deterministic**” if it always produces **the same result for the same input parameters**, and “not deterministic” otherwise.
  - A routine that contains the NOW() function (or its synonyms) or RAND() is non-deterministic, but it might still be replication-safe.
- Several characteristics provide information about the nature of data use by the routine. In MySQL, these characteristics are **advisory only**. The server does not use them to constrain what kinds of statements a routine will be allowed to execute.
  - **CONTAINS SQL** indicates that the routine does not contain statements that read or write data. This is the default if none of these characteristics is given explicitly. Examples of such statements are **SET @x = 1** or **DO RELEASE\_LOCK('abc')**, which execute but neither read nor write data.
  - **NO SQL** indicates that the routine contains no SQL statements.
  - **READS SQL DATA** indicates that the routine contains statements that read data (for example, **SELECT**), but not statements that write data.
  - **MODIFIES SQL DATA** indicates that the routine contains statements that may write data (for example, **INSERT** or **DELETE**).

# Commit **v.s.** rollback

- The most important aspect of a database is **the ability to store data and the ability to manipulate data**. COMMIT and ROLLBACK are two such keywords which are used in order store and **revert the process of data storage**.
- COMMIT and ROLLBACK are performed on transactions.
- A transaction is **the smallest unit of work** that is performed against a database. Its a sequence of instructions in a logical order. A transaction can be performed manually by a **programmer, DBA or it can be triggered using an automated program**.
- COMMIT is the SQL command that is used for storing changes performed by a transaction. **When a COMMIT command is issued it saves all the changes since last COMMIT or ROLLBACK**.

# Commit **v.s.** rollback

- Syntax for SQL Commit

**COMMIT;**

- SQL Commit Example

- CASE 1

**Customer:-**

CUSTOMER ID	CUSTOMER NAME	STATE	COUNTRY
1	Akash	Delhi	India
2	Amit	Hyderabad	India
3	Jason	California	USA
4	John	Texas	USA

intend removing  
this tuple if **COMMIT**  
is not published!  
Nothing has been  
removed here!

Now let us delete one row from the above table where State is "Texas".

```
DELETE from Customer where State = 'Texas';
```

SQL Delete without Commit

Post the DELETE command if we will not publish COMMIT, and if the session is closed then the change that is made due to the DELETE command will be lost.

# Commit **v.s.** rollback

- Syntax for SQL Commit

**COMMIT;**

- SQL Commit Example

- CASE 2

```
DELETE from Customer where State = 'Texas';  
COMMIT;
```

SQL Commit Execution

Truly remove  
this tuple if COMMIT  
is published!

Using the above-mentioned command sequence will ensure that the change post DELETE command will be saved successfully.

## Output After Commit

CUSTOMER ID	CUSTOMER NAME	STATE	COUNTRY
1	Akash	Delhi	India
2	Amit	Hyderabad	India
3	Jason	California	USA



# Commit **v.s.** rollback


- **Syntax for SQL Commit :** ROLLBACK is the SQL command that is used for **reverting changes performed by a transaction**. When a ROLLBACK command is issued it reverts all the changes since **last COMMIT or ROLLBACK**.

**ROLLBACK;**

- **SQL Rollback Example**

Post the DELETE command if we publish ROLLBACK it will revert the change that is performed due to the delete command.

```
DELETE from Customer where State = 'Texas';  
ROLLBACK;
```



intend removing  
this tuple if ROLLBACK  
is published! Nothing has been  
removed here! As compared to the idea on page  
23, this way is safer to rollback your data!

# Stored procedure **v.s.** function

- Basic Differences between Stored Procedure and Function in SQL Server. The function must return **a value** but in Stored Procedure it is optional. Even a procedure can return zero or n values. Functions can have only **input parameters** for it whereas Procedures can have **input or output parameters**.

# Primary key and Foreign key

# Primary key, Foreign key...

- Primary key
  - A primary key is a column -- or a group of columns -- in a table that uniquely identifies the rows in that table.
  - Should be meaningful to a table
  - primary key **cannot be null**
  - Only one primary key **allowed** in a table
- Foreign keys
  - Foreign keys are the columns of a table that points to the primary key of another table (ie The another table is called as a **parent table**. ).
- Candidate keys
  - **The minimal set of attributes** that can uniquely identify a tuple is known as a candidate key.
- Super keys
  - is a single key or a group of multiple keys that can uniquely identify tuples in a table. Super keys can contain **redundant attributes** that **might not** be important for identifying tuples.
  - Candidate keys are a subset of Super keys.

Super key  $\supset$  candidate key  $\supset$  primary key

# Example 1

- Consider that *Id* attribute is **unique** to every employee's tuples.
- In that case, we can say that the *Id* attribute can uniquely identify the tuples of this table. So, *Id* is a **Super key** of this table. Note that we can have other Super Keys too in this table.
- For instance – (*Id*, *Name*), (*Id*, *Email*), (*Id*, *Name*, *Email*), etc. can all be Super keys as they can all uniquely identify the tuples of the table. This is so because of the presence of the *Id* attribute which is able to uniquely identify the tuples. The other attributes in the keys are unnecessary. Nevertheless, they can still identify tuples and **mean nothing**.
- **Super keys are for logical view**. They never happens on Memory implementation since some of them are not meaningful except they are also primary key or foreign key or unique index.

Id	Name	Gender	City	Email	Dep_Id
1	Ajay	M	Delhi	ajay@gmail.com	1
2	Vijay	M	Mumbai	vijay@gmail.com	2
3	Radhika	F	Bhopal	radhika@gmail.com	1
4	Shikha	F	Jaipur	shikha@gmail.com	2
5	Hritik	M	Jaipur	hritik@gmail.com	2

5 rows in set (0.00 sec)

employee

# Example 2

- A **Candidate key** is a subset of Super keys and is devoid of any unnecessary attributes that are not important for uniquely identifying tuples.
- The value for the Candidate key is unique and **non-null** for all tuples. And **every table has to have at least one Candidate key**.
- For example, in the example, both *Id* and *Email* can act as a Candidate for the table as they contain **unique** and **non-null values**. However, not all of candidate keys can be implemented on Memory. Namely, some of them are for logical view only.

Id	Name	Gender	City	Email	Dep_Id
1	Ajay	M	Delhi	ajay@gmail.com	1
2	Vijay	M	Mumbai	vijay@gmail.com	2
3	Radhika	F	Bhopal	radhika@gmail.com	1
4	Shikha	F	Jaipur	shikha@gmail.com	2
5	Hritik	M	Jaipur	hritik@gmail.com	2

5 rows in set (0.00 sec)

employee

# Example 3

- **Secondary keys** are those candidate keys which are not the Primary key. Namely, **candidate keys = secondary keys + one primary key**
- since we have chosen **Id** as the Primary key, the other Candidate Key (**Email**), becomes the **Secondary key (or Alternate key)** for the table.

Id	Name	Gender	City	Email	Dep_Id
1	Ajay	M	Delhi	ajay@gmail.com	1
2	Vijay	M	Mumbai	vijay@gmail.com	2
3	Radhika	F	Bhopal	radhika@gmail.com	1
4	Shikha	F	Jaipur	shikha@gmail.com	2
5	Hritik	M	Jaipur	hritik@gmail.com	2

5 rows in set (0.00 sec)

employee

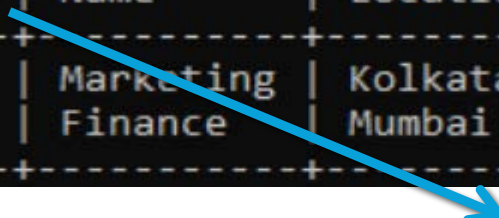


- ✓ candidate key = **unique constraint** + **non-null values constraint**
- ✓ **candidate keys = secondary keys + one primary key**
- ✓ Super key  $\supset$  candidate key  $\supset$  primary key
- ✓ Super key  $\supset$  candidate key  $\supset$  secondary key

# Example 4

- **Foreign key** is an attribute which is a **Primary key** in its parent table, but is included as an attribute in another host table. The referenced table is called the **parent table** while the table with the foreign key is called the **child table (or called host table)**.
- For example, in addition to the *Employee* table containing the personal details of the employees, we might have another table *Department* containing information related to the department of the employee.

Employee Id



Id	Name	Location
1	Marketing	Kolkata
2	Finance	Mumbai

**Department  
(parent table)**

Id	Name	Gender	City	Email	Dep_Id
1	Ajay	M	Delhi	ajay@gmail.com	1
2	Vijay	M	Mumbai	vijay@gmail.com	2
3	Radhika	F	Bhopal	radhika@gmail.com	1
4	Shikha	F	Jaipur	shikha@gmail.com	2
5	Hritik	M	Jaipur	hritik@gmail.com	2

**Employee (child table)**

The Primary key in this table is the Department *Id*. We can add this attribute to the Employee by making it the **Foreign key** in the table.

5 rows in set (0.00 sec)



- Here, *Dep\_Id* is now the **Foreign Key** in table Employee while it is a Primary Key in the Department table.
- The **Foreign key** allows you to create a relationship between two tables in a database. Each of these tables describes data related to a particular field (employee and department here). Using the Foreign key, we can easily retrieve data from both the tables.
- *Note: To operate on Foreign keys, you need to know about Joins which you can find out in detail in Ch72.pdf.*

Id	Name	Location
1	Marketing	Kolkata
2	Finance	Mumbai

department

Id	Name	Gender	City	Email	Dep_Id
1	Ajay	M	Delhi	ajay@gmail.com	1
2	Vijay	M	Mumbai	vijay@gmail.com	2
3	Radhika	F	Bhopal	radhika@gmail.com	1
4	Shikha	F	Jaipur	shikha@gmail.com	2
5	Hritik	M	Jaipur	hritik@gmail.com	2

5 rows in set (0.00 sec)

employee

The Primary key in this table is the Department *Id*. We can add this attribute to the Employee by making it the **Foreign key** in the table.

# Why use Foreign keys?

- Using **Foreign keys** makes it easier to require the data in the database when joining more than two tables.

# What are Composite keys?

- A Composite key is a **Candidate key** or **Primary key** that consists of more than one column.
- Sometimes it is possible that no single attribute will have the property to uniquely identify tuples in a table. In such cases, we can use a group of attributes to guarantee uniqueness. Combining these attributes will uniquely identify tuples in the table.
- Here, **neither** of the **attributes** contains unique values to identify the tuples. Therefore, we can combine **two or more attributes to create a key** that can uniquely identify the tuples. For example, we can group **Transaction\_Id** and **Product\_Id** to create a key that can uniquely identify the tuples. These are called composite keys.

Transaction_Id	Product_Id	Customer_Id	Product	Quantity
A1001	P1005	C9001	Smartphone	1
A1001	P2010	C9001	Screen guard	1
A1002	P2013	C9003	Smartwatch	1
A1003	P2010	C9010	Screen guard	2

# Key differences between SQL Keys

- **SQL keys can either be a single column or a group of columns.**
- **Super key** is a single key or a group of multiple keys that can uniquely identify tuples in a table.
- **Super keys** can contain **redundant attributes** that might not be important for identifying tuples.
- **Candidate keys** are a subset of **Super keys**. They contain only those attributes which are required to uniquely identify tuples.
- All **Candidate keys** are **Super keys**. But the vice-versa is not true.
- **Primary key** values should be **unique** and **non-null**.
- There can be multiple **Super keys** and **Candidate keys** in a table, but there can be only one **Primary key** in a table.
- **Secondary keys** are those **Candidate keys** that were not chosen to be the Primary key of the table.
- **Composite key** is a **Candidate key** that consists of more than one attribute.
- **Foreign key** is an attribute which is a **Primary key** in its parent table but is included as an attribute in the child table.
- **Foreign keys** **may** accept non-unique and null values for performance issue when its parent table has a composite key.

# MySQL Shell

# MySQL shell for fun!

- Go to bin directory + type “.\mysql.exe -u root”

```
C:\xampp\mysql\bin>mysql.exe -u root
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 58
Server version: 10.4.19-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

```
MariaDB [(none)]> ?
```

General information about MariaDB can be found at  
<http://mariadb.org>

List of all client commands:

Note that all text commands must be first on line and end with ';'.

```
?      (\?) Synonym for 'help'.
clear  (\c) Clear the current input statement.
connect (\r) Reconnect to the server. Optional arguments are db and host.
delimiter (\d) Set statement delimiter.
ego     (\G) Send command to MariaDB server, display result vertically.
exit    (\q) Exit mysql. Same as quit.
go      (\g) Send command to MariaDB server.
help    (\h) Display this help.
notee   (\t) Don't write into outfile.
print   (\p) Print current command.
prompt  (\R) Change your mysql prompt.
quit    (\q) Quit mysql.
rehash  (\#) Rebuild completion hash.
source  (\.) Execute an SQL script file. Takes a file name as an argument.
status  (\s) Get status information from the server.
tee      (\T) Set outfile [to_outfile]. Append everything into given outfile.
use     (\u) Use another database. Takes database name as argument.
charset (\C) Switch to another charset. Might be needed for processing binlog with multi-byte charsets.
warnings (\W) Show warnings after every statement.
nowarning (\w) Don't show warnings after every statement.
```

For server side help, type 'help contents'

```
MariaDB [(none)]>
```

```
MariaDB [(none)]> show databases
-> ;
```

```
+-----+
| Database |
+-----+
| information_schema |
| ksu_database      |
| ksu_db0914        |
| ksu_db1004-w4     |
| mysql             |
| performance_schema |
| phpmyadmin        |
| test              |
+-----+
8 rows in set (0.001 sec)
```

```
MariaDB [(none)]> show ksu_b0914;
```

ERROR 1064 (42000): You have an error in your SQL syntax; check the

```
MariaDB [(none)]> show databases;
```

```
+-----+
| Database |
+-----+
| information_schema |
| ksu_database      |
| ksu_db0914        |
| ksu_db1004-w4     |
| mysql             |
| performance_schema |
| phpmyadmin        |
| test              |
+-----+
8 rows in set (0.001 sec)
```

```
MariaDB [(none)]> use ksu_db0914;
```

Database changed

```
MariaDB [ksu_db0914]>
```

Database changed  
MariaDB [ksu\_db0914]> show tables;

```
+-----+
| Tables_in_ksu_db0914 |
+-----+
| advisor_detail |
| city_detail |
| customer |
| dept_detail |
| ksu_std_table |
| product_detail |
| store_information |
| store_information_l |
| student_detail |
| test_table01 |
| tracing_log |
| 議題意見討論版 |
+-----+
12 rows in set (0.002 sec)
```

MariaDB [ksu\_db0914]> select \* from customer;  
Empty set (0.011 sec)

MariaDB [ksu\_db0914]> select \* from ksu\_std\_table;

```
+-----+-----+-----+-----+-----+-----+
| ksu_std_id | ksu_std_name | ksu_std_age | ksu_std_department | ksu_std_signin | ksu_std_grade |
+-----+-----+-----+-----+-----+-----+
| 2323E1 | John1 | 33 | QQ | 2020-04-01 | 100 |
| 4040w1 | John1 | 22 | CS | 2020-04-01 | 100 |
| D01 | John Sieg | 22 | CS | 2019-12-05 | 100 |
| D02 | John Sieg | 44 | IE | 2019-12-04 | 99 |
| 1E01 | Canning | 33 | IE | 2019-11-12 | 100 |
| 1E02 | Mike Fire | 32 | IE | 2019-12-11 | 77 |
| 1E03 | Mary Wee | 34 | IM | 2019-12-02 | 80 |
| 1M01 | WuBer Eat | 22 | IM | 2019-11-12 | 33 |
| 1M02 | Foot Penny | 27 | CS | 2019-10-10 | 44 |
| 1M05 | John Sieg | 24 | CS | 2019-12-16 | 55 |
| ss | lJohn | 22 | CS | 2020-04-01 | 100 |
| 33 | 33 | 0 | | 0000-00-00 | 100 |
| 9898 | Mike | 0 | | 0000-00-00 | 100 |
| 777 | Taiwan | 0 | | 0000-00-00 | 100 |
| s | sss | 0 | | 0000-00-00 | 100 |
| ddd | dddd | 0 | | 0000-00-00 | 100 |
+-----+-----+-----+-----+-----+-----+
16 rows in set (0.008 sec)
```



# To leave the paradise

```
MariaDB [ksu_db0914]> quit  
Bye  
PS C:\xampp\mysql\bin>
```

# Data encryption

# Functions of Encryption and Decryption

- Two way and one way are the methods for functions of encryption and decryption.
- Two way
  - This way like using SSL or SSH connection can be encrypted and decrypted with proper key.
  - MySQL supports encrypted connections between clients and the server using **the TLS (Transport Layer Security) protocol**. TLS is sometimes referred to as SSL (Secure Sockets Layer) but MySQL does not actually use the SSL protocol for encrypted connections because its encryption is weak
- One way
  - This way like md5 or sha algorithm only can be encrypted.
- What is the difference between 1 way and 2 way encryption?
  - One-way encryption **stores the password as a secure hash value** that cannot be decrypted.

# Functions for two way

- `ENCODE(str, pass_str), DECODE(crpty, pass_str)`
  - `pass_str`: is a key value
  - `crpty` need to be a BLOB type.
- `AES_ENCRYPT(str,key_str), AES_DECRYPT(crypt_str,key_str)`
- `DES_ENCRYPT(), DES_DECRYPT()`

# ENCODE()

- MySQL ENCODE() function encrypts a string. This function returns a binary string of the same length of the original string.

## Syntax:

```
ENCODE(str, pass_str);
```

## Arguments

Name	Description
str	A string which is to be encrypted.
pass_str	A string to encrypt str.

## Example:

### Code:

```
1 SELECT ENCODE('mytext','mykeyststring');
```

### Explanation

The above MySQL statement will encrypt the string 'mytext' with 'mykeyststring'.

### Sample Output:

```
mysql> SELECT ENCODE('mytext','mykeyststring');
+-----+
| ENCODE('mytext','mykeyststring') |
+-----+
| ">ÿiË                               |
+-----+
1 row in set (0.00 sec)
```

# Example

- Table: testtable



Table structure



Relation view

#	Name	Type	Collation	Attributes
<input type="checkbox"/>	1 description	blob		

```
INSERT INTO testtable  
VALUE  
(ENCODE('myencodetext', 'mypassw'));
```

```
SELECT * FROM `testtable`
```



Show all

Number of rows

+ Options

**description**

[BLOB - 12 B]

啥x9Dv ] 泣



Sample table: testtable

description
^[@ú~,IšžÝy
Oœ??^]Z_!_<m
~(o 2¤@QŠ!''jD,=ET£9Z!
^[@£~,IsØY?y
Oo??^]Z_Ý_<m

# DECODE()

- MySQL DECODE() function decodes an encoded string and returns the original string.

### Syntax:

```
DECODE(crypt_str, pass_str);
```

## Arguments

Name	Description
crypt_str	An encoded string.
pass_str	The password string to decode crypt_str.

### Example:

Code:

```
1 SELECT DECODE(ENCODE('mytext','mykeystring'),'mykeystring');
```

Sample Output:

[illegible]

# Example

```
SELECT description, DECODE(description, 'mypassw')  
FROM testtable;
```

description	DECODE(description, 'mypassw')
[BLOB - 12 B]	[BLOB - 12 B]

```
SELECT description, convert( DECODE(description, 'mypassw') using utf8) FROM testtable;
```

☐ Profiling [ [Edit inline](#) ] [ [Edit](#) ] [ [Explain SQ](#) ]

☐ Show all | Number of rows: 25  Filter rows:

+ Options

description	convert( DECODE(description, 'mypassw') using utf8)
[BLOB - 12 B]	myencodetext

Why?



# Example – wrong key

```
SELECT description, convert(Decode(description,'mypw') using utf8) FROM testtable;
```

☐ Profiling [ [Edit inline](#) ] [ [Edit](#) ] [ [Explain](#) ]

☐ Show all | Number of rows: 25  Filter rows:

+ Options

description	convert(Decode(description,'mypw') using utf8)
-------------	--

[BLOB - 12 B]	@(?Zy?:?X?q
---------------	-------------

# BLOB type in MySQL

- A BLOB is a binary large object that can hold a variable amount of data. ... BLOB values are treated as binary strings (byte strings).
- They have the binary character set and collation, and comparison and sorting are based on the numeric values of the bytes in column values.
  - What is MySQL collation? A collation is **a set of rules that defines how to compare and sort character strings**. Each collation in MySQL belongs to a single character set. Every character set has at least one collation, and most have two or more collations. A collation orders characters based on weights.
  - What is the default collation in MySQL? latin1. The default MySQL server character set and collation are **latin1 and latin1\_swedish\_ci** , but you can specify character sets at the server, database, table, column, and string literal levels.
  - What is the difference between collation and character set? A character set is a set of characters while **a collation is the rules for comparing and sorting a particular character set**. For example, a subset of a character set could consist of the letters A , B and C . A default collation could define these as appearing in an ascending order of A, B, C .

# BLOB type in MySQL

- Why BLOB is used in MySQL? A BLOB column **stores actual binary strings or byte strings in a MySQL database instead of a file path reference**. This has one advantage; when you back up your database, your entire application data is copied over.
- Should I use BLOB? The reason to use BLOBs is quite simply **manageability** - you have exactly one method to back and restore up the database, you can easily do incremental backups, there is zero risk of the image and its meta data stored in DB tables ever getting out of sync, you also have one programming interface to run queries or load....
- When should I use BLOB? Also another advantage of storing files in BLOB fields is that they can be accessed more efficiently than files on the disk (there is no need for directory traversal, open, read, close). If you are **planning to store lots of files** in MYSQL, it's usually a good practice to have the files stored in a separate table.

# BLOB type in MySQL

- If you on occasion need to retrieve an image and it has to be available on several different web servers. But I think that's pretty much it.
  - If it doesn't have to be available on several servers, it's always better to put them in the file system.
  - If it has to be available on several servers and there's actually some kind of load in the system, you'll need some kind of distributed storage.

# AES\_ENCRYPT()

- This function encrypts a string using AES algorithm.
  - AES stands for Advance Encryption Standard. This function encodes the data with 128 bits key length but it can be extended up to 256 bits key length. It encrypts a string and returns a binary string.

### Syntax:

```
AES_ENCRYPT(str, key_str);
```

## Arguments

Name	Description
str	A string which will be encrypted.
key_str	String to encrypt str.

**Example:**

Code:

```
1 SELECT AES_ENCRYPT('mytext', 'mykeyststring');
```

### Explanation

The above MySQL statement encrypts the string 'mytext' with key myteststring.

Sample Output:

```
mysql> SELECT AES_ENCRYPT('mytext', 'mykeyststring');
+-----+
| AES_ENCRYPT('mytext', 'mykeyststring') |
+-----+
| •>"í fǒbáò9•j |
+-----+
1 row in set (0.00 sec)
```

# Example

---

```
DELETE FROM testtable;
```

```
INSERT INTO  
testtable VALUE(AES_ENCRYPT('mytext','passw'));
```

**description**

[BLOB - 16 B]

# AES\_DECRYPT()

- This function decrypts an encrypted string using AES algorithm to return the original string.

## Syntax:

```
AES_DECRYPT(encrypt_str, key_str);
```

## Arguments

Name	Description
encrypt_str	An encrypted string.
key_str	String to use to decrypt encrypt_str.

# Example

```
SELECT description,  
        AES_DECRYPT(description,'passw')  
FROM testtable;
```

description	AES_DECRYPT(description,'passw')
[BLOB - 16 B]	[BLOB - 6 B]

```
SELECT description,  
        convert (AES_DECRYPT(description,'passw')using utf8)  
FROM testtable;
```

description	convert (AES_DECRYPT(description,'passw')using utf8)
[BLOB - 16 B]	mytext



# DES\_ENCRYPT()

- This function encrypts a string with a key Triple-DES algorithm.
- This function works only with Secure Sockets Layer (SSL) if support for SSL is available in MySql configuration.

## Syntax:

```
DES_ENCRYPT(str, [{key_num | key_str}]);
```

## Arguments

Name	Description
str	A string which is to be encrypted.
key_num	A number from 0 to 9 from DES key file.
key_str	A string to encrypt str.

# Example

- The MySQL statement encrypts the string mytext with key number 5; for the second instance of the function, mytext is encrypted with mypassword.

---

```
DELETE FROM testtable;
```

Sample Output:

```
mysql> SELECT DES_ENCRYPT('mytext',5),DES_ENCRYPT('mytext','mypassword');
+-----+-----+
| DES_ENCRYPT('mytext',5) | DES_ENCRYPT('mytext','mypassword') |
+-----+-----+
| ...ÿc}æð~           | ÿ]ixñ"Å                          |
+-----+-----+
1 row in set (0.00 sec)
```

# Example

```
INSERT INTO testtable  
VALUE(DES_ENCRYPT('mydesencrypttext','mydespassw'));
```

description

[BLOB - 16 B]

[BLOB - 25 B]

```
INSERT INTO testtable  
VALUE(DES_ENCRYPT('mydesencrypttext',5));
```

description

[BLOB - 16 B]

[BLOB - 25 B]

[BLOB - 25 B]

DES\_DECRYPT()

# Types of Databases

- A company should use the type of database that fit in its requirements and needs. There are various types of database structures:
  - **Relational databases**-Relational databases have been around since the 1970s. The name comes from the way that data is stored in multiple, related tables. Within the tables, data is stored in rows and columns. Organizations that have a lot of **unstructured** or **semi-structured** data should not be considering a relational database. (Examples of unstructured data are: Rich media. Media and entertainment data, surveillance data, geo-spatial data, audio, weather data. Semi Structured Data Examples: Email 、 CSV 、 XML and JSON documents 、 HTML, and so on)
    - Microsoft SQL Server, Oracle Database, MySQL, PostgreSQL and IBM DB2

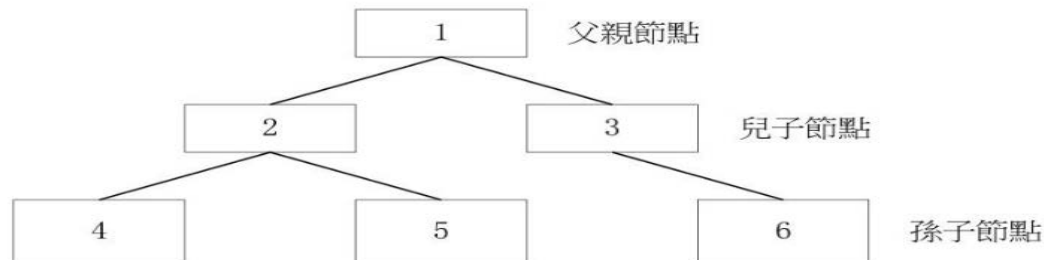
# Types of Databases

- A company should use the type of database that fit in its requirements and needs. There are various types of database structures: (**We just introduce few types here!**)
  - **NoSQL** is a broad category that includes any database that doesn't use SQL as its primary data access language.
    - These types of databases are also sometimes referred to as non-relational databases. Unlike in relational databases, data in a NoSQL database doesn't have to conform to a pre-defined schema, so these types of databases are great for organizations seeking to store **unstructured** or **semi-structured** data.
    - One advantage of NoSQL databases is that developers can make changes to the database on the fly, without affecting applications that are using the database.
    - Apache Cassandra, MongoDB, CouchDB, and CouchBase

# Types of Databases

- A company should use the type of database that fit in its requirements and needs. There are various types of database structures: (**We just introduce few types here!**)
  - **Hierarchical databases** use a parent-child model to store data. If you were to draw a picture of a hierarchical database, it would look like a family tree, with one object on top branching down to multiple objects beneath it.
  - **Examples:** IBM Information Management System (IMS), Windows Registry

■ 除了樹根以外的節點均只有一個直屬的「父親」節點



# Types of Databases

- A company should use the type of database that fit in its requirements and needs. There are various types of database structures: (**We just introduce few types here!**)
  - **Document databases** Document databases, also known as document stores, use JSON-like documents to model data instead of rows and columns.
    - Sometimes referred to as document-oriented databases, document databases are designed to store and manage document-oriented information, also referred to as semi-structured data. Document databases are simple and scalable, making them useful for mobile apps that need fast iterations.
    - **Examples:** MongoDB, Amazon DocumentDB, Apache CouchDB



# Why SQL Statements

# What is a Relational Database (RDBMS)

- A relational database is a type of database that stores and provides access to data points that are related to one another.
- Relational databases are based on the relational model, an intuitive, straightforward way of representing data in tables.
- In a relational database, each row in the table is a record with a unique ID called the key. The columns of the table hold attributes of the data, and each record usually has a value for each attribute, making it easy to establish the relationships among data points.

# Industry's Best RDBMS

- Oracle database products offer customers cost-optimized and high-performance versions of Oracle Database, the world's leading converged, multi-model database management system, as well as in-memory, NoSQL and MySQL databases.
- Oracle Autonomous Database enables customers to simplify relational database environments and reduce management workloads.
- Good procedural computer language: PL/ SQL



# Benefits of RDBMS

- The simple yet powerful relational model is used by organizations of all types and sizes for a broad variety of information needs.
- Relational databases are used to track inventories, process ecommerce transactions, manage huge amounts of mission-critical customer information, and much more.
- A relational database can be considered for any information need in which data points relate to each other and must be managed in a secure, rules-based, consistent way.
- Relational databases have been around since the 1970s. Today, the advantages of the relational model continue to make it the most widely accepted model for databases.

# Relational model and data consistency

- The relational model is the best at maintaining data consistency across applications and database copies (called instances). For example, when a customer deposits money at an ATM and then looks at the account balance on a mobile phone, the customer expects to see that deposit reflected immediately in an updated account balance. Relational databases excel at this kind of data consistency, ensuring that multiple instances of a database have the same data all the time.
- It's difficult for other types of databases to maintain this level of **timely consistency** with large amounts of data. Some recent databases, such as NoSQL, can supply only “**eventual consistency**.” Under this principle, when the database is scaled or when multiple users access the same data at the same time, the data needs some time to “catch up.” Eventual consistency is acceptable for some uses, such as to maintain listings in a product catalog, but for critical business operations such as shopping cart transactions, the relational database is still the gold standard.

# The Differences between MySQL Workbench VS phpMyAdmin

Feature	MySQL Workbench	phpMyAdmin
Database	✓	✓
GUI Designer	✓	✓
Server Management	✓	✓
Spell checking	✓	✓
Data export/import	✓	✓
Autocompletion	✓	✓
Database Management	✓	✓
ER Diagrams	✓	✓
Lightweight	✓	✓
Data Report Wizard	✓	✓
MySQL Support	✓	✓
User interface	✓	✓
PostgreSQL support	✓	✓
Schema editor	✓	✓
Import CSV data	✓	✓

# The Differences between MySQL Workbench VS phpMyAdmin

Feature	MySQL Workbench	phpMyAdmin
Data-management	✓	✓
Night mode/Dark Theme	✓	✓
Local based GUI	✓	✗
MariaDB	✓	✓
Backend	✗	✓
Charts	✗	✓
Code completion	✗	✓
Intellisense	✗	✓
Backup	✗	✓
Built-in editor	✗	✓
PhpMyAdmin	✗	✓

Q&A