

Full Stack Web Development

Advanced Database Design and Development

Job Connector Program

Database design can be generally defined as a collection of tasks or processes that enhance the designing, development, implementation, and maintenance of enterprise data management system. Designing a proper database reduces the maintenance cost thereby improving data consistency and the cost-effective measures are greatly influenced in terms of disk storage space. Therefore, there has to be a brilliant concept of designing a database. The designer should follow the constraints and decide how the elements correlate and what kind of data must be stored.

What is good database design?

A good database design is, therefore, one that:

- Divides your information into subject-based tables to reduce redundant data.
- Provides Access with the information it requires to join the information in the tables together as needed.
- Helps support and ensure the accuracy and integrity of your information.
- Accommodates your data processing and reporting needs.

The design process consists of the following steps:

- **Determine the purpose of your database** – This helps prepare you for the remaining steps.
- **Find and organize the information required** – Gather all of the types of information you might want to record in the database, such as product name and order number.
- **Divide the information into tables** – Divide your information items into major entities or subjects, such as Products or Orders. Each subject then becomes a table.
- **Turn information items into columns** – Decide what information you want to store in each table. Each item becomes a field, and is displayed as a column in the table. For example, an Employees table might include fields such as Last Name and Hire Date.

- **Specify primary keys** – Choose each table's primary key. The primary key is a column that is used to uniquely identify each row. An example might be Product ID or Order ID.
- **Set up the table relationships** – Look at each table and decide how the data in one table is related to the data in other tables. Add fields to tables or create new tables to clarify the relationships, as necessary.
- **Refine your design** – Analyze your design for errors. Create the tables and add a few records of sample data. See if you can get the results you want from your tables. Make adjustments to the design, as needed.
- **Apply the normalization rules** – Apply the data normalization rules to see if your tables are structured correctly. Make adjustments to the tables, as needed.

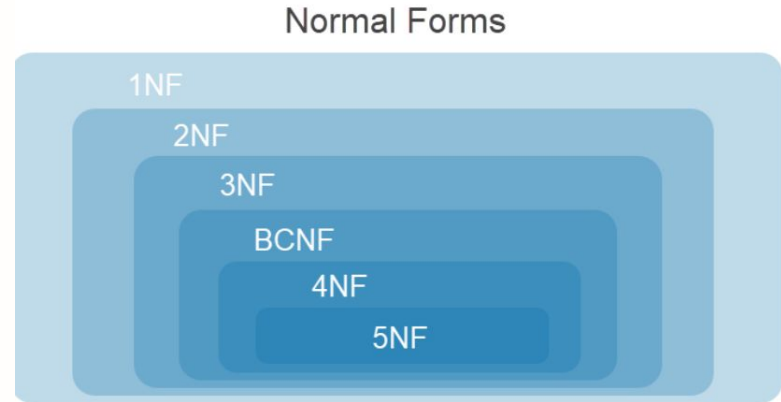
The elementary concepts used in database normalization are:

- **Keys**. Column attributes that identify a database record uniquely.
- **Functional Dependencies**. Constraints between two attributes in a relation.
- **Normal Forms**. Steps to accomplish a certain quality of a database.

Database Normal Form

Normalizing a database is achieved through a set of rules known as normal forms. The central concept is to help a database designer achieve the desired quality of a relational database.

A database is normalized when it fulfills the third normal form. Further steps in normalization make the database design complicated and could compromise the functionality of the system.



Stage of Normalization

Stage	Redundancy Anomalies Addressed
Unnormalized Form (UNF)	The state before any normalization. Redundant and complex values are present.
First Normal Form (1NF)	Repeating and complex values split up, making all instances atomic.
Second Normal Form (2NF)	Partial dependencies decompose to new tables. All rows functionally depend on the primary key.
Third Normal Form (3NF)	Transitive dependencies decompose to new tables. Non-key attributes depend on the primary key.
Boyce-Codd Normal Form (BCNF)	Transitive and partial functional dependencies for all candidate keys decompose to new tables.
Fourth Normal Form (4NF)	Removal of multivalued dependencies.
Fifth Normal Form (5NF)	Removal of JOIN dependencies.

What is a KEY?

A database key is an attribute or a group of features that uniquely describes an entity in a table. The types of keys used in normalization are:

- **Super Key.** A set of features that uniquely define each record in a table.
- **Candidate Key.** Keys selected from the set of super keys where the number of fields is minimal.
- **Primary Key.** The most appropriate choice from the set of candidate keys serves as the table's primary key.
- **Foreign Key.** The primary key of another table.
- **Composite Key.** Two or more attributes together form a unique key but are not keys individually.

Database Key – Super Key

employeeID	name	age	email
1	Adam A.	30	adam.a@email.com
2	Jacob J.	27	jacob.j@email.com
3	David D.	35	david.d@email.com

Some examples of super keys in the table are:

- employeeID
- (employeeID, name)
- email

All super keys can serve as a unique identifier for each row. On the other hand, the employee's name or age are not unique identifiers because two people could have the same name or age.

Database Key – Candidate Key

employeeID	name	age	email
1	Adam A.	30	adam.a@email.com
2	Jacob J.	27	jacob.j@email.com
3	David D.	35	david.d@email.com

The candidate keys come from the set of super keys where the number of fields is minimal. The choice comes down to two options:

- employeeID
- email

Both options contain a minimal number of fields, making them optimal candidate keys.

Database Key – Candidate Key

employeeID	name	age	email
1	Adam A.	30	adam.a@email.com
2	Jacob J.	27	jacob.j@email.com
3	David D.	35	david.d@email.com

The candidate keys come from the set of super keys where the number of fields is minimal. The choice comes down to two options:

- employeeID
- email

Both options contain a minimal number of fields, making them optimal candidate keys.

Database Key – Primary Key

- The Primary keys constraint uniquely identifies each record in a database table.
- Primary keys must contain UNIQUE values, and cannot contain NULL values.
- A table can have only one primary key, which may consist of single or multiple fields.

Example: No_KTP, product_key, ID_sidikjari

Database Key – Foreign Key

A foreign key is a field or a column that is used to establish a link between two tables. In simple words you can say that, a foreign key in one table used to point primary key in another table.

Foreign Key

First table:

S_Id	LastName	FirstName	CITY
1	MAURYA	AJEET	ALLAHABAD
2	JAISWAL	RATAN	GHAZIABAD
3	ARORA	SAUMYA	MODINAGAR

Second table:

O_Id	OrderNo	S_Id
1	99586465	2
2	78466588	2
3	22354846	3
4	57698656	1

The "S_Id" column in the 1st table is the PRIMARY KEY in the 1st table.

The "S_Id" column in the 2nd table is a FOREIGN KEY in the 2nd table.

A **JOIN** clause is used to combine rows from two or more tables, based on a related column between them.

Join Table

Let's look at a selection from the "Orders" table:

OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

Then, look at a selection from the "Customers" table:

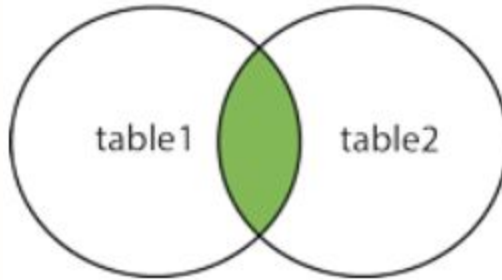
CustomerID	CustomerName	ContactName	Country
1	Alfreds Futterkiste	Maria Anders	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mexico

Join Table

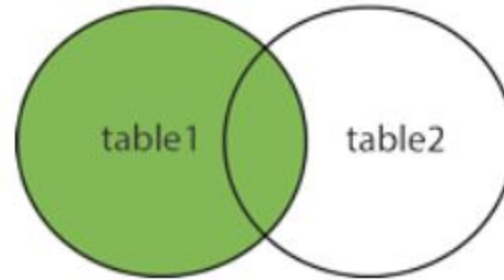
Notice that the "CustomerID" column in the "Orders" table refers to the "CustomerID" in the "Customers" table. The relationship between the two tables above is the "CustomerID" column.

Type of SQL Join

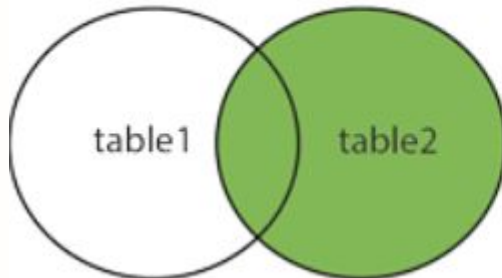
INNER JOIN



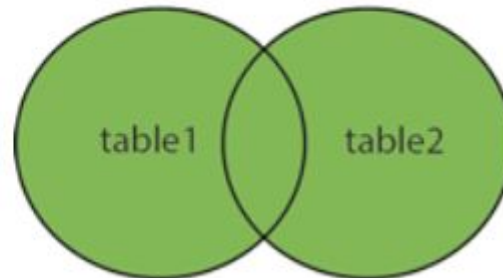
LEFT JOIN



RIGHT JOIN



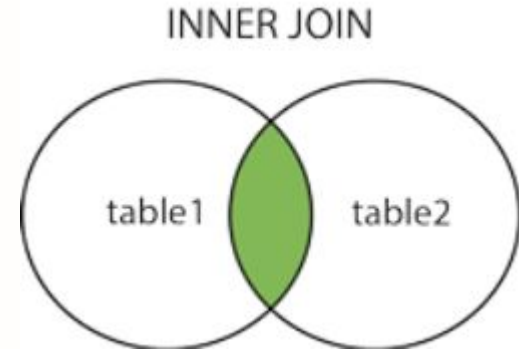
FULL OUTER JOIN



Inner Join

The INNER JOIN keyword selects records that have matching values in both tables.

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```



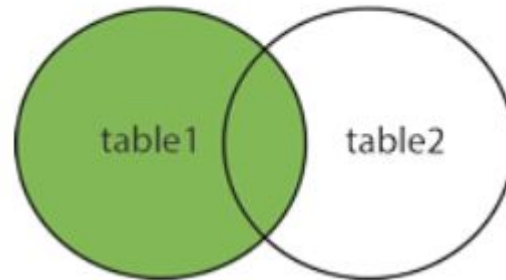
Left Join

The LEFT JOIN keyword returns all records from the left table (table1), and the matching records from the right table (table2). The result is 0 records from the right side, if there is no match.



```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

LEFT JOIN



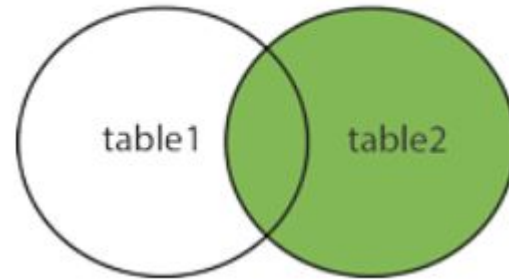
Right Join

The RIGHT JOIN keyword returns all records from the right table (table2), and the matching records from the left table (table1). The result is 0 records from the left side, if there is no match.



```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```

RIGHT JOIN

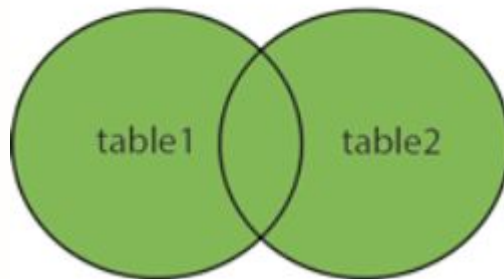


The FULL OUTER JOIN keyword returns all records when there is a match in left (table1) or right (table2) table records.

Tip: FULL OUTER JOIN and FULL JOIN are the same.

```
SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2
ON table1.column_name = table2.column_name;
WHERE condition
```

FULL OUTER JOIN



Aggregate Function

An aggregate function in SQL performs a calculation on multiple values and returns a single value. SQL provides many aggregate functions that include avg, count, sum, min, max, etc. An aggregate function ignores NULL values when it performs the calculation, except for the count function.

An aggregate function in SQL returns one value after calculating multiple values of a column. We often use aggregate functions with the GROUP BY and HAVING clauses of the SELECT statement.

Various types of SQL aggregate functions are:

- Count()
- Sum()
- Avg()
- Min()
- Max()

Aggregate Function – Count

The COUNT() function returns the number of rows in a database table



```
COUNT(*)
```

```
COUNT( [ALL|DISTINCT] expression )
```



```
SELECT COUNT(product_id)  
FROM Products;
```

```
COUNT(product_id)
```

```
10
```

product_id	name	quantity_in_stock	unit_price
1	Foam Dinner Plate	70	1.21
2	Pork - Bacon,back Peameal	49	4.65
3	Lettuce - Romaine, Heart	38	3.35
4	Brocolinni - Gaylan, Chinese	90	4.53
5	Sauce - Ranch Dressing	94	1.63
6	Petit Baguette	14	2.39
7	Sweet Pea Sprouts	98	3.29
8	Island Oasis - Raspberry	26	0.74
9	Longan	67	2.26
10	Broom - Push	6	1.09

Aggregate Function – Sum

The SUM() function returns the total sum of a numeric column.



```
SUM( )
```

```
SUM( [ALL|DISTINCT] expression )
```



```
SELECT SUM(unit_price)  
FROM Products;
```

product_id	name	quantity_in_stock	unit_price
1	Foam Dinner Plate	70	1.21
2	Pork - Bacon,back Peameal	49	4.65
3	Lettuce - Romaine, Heart	38	3.35
4	Brocolinni - Gaylan, Chinese	90	4.53
5	Sauce - Ranch Dressing	94	1.63
6	Petit Baguette	14	2.39
7	Sweet Pea Sprouts	98	3.29
8	Island Oasis - Raspberry	26	0.74
9	Longan	67	2.26
10	Broom - Push	6	1.09

Aggregate Function – Avg

The AVG() function calculates the average of a set of values.



```
AVG( )
```

```
AVG( [ALL|DISTINCT] expression )
```



```
SELECT AVG(quantity_in_stock)  
FROM Products;
```

product_id	name	quantity_in_stock	unit_price
1	Foam Dinner Plate	70	1.21
2	Pork - Bacon,back Peameal	49	4.65
3	Lettuce - Romaine, Heart	38	3.35
4	Brocolinni - Gaylan, Chinese	90	4.53
5	Sauce - Ranch Dressing	94	1.63
6	Petit Baguette	14	2.39
7	Sweet Pea Sprouts	98	3.29
8	Island Oasis - Raspberry	26	0.74
9	Longan	67	2.26
10	Broom - Push	6	1.09

Aggregate Function – Min

The MIN() aggregate function returns the lowest value (minimum) in a set of non-NULL values.



```
MIN( )
```

```
MIN( [ALL|DISTINCT] expression )
```



```
SELECT MIN(quantity_in_stock)
FROM Products;
```

product_id	name	quantity_in_stock	unit_price
1	Foam Dinner Plate	70	1.21
2	Pork - Bacon,back Peameal	49	4.65
3	Lettuce - Romaine, Heart	38	3.35
4	Brocolinni - Gaylan, Chinese	90	4.53
5	Sauce - Ranch Dressing	94	1.63
6	Petit Baguette	14	2.39
7	Sweet Pea Sprouts	98	3.29
8	Island Oasis - Raspberry	26	0.74
9	Longan	67	2.26
10	Broom - Push	6	1.09

Aggregate Function – Max

The MAX() aggregate function returns the highest value (maximum) in a set of non-NULL values.



```
MAX( * )
```

```
MAX( [ALL|DISTINCT] expression )
```



```
SELECT MAX(quantity_in_stock)
FROM Products;
```

product_id	name	quantity_in_stock	unit_price
1	Foam Dinner Plate	70	1.21
2	Pork - Bacon,back Peameal	49	4.65
3	Lettuce - Romaine, Heart	38	3.35
4	Brocolinni - Gaylan, Chinese	90	4.53
5	Sauce - Ranch Dressing	94	1.63
6	Petit Baguette	14	2.39
7	Sweet Pea Sprouts	98	3.29
8	Island Oasis - Raspberry	26	0.74
9	Longan	67	2.26
10	Broom - Push	6	1.09

Aggregate Function with Group By and Having – Count

customer_id	first_name	last_name	birth_date	phone	address	city	state	points
1	Alfred	MacCaffrey	1986-03-28	781-932-9754	0 Sage Terrace	Hampton	MA	2273
2	Ines	Brushfield	1986-04-13	804-427-9456	14187 Commercial Trail	Hampton	VA	947
3	Freddi	Boagey	1985-02-07	719-724-7869	251 Springs Junction	Colorado Springs	CO	2967
4	Ambur	Roseburgh	1974-04-14	407-231-8017	30 Arapahoe Terrace	Orlando	FL	457
5	Clemmie	Betchley	1973-11-07	NULL	5 Spohn Circle	Arlington	TX	3675
6	Elka	Twiddell	1991-09-04	312-480-8498	7 Manley Drive	Chicago	IL	3073
7	Ilene	Dowson	1964-08-30	615-641-4759	50 Lillian Crossing	Nashville	TN	1672
8	Thacher	Naseby	1993-07-17	941-527-3977	538 Mosinee Center	Sarasota	FL	205
9	Romola	Rumgay	1992-05-23	559-181-3744	3520 Ohio Trail	Visalia	CA	1486
10	Levy	Mynett	1969-10-13	404-246-3370	68 Lawn Avenue	Atlanta	GA	796
11	Sam	Tully	NULL	NULL	16 commecial centre	New York	TX	2600



```
SELECT COUNT(customer_id), city FROM Customers GROUP BY city HAVING COUNT(customer_id) > 0;
```

Aggregate Function with Group By and Having – Count

COUNT(customer_id)	city
2	Hampton
1	Colorado Springs
1	Orlando
1	Arlington
1	Chicago
1	Nashville
1	Sarasota
1	Visalia
1	Atlanta
2	New York



```
SELECT COUNT(customer_id), city FROM Customers GROUP BY city HAVING COUNT(customer_id) > 0;
```


Aggregate Function with Group By and Having – Count


customer_id	first_name	last_name	birth_date	phone	address	city	state	points
1	Alfred	MacCaffrey	1986-03-28	781-932-9754	0 Sage Terrace	Hampton	MA	2273
2	Ines	Brushfield	1986-04-13	804-427-9456	14187 Commercial Trail	Hampton	VA	947
3	Freddi	Boagey	1985-02-07	719-724-7869	251 Springs Junction	Colorado Springs	CO	2967
4	Ambur	Roseburgh	1974-04-14	407-231-8017	30 Arapahoe Terrace	Orlando	FL	457
5	Clemmie	Betchley	1973-11-07	NULL	5 Spohn Circle	Arlington	TX	3675
6	Elka	Twiddell	1991-09-04	312-480-8498	7 Manley Drive	Chicago	IL	3073
7	Ilene	Dowson	1964-08-30	615-641-4759	50 Lillian Crossing	Nashville	TN	1672
8	Thacher	Naseby	1993-07-17	941-527-3977	538 Mosinee Center	Sarasota	FL	205
9	Romola	Rumgay	1992-05-23	559-181-3744	3520 Ohio Trail	Visalia	CA	1486
10	Levy	Mynett	1969-10-13	404-246-3370	68 Lawn Avenue	Atlanta	GA	796
11	Sam	Tully	NULL	NULL	16 commecial centre	New York	TX	2600
12	Brian	Ross	NULL	NULL	43 Bakers Street	New York	CO	1435



```
SELECT COUNT(customer_id), city FROM Customers GROUP BY city HAVING SUM(points) > 3000;
```

Aggregate Function with Group By and Having - Count

COUNT(customer_id)	city
2	Hampton
1	Arlington
1	Chicago
2	New York



```
SELECT COUNT(customer_id), city FROM Customers GROUP BY city HAVING SUM(points) > 3000;
```

-
- A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.
 - A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.
 - Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

Subquery Example



```
SELECT * FROM CUSTOMERS WHERE ID IN  
(SELECT ID FROM CUSTOMERS WHERE SALARY > 45000);
```



```
SELECT column_name [, column_name ] FROM table1 [, table2 ]  
WHERE column_name OPERATOR  
(SELECT column_name [, column_name ] FROM table1 [, table2 ]  
[WHERE])
```

A **transaction** is a unit of work that is performed against a database. Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program.

A transaction is the propagation of one or more changes to the database. For example, if you are creating a record or updating a record or deleting a record from the table, then you are performing a transaction on that table. It is important to control these transactions to ensure the data integrity and to handle database errors.

Practically, you will club many SQL queries into a group and you will execute all of them together as a part of a transaction.

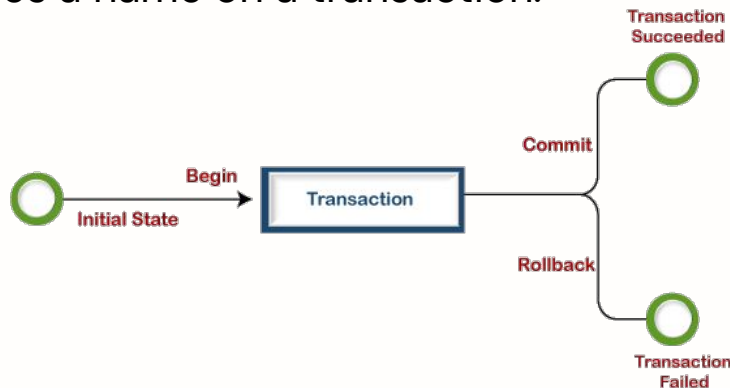
SQL Transaction – Control

COMMIT – to save the changes.

ROLLBACK – to roll back the changes.

SAVEPOINT – creates points within the groups of transactions in which to ROLLBACK.

SET TRANSACTION – Places a name on a transaction.

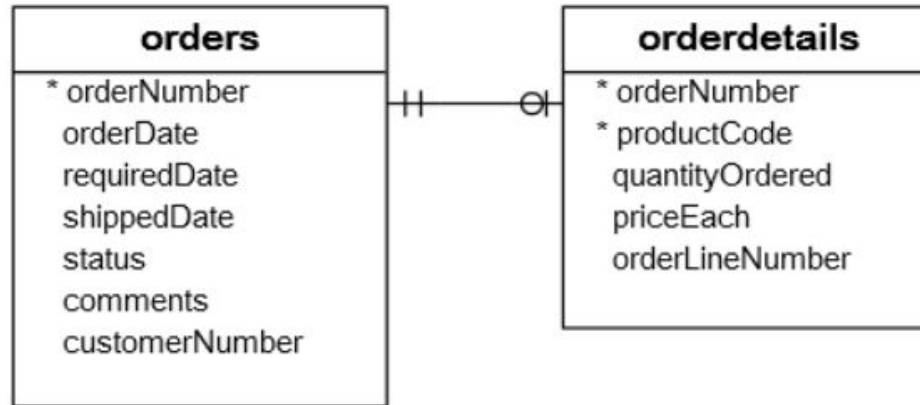


MySQL provides us with the following important statement to control transactions:

- To start a transaction, you use the `START TRANSACTION` statement. The `BEGIN` or `BEGIN WORK` are the aliases of the `START TRANSACTION`.
- To commit the current transaction and make its changes permanent, you use the `COMMIT` statement.
- To roll back the current transaction and cancel its changes, you use the `ROLLBACK` statement.
- To disable or enable the auto-commit mode for the current transaction, you use the `SET autocommit` statement.

MySQL Transaction Example

We will use the orders and orderDetails table for the demonstration



The following illustrates the step of creating a new sales order:

- First, start a transaction by using the `START TRANSACTION` statement.
- Next, select the latest sales order number from the orders table and use the next sales order number as the new sales order number.
- Then, insert a new sales order into the orders table.
- After that, insert sales order items into the orderdetails table.
- Finally, commit the transaction using the `COMMIT` statement.
- Optionally, you can select data from both orders and orderdetails tables to check the new sales order.

MySQL Transaction Commit Example

```
-- 1. start a new transaction
START TRANSACTION;
-- 2. Get the latest order number
SELECT @orderNumber:=MAX(orderNumber)+1 FROM orders;
-- 3. insert a new order for customer 145
INSERT INTO orders(orderNumber,
                  orderDate,
                  requiredDate,
                  shippedDate,
                  status,
                  customerNumber)
VALUES(@orderNumber,'2005-05-31','2005-06-10','2005-06-11','In Process',145);
-- 4. Insert order line items
INSERT INTO orderdetails(orderNumber,
                       productCode,
                       quantityOrdered,
                       priceEach,
                       orderLineNumber)
VALUES(@orderNumber,'S18_1749', 30, '136', 1),
      (@orderNumber,'S18_2248', 50, '55.09', 2);
-- 5. commit changes
COMMIT;
```

MySQL Transaction Rollback Example

First, log in to the MySQL database server and delete data from the orders table. As you can see from the output, MySQL confirmed that all the rows from the orders table were deleted.



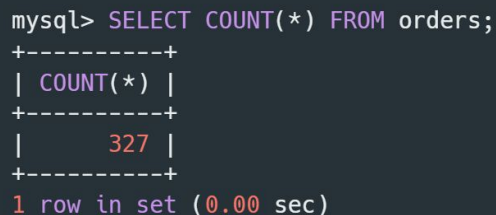
```
mysql> START TRANSACTION;  
Query OK, 0 rows affected (0.00 sec)  
  
mysql> DELETE FROM orders;  
Query OK, 327 rows affected (0.03 sec)
```

MySQL Transaction Rollback Example

Second, log in to the MySQL database server in a separate session and query data from the orders table. In this second session, we still can see the data from the orders table.

We have made the changes in the first session. However, the changes are not permanent. In the first session, we can either commit or roll back the changes.

For the demonstration purpose, we will roll back the changes in the first session.



```
mysql> SELECT COUNT(*) FROM orders;
+-----+
| COUNT(*) |
+-----+
|      327 |
+-----+
1 row in set (0.00 sec)
```




```
mysql> ROLLBACK;
Query OK, 0 rows affected (0.04 sec)
```

The **CREATE INDEX** statement is used to create indexes in tables.

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

Create Index Syntax,


Creates an index on a table. Duplicate values are allowed:



```
CREATE INDEX index_name  
ON table_name (column1, column2, ...);
```

Create Unique Index Syntax,

Creates an unique index on a table. Duplicate values are allowed:



```
CREATE UNIQUE INDEX index_name  
ON table_name (column1, column2, ...);
```

SQL Indexing – Example



```
CREATE INDEX idx_lastname  
ON Persons (LastName);  
  
CREATE INDEX idx_pname  
ON Persons (LastName, FirstName);
```


Thank You!

